The Z Shell Manual

Version 4.0.3
Updated October 24, 2001

Original documentation by Paul Falstad

This is a texinfo version of the documentation for the Z Shell, originally by Paul Falstad.

1 The Z Shell Manual

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

1.1 Producing documentation from zsh.texi

The texinfo source may be converted into several formats:

The Info manual
: 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 manual
: 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 manual.

The HTML manual
: An HTML version of this manual is available at the Zsh web site via:

  `http://zsh.sunsite.dk/Doc/`.

  (The HTML version is produced with *texi2html*, which may be obtained from `http://www.mathematik.uni-kl.de/~obachman/Texi2html/`. The command is '`texi2html -split chapter -expand info zsh.texi`'. If necessary, upgrade to version 1.64 of texi2html.)

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 2 for a list of sites.)

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 `<pf@zsh.org>`. Zsh is now maintained by the members of the zsh-workers mailing list `<zsh-workers@sunsite.dk>`. The development is currently coordinated by Peter Stephenson `<pws@zsh.org>`. The coordinator can be contacted at `<coordinator@zsh.org>`, but matters relating to the code should generally go to the mailing list.

2.2 Availability

Zsh is available from the following anonymous FTP sites. These mirror sites are kept frequently up to date. The sites marked with *(H)* may be mirroring `ftp.cs.elte.hu` instead of the primary site.

Primary site
: `ftp://ftp.zsh.org/pub/zsh/`
`http://www.zsh.org/pub/zsh/`

Australia
: `ftp://ftp.zsh.org/pub/zsh/`
`http://www.zsh.org/pub/zsh/`
`ftp://ftp.ips.gov.au/pub/packages/zsh/` *(H)*

Denmark
: `ftp://sunsite.dk/pub/unix/shells/zsh/`

Finland
: `ftp://ftp.funet.fi/pub/unix/shells/zsh/`

France
: `ftp://ftp.cenatls.cena.dgac.fr/shells/zsh/`

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

Hungary
: `ftp://ftp.cs.elte.hu/pub/zsh/`
`http://www.cs.elte.hu/pub/zsh/`
`ftp://ftp.kfki.hu/pub/packages/zsh/`

Israel
: `ftp://ftp.math.technion.ac.il/pub/zsh/`
`http://www.math.technion.ac.il/pub/zsh/`

Italy
: `ftp://ftp.unina.it/pub/Unix/pkgs/shell/zsh/`

Japan
: `ftp://ftp.nisiq.net/pub/shells/zsh/` *(H)*
`ftp://ftp.win.ne.jp/pub/shell/zsh/`

Norway
: `ftp://ftp.uit.no/pub/unix/shells/zsh/`

Poland
: `ftp://sunsite.icm.edu.pl/pub/unix/shells/zsh/`

Romania
: `ftp://ftp.roedu.net/pub/mirrors/ftp.zsh.org/pub/zsh/`
`ftp://ftp.kappa.ro/pub/mirrors/ftp.zsh.org/pub/zsh/`

Slovenia
: `ftp://ftp.siol.net/mirrors/zsh/`

Sweden
: `ftp://ftp.lysator.liu.se/pub/unix/zsh/`

UK
: `ftp://ftp.net.lut.ac.uk/zsh/`
`ftp://sunsite.org.uk/packages/zsh/`

USA
: `ftp://uiarchive.uiuc.edu/pub/packages/shells/zsh/`
`ftp://ftp.rge.com/pub/shells/zsh/`
`ftp://foad.org/pub/zsh/`
`http://foad.org/zsh/`

2.3 Mailing Lists

Zsh has 3 mailing lists:

`<zsh-announce@sunsite.dk>`
: Announcements about releases, major changes in the shell and the monthly posting of the Zsh FAQ. (moderated)

`<zsh-users@sunsite.dk>`
: User discussions.

`<zsh-workers@sunsite.dk>`
: Hacking, development, bug reports and patches.

To subscribe or unsubscribe, send mail to the associated administrative address for the mailing list.

`<zsh-announce-subscribe@sunsite.dk>`

`<zsh-users-subscribe@sunsite.dk>`

`<zsh-workers-subscribe@sunsite.dk>`

`<zsh-announce-unsubscribe@sunsite.dk>`

`<zsh-users-unsubscribe@sunsite.dk>`

`<zsh-workers-unsubscribe@sunsite.dk>`

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*.

If you have problems subscribing/unsubscribing to any of the mailing lists, send mail to `<listmaster@zsh.org>`. The mailing lists are maintained by Karsten Thygesen `<karthy@kom.auc.dk>`.

The mailing lists are archived; the archives can be accessed via the administrative addresses listed above. There is also a hypertext archive, maintained by Geoff Wing `<gcw@zsh.org>`, available at `http://www.zsh.org/mla/`.

2.4 The Zsh FAQ

Zsh has a list of Frequently Asked Questions (FAQ), maintained by Peter Stephenson `<pws@zsh.org>`. 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.zsh.org/FAQ/`. The contact address for FAQ-related matters is `<faqmaster@zsh.org>`.

2.5 The Zsh Web Page

Zsh has a web page which is located at `http://www.zsh.org/`. This is maintained by Karsten Thygesen `<karthy@zsh.org>`, of SunSITE Denmark. The contact address for web-related matters is `<webmaster@zsh.org>`.

2.6 The Zsh Userguide

A userguide is currently in preparation. It is intended to complement the manual, with explanations and hints on issues where the manual can be cabbalistic, hierographic, or downright mystifying (for example, the word 'hierographic' does not exist). It can be viewed in its current state at `http://zsh.sunsite.dk/Guide/`. At the time of writing, chapters dealing with startup files and their contents and the new completion system were essentially complete.

2.7 See Also

man page sh(1), man page csh(1), man page tcsh(1), man page rc(1), man page bash(1), man page ksh(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

3.1 Invocation Options

The following flags are interpreted by the shell when invoked to determine where the shell will read commands from:

`-c` Take the first argument as a command to execute, rather than reading commands from a script or standard input. If any further arguments are given, the first one is assigned to `$0`, rather than being used as a positional parameter.

`-i` Force shell to be interactive.

`-s` Force shell to read commands from the standard input. 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.

After the first one or two arguments have been appropriated as described above, the remaining arguments are assigned to the positional parameters.

For further options, which are common to invocation and the `set` builtin, see Chapter 15 [Options], page 59.

Options may be specified by name using the `-o` option. `-o` acts like a single-letter option, but takes a following string as the option name. For example,

```
zsh -x -o shwordsplit scr
```

runs the script `scr`, setting the `XTRACE` option by the corresponding letter '`-x`' and the `SH_WORD_SPLIT` option by name. Options may be turned *off* by name by using `+o` instead of `-o`. `-o` can be stacked up with preceding single-letter options, so for example '`-xo shwordsplit`' or '`-xoshwordsplit`' is equivalent to '`-x -o shwordsplit`'.

Options may also be specified by name in GNU long option style, '`--`*option-name*'. When this is done, '`-`' characters in the option name are permitted: they are translated into '`_`', and thus ignored. So, for example, '`zsh --sh-word-split`' invokes zsh with the `SH_WORD_SPLIT` option turned on. Like other option syntaxes, options can be turned off by replacing the initial '`-`' with a '`+`'; thus '`+-sh-word-split`' is equivalent to '`--no-sh-word-split`'. Unlike other option syntaxes, GNU-style long options cannot be stacked with any other options, so for example '`-x-shwordsplit`' is an error, rather than being treated like '`-x --shwordsplit`'.

The special GNU-style option '`--version`' is handled; it sends to standard output the shell's version information, then exits successfully. '`--help`' is also handled; it sends to standard output a list of options that can be used when invoking the shell, then exits successfully.

Option processing may be finished, allowing following arguments that start with '`-`' or '`+`' to be treated as normal arguments, in two ways. Firstly, a lone '`-`' (or '`+`') as an argument by itself ends option processing. Secondly, a special option '`--`' (or '`+-`'), which may be specified on its own (which is the standard POSIX usage) or may be stacked with preceding options (so '`-x-`' is equivalent to '`-x --`'). Options are not permitted to be stacked after '`--`' (so '`-x-f`' is an error),

but note the GNU-style option form discussed above, where '`--shwordsplit`' is permitted and does not end option processing.

Except when the *sh*/*ksh* emulation single-letter options are in effect, the option '`-b`' (or '`+b`') ends option processing. '`-b`' is like '`--`', except that further single-letter options can be stacked after the '`-b`' and will take effect as normal.

3.2 Compatibility

Zsh tries to emulate *sh* or *ksh* when it is invoked as `sh` or `ksh` respectively; more precisely, it looks at the first letter of the name by which it was invoked, excluding any initial '`r`' (assumed to stand for 'restricted'), and if that is '`s`' or '`k`' it will emulate *sh* or *ksh*. Furthermore, if invoked as `su` (which happens on certain systems when the shell is executed by the `su` command), the shell will try to find an alternative name from the `SHELL` environment variable and perform emulation based on that.

In *sh* and *ksh* compatibility modes the following parameters are not special and not initialized by the shell: `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.

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_GLOBAL_EXPORT`, `NO_HUP`, `INTERACTIVE_COMMENTS`, `KSH_ARRAYS`, `NO_MULTIOS`, `NO_NOMATCH`, `NO_NOTIFY`, `POSIX_BUILTINS`, `NO_PROMPT_PERCENT`, `RM_STAR_SILENT`, `SH_FILE_EXPANSION`, `SH_GLOB`, `SH_OPTION_LETTERS`, `SH_WORD_SPLIT`. Additionally the `BSD_ECHO` and `IGNORE_BRACES` options are set if zsh is invoked as `sh`. Also, the `KSH_OPTION_PRINT`, `LOCAL_OPTIONS`, `PROMPT_BANG`, `PROMPT_SUBST` and `SINGLE_LINE_ZLE` options are set if zsh is invoked as `ksh`.

3.3 Restricted Shell

When the basename of the command used to invoke zsh starts with the letter '`r`' or the '`-r`' command line option is supplied at invocation, the shell becomes restricted. Emulation mode is determined after stripping the letter '`r`' from the invocation name. The following are disabled in restricted mode:

- changing directories with the `cd` builtin
- changing or unsetting the `PATH`, `path`, `MODULE_PATH`, `module_path`, `SHELL`, `HISTFILE`, `HISTSIZE`, `GID`, `EGID`, `UID`, `EUID`, `USERNAME`, `LD_LIBRARY_PATH`, `LD_AOUT_LIBRARY_PATH`, `LD_PRELOAD` and `LD_AOUT_PRELOAD` parameters
- specifying command names containing `/`
- specifying command pathnames using `hash`
- redirecting output to files
- using the `exec` builtin command to replace the shell with another command
- using `jobs -Z` to overwrite the shell process' argument and environment space
- using the `ARGV0` parameter to override `argv[0]` for external commands
- turning off restricted mode with `set +r` or `unsetopt RESTRICTED`

These restrictions are enforced after processing the startup files. The startup files should set up PATH to point to a directory of commands which can be safely invoked in the restricted environment. They may also add further restrictions by disabling selected builtins.

Restricted mode can also be activated any time by setting the RESTRICTED option. This immediately enables all the restrictions described above even if the shell still has not processed all startup files.

4 Files

4.1 Startup/Shutdown Files

Commands are first read from /etc/zshenv; this cannot be overridden. Subsequent behaviour is modified by the RCS and GLOBAL_RCS options; the former affects all startup files, while the second only affects those in the /etc directory. If one of the options is unset at any point, any subsequent startup file(s) of the corresponding type will not be read. It is also possible for a file in $ZDOTDIR to re-enable GLOBAL_RCS. Both RCS and GLOBAL_RCS are set by default.

Commands are then read from $ZDOTDIR/.zshenv. If the shell is a login shell, commands are read from /etc/zprofile and then $ZDOTDIR/.zprofile. Then, if the shell is interactive, 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.

When a login shell exits, the files $ZDOTDIR/.zlogout and then /etc/zlogout are read. This happens with either an explicit exit via the exit or logout commands, or an implicit exit by reading end-of-file from the terminal. However, if the shell terminates due to exec'ing another process, the logout files are not read. These are also affected by the RCS and GLOBAL_RCS options. Note also that the RCS option affects the saving of history files, i.e. if RCS is unset when the shell exits, no history file will be saved.

If ZDOTDIR is unset, HOME is used instead. Those files listed above as being in /etc may be in another directory, depending on the installation.

As /etc/zshenv is run for all instances of zsh, it is important that it be kept as small as possible. In particular, it is a good idea to put code that does not need to be run for every single shell behind a test of the form '`if [[ -o rcs ]]; then ...`' so that it will not be executed when zsh is invoked with the '`-f`' option.

4.2 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 (installation-specific - /etc is the default)

Any of these files may be pre-compiled with the `zcompile` builtin command (Chapter 16 [Shell Builtin Commands], page 74). If a compiled file exists (named for the original file plus the `.zwc` extension) and it is newer than the original file, the compiled file will be used instead.

5 Shell Grammar

5.1 Simple Commands & Pipelines

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. For example,

```
echo foo
```

is a simple command with arguments.

A *pipeline* is either a simple command, or a sequence of two or more simple commands where each command is separated from the next by '`|`' or '`|&`'. Where commands are separated by '`|`', the standard output of the first command is connected to the standard input of the next. '`|&`' is shorthand for '`2>&1 |`', which connects both the standard output and the standard error of the command to the standard input of the next. The value of a pipeline is the value of the last command, unless the pipeline is preceded by '`!`' in which case the value is the logical inverse of the value of the last command. For example,

```
echo foo | sed 's/foo/bar/'
```

is a pipeline, where the output ('`foo`' plus a newline) of the first command will be passed to the input of the second.

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 or with '`print -p`' and '`read -p`'. A pipeline cannot be preceded by both '`coproc`' and '`!`'. If job control is active, the coprocess can be treated in other than input and output as an ordinary background job.

A *sublist* is either a single pipeline, or a sequence of two or more pipelines separated by '`&&`' or '`||`'. If two pipelines are separated by '`&&`', the second pipeline is executed only if the first succeeds (returns a zero value). If two pipelines are separated by '`||`', the second is executed only if the first fails (returns a nonzero value). Both operators have equal precedence and are left associative. The value of the sublist is the value of the last pipeline executed. For example,

```
dmesg | grep panic && print yes
```

is a sublist consisting of two pipelines, the second just a simple command which will be executed if and only if the `grep` command returns a zero value. If it does not, the value of the sublist is that return value, else it is the value returned by the `print` (almost certainly zero).

A *list* is a sequence of zero or more sublists, in which each sublist is terminated by '`;`', '`&`', '`&|`', '`&!`', or a newline. This terminator may optionally be omitted from the last sublist in the list when the list appears as a complex command inside '`(...)`' or '`{...}`'. When a sublist is terminated by '`;`' or newline, the shell waits for it to finish before executing the next sublist. If a sublist is terminated by a '`&`', '`&|`', or '`&!`', the shell executes the last pipeline in it in the background, and does not wait for it to finish (note the difference from other shells which execute the whole sublist in the background). A backgrounded pipeline returns a status of zero.

More generally, a list can be seen as a set of any shell commands whatsoever, including the complex commands below; this is implied wherever the word 'list' appears in later descriptions. For example, the commands in a shell function form a special sort of list.

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. This must appear before any other precommand modifier, as it is interpreted immediately, before any parsing is done. It has no effect in non-interactive shells.

`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.

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

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 at least one newline or `;`. Expand the list of *word*s, 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 *word*s.

`for ((` [*expr1*] `;` [*expr2*] `;` [*expr3*] `)) do` *list* `done`
The arithmetic expression *expr1* is evaluated first (see Chapter 10 [Arithmetic Evaluation], page 18). The arithmetic expression *expr2* is repeatedly evaluated until it evaluates to zero and when non-zero, *list* is executed and the arithmetic expression *expr3* evaluated. If any expression is omitted, then it behaves as if it evaluated to 1.

`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 13.8 [Filename Generation], page 38. If the *list* that is executed is terminated with `;&` rather than `;;`, the following list is also executed. This continues until either a list is terminated with `;;` or the `esac` is reached.

`select` *name* [`in` *word* ... *term*] `do` *list* `done`
: where *term* is one or more newline or `;` to terminate the *word*s. Print the set of *word*s, 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 the line editor if the shell is interactive and that is active, or else standard input. If this line consists of the number of one of the listed *word*s, 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 or more newline or `;`. Define a function which is referenced by any one of *word*. Normally, only one *word* is provided; multiple *word*s are usually only useful for setting traps. The body of the function is the *list* between the `{` and `}`. See Chapter 8 [Functions], page 14.

  If the option `SH_GLOB` is set for compatibility with other shells, then whitespace may appear between between the left and right parentheses when there is a single *word*; otherwise, the parentheses will be treated as forming a globbing pattern in that case.

`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 11 [Conditional Expressions], page 20 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 `SHORT_LOOPS` option is set. For the `if`, `while` and `until` commands, in both these cases the test part of the loop must also be suitably delimited, such as by '`[[ ... ]]`' or '`(( ... ))`', else the end of the test will not be recognized. For the `for`, `repeat`, `case` and `select` commands no such special form for the arguments is necessary, but the other condition (the special form of *sublist* or use of the `SHORT_LOOPS` option) still applies.

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

```
if [[ -o ignorebraces ]] {
  print yes
}
```

works, but

```
if true {  # Does not work!
  print yes
}
```

does *not*, since the test is not suitably delimited.

`if` *list sublist*
: A short form of the alternate 'if'. The same limitations on the form of *list* apply as for the previous form.

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

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

`for ((` [*expr1*] `;` [*expr2*] `;` [*expr3*] `))` *sublist*
: A short form of the arithmetic `for` command.

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

`while` *list* `{` *list* `}`
: An alternative form of `while`. Note the limitations on the form of *list* mentioned above.

`until` *list* `{` *list* `}`
: An alternative form of `until`. Note the limitations on the form of *list* mentioned above.

`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 at least one 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 noninteractive 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 expansion is done on the shell input before any other expansion except history expansion. Therefore, if an alias is defined for the word `foo`, alias expansion 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.

A string enclosed between '`$'` and '`'` is processed the same way as the string arguments of the `print` builtin, and the resulting string is considered to be entirely quoted. A literal '`'` character can be included in the string by using the '`\'`' escape.

All characters enclosed between a pair of single quotes (`''`) that is not preceded by a '`$`' are quoted. A single quote cannot appear within single quotes unless the option `RC_QUOTES` is set, in which case a pair of single quotes are turned into a single quote. For example,

```
print ''''
```

outputs nothing apart from a newline if `RC_QUOTES` is not set, but one single quote if it is set.

Inside double quotes (`""`), parameter and command substitution occur, and '\' quotes the characters '\', '`', '"', and '`$`'.

6 Redirection

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.

The following may appear anywhere in a simple command or may precede or follow a complex command. Expansion 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* for reading 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* for writing 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* for writing in append mode as standard output. 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 expansion, 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. This is known as a *here-string*.

`<&` *number*
`>&` *number*
The standard input/output is duplicated from file descriptor *number* (see man page dup2(2)).

`<& -`
`>& -` Close the standard input/output.

`<& p`
`>& p` The input/output from/to the coprocess is moved to the standard input/output.

`>&` *word*
`&>` *word* (Except where '`>&` *word*' matches one of the above syntaxes; '`&>`' can always be used to avoid this ambiguity.) Redirects both standard output and standard error (file descriptor 2) in the manner of '`>` *word*'. Note that this does *not* have the same effect as '`>` *word* `2>&1`' in the presence of multios (see the section below).

`>&|` *word*
`>&!` *word*
`&>|` *word*
`&>!` *word* Redirects both standard output and standard error (file descriptor 2) in the manner of '`>|` *word*'.

`>>&` *word*
`&>>` *word* Redirects both standard output and standard error (file descriptor 2) in the manner of '`>>` *word*'.

`>>&|` *word*
`>>&!` *word*
`&>>|` *word*
`&>>!` *word* Redirects both standard output and standard error (file descriptor 2) in the manner of '`>>|` *word*'.

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*.

6.1 Multios

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*, provided the `MULTIOS` option is set, as it is by default. Thus:

```
date >foo >bar
```

writes the date to two files, named '`foo`' and '`bar`'. Note that a pipe is an implicit redirection; 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 '`*`'.) Similarly, you can do

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

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*, provided the `MULTIOS` option is set. Thus

```
sort <foo <fubar
```

or even

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

is equivalent to '`cat foo fubar | sort`'.

Note that a pipe is an implicit redirection; thus

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

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

If the `MULTIOS` option is *un*set, 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.

6.2 Redirections with no command

When a simple command consists of one or more redirection operators and zero or more parameter assignments, but no command name, zsh can behave in several ways.

If the parameter `NULLCMD` is not set or the option `CSH_NULLCMD` is set, an error is caused. This is the *csh* behavior and `CSH_NULLCMD` is set by default when emulating *csh*.

If the option `SH_NULLCMD` is set, the builtin '`:`' is inserted as a command with the given redirections. This is the default when emulating *sh* or *ksh*.

Otherwise, if the parameter `NULLCMD` is set, its value will be used as a command with the given redirections. If both `NULLCMD` and `READNULLCMD` are set, then the value of the latter will be used

instead of that of the former when the redirection is an input. The default for `NULLCMD` is '`cat`' and for `READNULLCMD` is '`more`'. Thus

```
< file
```

shows the contents of `file` on standard output, with paging if that is a terminal. `NULLCMD` and `READNULLCMD` may refer to shell functions.

7 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 in Chapter 8 [Functions], page 14. 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.

8 Functions

Shell functions are defined with the `function` reserved word or the special syntax '*funcname* `()`'. 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. (See Chapter 7 [Command Execution], page 14.)

Functions execute in the same process as the caller and share all files and 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.

8.1 Autoloading Functions

A function can be marked as *undefined* using the `autoload` builtin (or '`functions -u`' or '`typeset -fu`'). Such a function has no body. When the function is first executed, the shell searches for its definition using the elements of the `fpath` variable. Thus to define functions for autoloading, a typical sequence is:

```
fpath=(~/myfuncs $fpath)
autoload myfunc1 myfunc2 ...
```

The usual alias expansion during reading will be suppressed if the `autoload` builtin or its equivalent is given the option `-U`. This is recommended for the use of functions supplied with the zsh distribution. Note that for functions precompiled with the `zcompile` builtin command the flag `-U` must be provided when the `.zwc` file is created, as the corresponding information is compiled into the latter.

For each *element* in `fpath`, the shell looks for three possible files, the newest of which is used to load the definition for the function:

element`.zwc`
: A file created with the `zcompile` builtin command, which is expected to contain the definitions for all functions in the directory named *element*. The file is treated in the same manner as a directory containing files for functions and is searched for the definition of the function. If the definition is not found, the search for a definition proceeds with the other two possibilities described below.

  If *element* already includes a `.zwc` extension (i.e. the extension was explicitly given by the user), *element* is searched for the definition of the function without comparing its age to that of other files; in fact, there does not need to be any directory named *element* without the suffix. Thus including an element such as '`/usr/local/funcs.zwc`' in `fpath` will speed up the search for functions, with the disadvantage that functions included must be explicitly recompiled by hand before the shell notices any changes.

element`/`*function*`.zwc`
: A file created with `zcompile`, which is expected to contain the definition for *function*. It may include other function definitions as well, but those are neither loaded nor executed; a file found in this way is searched *only* for the definition of *function*.

element`/`*function*
: A file of zsh command text, taken to be the definition for *function*.

In summary, the order of searching is, first, in the *parents of* directories in `fpath` for the newer of either a compiled directory or a directory in `fpath`; second, if more than one of these contains a definition for the function that is sought, the leftmost in the `fpath` is chosen; and third, within a directory, the newer of either a compiled function or an ordinary function definition is used.

If the `KSH_AUTOLOAD` option is set, or the file contains only a simple definition of the function, the file's contents will be executed. This will normally define the function in question, but may also perform initialization, which is executed in the context of the function execution, and may therefore define local parameters. It is an error if the function is not defined by loading the file.

Otherwise, the function body (with no surrounding '*funcname*`() {...}`') is taken to be the complete contents of the file. This form allows the file to be used directly as an executable shell script. If processing of the file results in the function being re-defined, the function itself is not re-executed. To force the shell to perform initialization and then call the function defined, the file should contain initialization code (which will be executed then discarded) in addition to a complete function definition (which will be retained for subsequent calls to the function), and a call to the shell function, including any arguments, at the end.

For example, suppose the autoload file `func` contains

```
func() { print This is func; }
print func is initialized
```

then '`func; func`' with `KSH_AUTOLOAD` set will produce both messages on the first call, but only the message '`This is func`' on the second and subsequent calls. Without `KSH_AUTOLOAD` set, it will produce the initialization message on the first call, and the other message on the second and subsequent calls.

It is also possible to create a function that is not marked as autoloaded, but which loads its own definition by searching `fpath`, by using '`autoload -X`' within a shell function. For example, the following are equivalent:

```
myfunc() {
  autoload -X
}
myfunc args...
```

and

```
unfunction myfunc   # if myfunc was defined
autoload myfunc
myfunc args...
```

In fact, the `functions` command outputs '`builtin autoload -X`' as the body of an autoloaded function. A true autoloaded function can be identified by the presence of the comment '`# undefined`' in the body, because all comments are discarded from defined functions. This is done so that

```
eval "$(functions)"
```

produces a reasonable result.

To load the definition of an autoloaded function `myfunc` without executing `myfunc`, use:

```
autoload +X myfunc
```

8.2 Special Functions

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

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

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

`precmd` Executed before each prompt.

`preexec` Executed just after a command has been read and is about to be executed. If the history mechanism is active (and the line was not discarded from the history buffer), the string that the user typed is passed as the first argument, otherwise it is an empty string. The actual command that will be executed (including expanded aliases) is passed in two different forms: the second argument is a single-line, size-limited version of the command (with things like function bodies elided); the third argument contains the full text what what is being executed.

`TRAP`*NAL* If defined and non-null, this function will be executed whenever the shell catches a signal `SIG`*NAL*, where *NAL* is a signal name as specified for the `kill` builtin. The signal number will be passed as the first parameter to the function.

If a function of this form is defined and null, the shell and processes spawned by it will ignore `SIG`*NAL*.

`TRAPDEBUG`
Executed after each command.

`TRAPEXIT` Executed when the shell exits, or when the current function exits if defined inside a function.

`TRAPZERR` Executed whenever a command has a non-zero exit status. However, the function is not executed if the command occurred in a sublist followed by '`&&`' or '`||`'; only the final command in a sublist of this type causes the trap to be executed.

The functions beginning '`TRAP`' may alternatively be defined with the `trap` builtin: this may be preferable for some uses, as they are then run in the environment of the calling process, rather than in their own function environment. Apart from the difference in calling procedure and the fact that the function form appears in lists of functions, the forms

```
TRAPNAL() {
 # code
}
```

and

```
trap '
 # code
' NAL
```

are equivalent.

9 Jobs & Signals

9.1 Jobs

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: this key may be redefined by the `susp` option of the external `stty` command. 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 in 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.

When a command is suspended and continued later with the `fg` or `wait` builtins, zsh restores tty modes that were in effect when it was suspended. This (intentionally) does not apply if the command is continued via '`kill -CONT`', nor when it is continued with `bg`.

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 the `NOTIFY` option 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, if the `HUP` option is set.

To avoid having the shell terminate the running jobs, either use the *nohup* command (see man page nohup(1)) or the `disown` builtin.

9.2 Signals

The `INT` and `QUIT` signals for an invoked command are ignored if the command is followed by '`&`' and the `MONITOR` option is not active. Otherwise, signals have the values inherited by the shell from its parent (but see the `TRAP`*NAL* special functions in Chapter 8 [Functions], page 14).

10 Arithmetic Evaluation

The shell can perform integer and floating point arithmetic, either using the builtin `let`, or via a substitution of the form `$((...))`. For integers, the shell is usually compiled to use 8-byte precision where this is available, otherwise precision is 4 bytes. This can be tested, for example, by giving the command '`print - $(( 12345678901 ))`'; if the number appears unchanged, the precision is at least 8 bytes. Floating point arithmetic is always double precision.

The `let` builtin command takes arithmetic expressions as arguments; each is evaluated separately. Since many of the arithmetic operators, as well as spaces, require quoting, an alternative form is provided: for any command which begins with a '`((`', all the characters until a matching '`))`' are treated as a quoted expression and arithmetic expansion performed as for an argument of `let`. More precisely, '`((...))`' is equivalent to '`let "..."`'. For example, the following statement

```
(( val = 2 + 1 ))
```

is equivalent to

```
let "val = 2 + 1"
```

both assigning the value 3 to the shell variable `var` and returning a zero status.

Integers can be in bases other than 10. A leading '`0x`' or '`0X`' denotes hexadecimal. Integers may also be 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). The *base*`#` may also be omitted, in which case base 10 is used. For backwards compatibility the form '`[`*base*`]`*n*' is also accepted.

It is also possible to specify a base to be used for output in the form '`[#`*base*`]`', for example '`[#16]`'. This is used when outputting arithmetical substitutions or when assigning to scalar parameters, but an explicitly defined integer or floating point parameter will not be affected. If an integer variable is implicitly defined by an arithmetic expression, any base specified in this way will be set as the variable's output arithmetic base as if the option '`-i` *base*' to the `typeset` builtin had been used. The expression has no precedence and if it occurs more than once in a mathematical expression, the last encountered is used. For clarity it is recommended that it appear at the beginning of an expression. As an example:

```
typeset -i 16 y
print $(( [#8] x = 32, y = 32 ))
print $x $y
```

outputs first '`8#40`', the rightmost value in the given output base, and then '`8#40 16#20`', because `y` has been explicitly declared to have output base 16, while `x` (assuming it does not already exist) is implicitly typed by the arithmetic evaluation, where it acquires the output base 8.

When an output base is specified using the '`[#`*base*`]`' syntax, an appropriate base prefix will be output if necessary, so that the value output is valid syntax for input. If the `#` is doubled, for example '`[##16]`', then no base prefix is output.

Floating point constants are recognized by the presence of a decimal point or an exponent. The decimal point may be the first character of the constant, but the exponent character `e` or `E` may not, as it will be taken for a parameter name.

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.

Mathematical functions can be called with the syntax '*func*`(`*args*`)`', where the function decides if the *args* is used as a string or a comma-separated list of arithmetic expressions. The shell currently defines no mathematical functions by default, but the module `zsh/mathfunc` may be loaded with the `zmodload` builtin to provide standard floating point mathematical functions.

An expression of the form '`##`*x*' where *x* is any character sequence such as '`a`', '`^A`', or '`\M-\C-x`' gives the ASCII value of this character and an expression of the form '`#`*foo*' gives the ASCII value of the first character of the value of the parameter *foo*. Note that this is different from the expression '`$#`*foo*', a standard parameter substitution which gives the length of the parameter *foo*. '`#\`' is accepted instead of '`##`', but its use is deprecated.

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

```
((val2 = val1 * 2))
```

assigns twice the value of `$val1` to the parameter named `val2`.

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. Assigning a floating point number to an integer results in rounding down to the next integer.

Likewise, floating point numbers can be declared with the `float` builtin; there are two types, differing only in their output format, as described for the `typeset` builtin. The output format can be bypassed by using arithmetic substitution instead of the parameter substitution, i.e. '`${`*float*`}`' uses the defined format, but '`$((`*float*`))`' uses a generic floating point format.

Promotion of integer to floating point values is performed where necessary. In addition, if any operator which requires an integer ('`~`', '`&`', '`|`', '`^`', '`%`', '`<<`', '`>>`' and their equivalents with assignment) is given a floating point argument, it will be silently rounded down to the next integer.

Scalar variables can hold integer or floating point values at different times; there is no memory of the numeric type in this case.

If a variable is first assigned in a numeric context without previously being declared, it will be implicitly typed as `integer` or `float` and retain that type either until the type is explicitly changed or until the end of the scope. This can have unforeseen consequences. For example, in the loop

```
for (( f = 0; f < 1; f += 0.1 )); do
# use $f
done
```

if `f` has not already been declared, the first assignment will cause it to be created as an integer, and consequently the operation '`f += 0.1`' will always cause the result to be truncated to zero, so that the loop will fail. A simple fix would be to turn the initialization into '`f = 0.0`'. It is therefore best to declare numeric variables with explicit types.

11 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 a regular 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 15.1 [Specifying Options], page 59.)

`-p` *file* true if *file* exists and is a FIFO special file (named pipe).

`-r` *file* true if *file* exists and is readable by 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 '`==`' form is the preferred one. The '`=`' 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 numerically equal to *exp2*.

exp1 `-ne` *exp2*
true if *exp1* is numerically not equal to *exp2*.

exp1 `-lt` *exp2*
true if *exp1* is numerically less than *exp2*.

exp1 `-gt` *exp2*
true if *exp1* is numerically greater than *exp2*.

exp1 `-le` *exp2*
true if *exp1* is numerically less than or equal to *exp2*.

exp1 `-ge` *exp2*
true if *exp1* is numerically 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.

Normal shell expansion is performed on the *file*, *string* and *pattern* arguments, but the result of each expansion is constrained to be a single word, similar to the effect of double quotes. However, pattern metacharacters are active for the *pattern* arguments; the patterns are the same as those used for filename generation, see Section 13.8 [Filename Generation], page 38, but there is no special behaviour of '`/`' nor initial dots, and no glob qualifiers are allowed.

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

In the forms which do numeric comparison, the expressions *exp* undergo arithmetic expansion as if they were enclosed in `$((...))`.

For example, the following:

```
[[ ( -f foo || -f bar ) && $report = y* ]] && print File exists.
```

tests if either file `foo` or file `bar` exists, and if so, if the value of the parameter `report` begins with '`y`'; if the complete condition is true, the message '`File exists.`' is printed.

12 Prompt Expansion

Prompt sequences undergo a special form of expansion. This type of expansion is also available using the `-P` option to the `print` builtin.

If the `PROMPT_SUBST` option is set, the prompt string is first subjected to *parameter expansion*, *command substitution* and *arithmetic expansion*. See Chapter 13 [Expansion], page 25.

Certain escape sequences may be recognised in the prompt string.

If the `PROMPT_BANG` option is set, a '`!`' in the prompt is replaced by the current history event number. A literal '`!`' may then be represented as '`!!`'.

If the `PROMPT_PERCENT` option is set, certain escape sequences that start with '`%`' are expanded. Some escapes take an optional integer argument, which should appear between the '`%`' and the next character of the sequence. The following escape sequences are recognized:

`%%` A '`%`'.

`%)` A '`)`'.

`%d`
`%/` Present working directory (`$PWD`). If an integer follows the '`%`', it specifies a number of trailing components of `$PWD` to show; zero means the whole path. A negative integer specifies leading components, i.e. `%-1d` specifies the first component.

`%~` As `%d` and `%/`, but if `$PWD` has a named directory as its prefix, that part is replaced by a '`~`' followed by the name of the directory. If it starts with `$HOME`, that part is replaced by a '`~`'.

`%h`
`%!` Current history event number.

`%L` The current value of `$SHLVL`.

`%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. With a negative integer, trailing components of the hostname are shown.

`%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`.

`%N` The name of the script, sourced file, or shell function that zsh is currently executing, whichever was started most recently. If there is none, this is equivalent to the parameter `$0`. An integer may follow the '`%`' to specify a number of trailing path components to show; zero means the full path. A negative integer specifies leading components.

`%i` The line number currently being executed in the script, sourced file, or shell function given by `%N`. This is most useful for debugging as part of `$PS4`.

`%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 man page strftime(3) for more details. Three additional codes are available: `%f` prints the day of the month, like `%e` but without any preceding space if the day is a single digit, and `%K`/`%L` correspond to `%k`/`%l` for the hour of the day (24/12 hour clock) in the same way.

`%l` The line (tty) the user is logged in on without `/dev/` prefix. If name starts with `/dev/tty` this is stripped.

`%y` The line (tty) the user is logged in on without `/dev/` prefix. It does not treat `/dev/tty*` specially.

`%?` 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 negative or no integer means print as many as there are. This is most useful in prompts `PS2` for continuation lines and `PS4` for debugging with the `XTRACE` option; in the latter case it will also work non-interactively.

`%E` Clears to end of line.

`%#` A '`#`' if the shell is running with privileges, a '`%`' if not. Equivalent to '`%(!.#.%%)`'. The definition of 'privileged', for these purposes, is that either the effective user ID is zero, or, if POSIX.1e capabilities are supported, that at least one capability is raised in either the Effective or Inheritable capability vectors.

`%v` The value of the first element of the `psvar` array parameter. Following the '`%`' with an integer gives that element of the array. Negative integers count from the end of the array.

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

`%(`*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. This separator may not appear in the *true-text*, except as part of a %-escape sequence. A '`)`' may appear in the *false-text* as '`%)`'. *true-text* and *false-text* may both 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. A negative integer will be multiplied by -1. 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 at least *n* characters have already been printed on the current line.

`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.

`!` True if the shell is running with privileges.

`%<`*string*`<`
`%>`*string*`>`
`%[`*xstring*`]`
Specifies truncation behaviour for the remainder of the prompt string. The third, deprecated, form is equivalent to '`%`*xstring*x', 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. The *string* will be displayed in place of the truncated portion of any string; note this does not undergo prompt expansion.

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`'. In this string, the terminating

character ('<', '>' or ']'), or in fact any character, may be quoted by a preceding '\'; note when using `print -P`, however, that this must be doubled as the string is also subject to standard `print` processing, in addition to any backslashes removed by a double quoted string: the worst case is therefore '`print -P "%<\\\\<<..."`'.

If the *string* is longer than the specified truncation length, it will appear in full, completely replacing the truncated string.

The part of the prompt string to be truncated runs to the end of the string, or to the end of the next enclosing group of the '`%(`' construct, or to the next truncation encountered at the same grouping level (i.e. truncations inside a '`%(`' are separate), which ever comes first. In particular, a truncation with argument zero (e.g. '`%<<`') marks the end of the range of the string to be truncated while turning off truncation from there on. For example, the prompt '%10<...<%~%<<%# ' will print a truncated representation of the current directory, followed by a '`%`' or '`#`', followed by a space. Without the '`%<<`', those two characters would be included in the string to be truncated.

`%c`
`%.`
`%C` Trailing component of `$PWD`. An integer may follow the '`%`' to get more than one component. Unless '`%C`' is used, tilde contraction is performed first. These are deprecated as `%c` and `%C` are equivalent to `%1~` and `%1/`, respectively, while explicit positive integers have the same effect as for the latter two sequences.

13 Expansion

The following types of expansions are performed in the indicated order in five steps:

History Expansion
: This is performed only in interactive shells.

Alias Expansion
: Aliases are expanded immediately before the command line is parsed as explained in Section 5.7 [Aliasing], page 11.

Process Substitution
Parameter Expansion
Command Substitution
Arithmetic Expansion
Brace Expansion
: These five are performed in one step in left-to-right fashion. After these expansions, all unquoted occurrences of the characters '\', "'" and '"' are removed.

Filename Expansion
: If the `SH_FILE_EXPANSION` option is set, the order of expansion is modified for compatibility with *sh* and *ksh*. In that case *filename expansion* is performed immediately after *alias expansion*, preceding the set of five expansions mentioned above.

Filename Generation
: This expansion, commonly referred to as *globbing*, is always done last.

The following sections explain the types of expansion in detail.

13.1 History Expansion

History expansion 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. Immediately before execution, each command is saved in the history list, the size of which is controlled by the `HISTSIZE` parameter. The one most recent command is always retained in any case. Each saved command in the history list is called a history *event* and is assigned a number, beginning with 1 (one) when the shell starts up. The history number that you may see in your prompt (see Chapter 12 [Prompt Expansion], page 22) is the number that is to be assigned to the *next* command.

13.1.1 Overview

A history expansion begins with the first character of the `histchars` parameter, which is '`!`' by default, and may occur anywhere on the command line; history expansions 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. Following this history character is an optional event designator (Section 13.1.2 [Event Designators], page 26) and then an optional word designator (Section 13.1.3 [Word Designators], page 27); if neither of these designators is present, no history expansion occurs.

Input lines containing history expansions are echoed after being expanded, but before any other expansions take place and before the command is executed. It is this expanded form that is recorded as the history event for later references.

By default, a history reference with no event designator refers to the same event as any preceding history reference on that command line; if it is the only history reference in a command, it refers to the previous command. However, if the option `CSH_JUNKIE_HISTORY` is set, then every history reference with no event specification *always* refers to the previous command.

For example, '`!`' is the event designator for the previous command, so '`!!:1`' always refers to the first word of the previous command, and '`!!$`' always refers to the last word of the previous command. With `CSH_JUNKIE_HISTORY` set, then '`!:1`' and '`!$`' function in the same manner as '`!!:1`' and '`!!$`', respectively. Conversely, if `CSH_JUNKIE_HISTORY` is unset, then '`!:1`' and '`!$`' refer to the first and last words, respectively, of the same event referenced by the nearest other history reference preceding them on the current command line, or to the previous command if there is no preceding reference.

The character sequence '*^foo^bar*' (where '`^`' is actually the second character of the `histchars` parameter) repeats the last command, replacing the string *foo* with *bar*. More precisely, the sequence '*^foo^bar^*' is synonymous with '`!!:s^`*foo*`^`*bar*`^`', hence other modifiers (see Section 13.1.4 [Modifiers], page 27) may follow the final '`^`'.

If the shell encounters the character sequence '`!"`' in the input, the history mechanism is temporarily disabled until the current list (see Chapter 5 [Shell Grammar], page 7) 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.

13.1.2 Event Designators

An event designator is a reference to a command-line entry in the history list. In the list below, remember that the initial '`!`' in each item may be changed to another character by setting the `histchars` parameter.

`!` Start a history expansion, except when followed by a blank, newline, '`=`' or '`(`'. If followed immediately by a word designator (Section 13.1.3 [Word Designators],

page 27), this forms a history reference with no event designator (Section 13.1.1 [Overview], page 26).

`!!` Refer to the previous command. By itself, this expansion 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*. The trailing '`?`' is necessary if this reference is to be followed by a modifier or followed by any text that is not to be considered part of *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).

13.1.3 Word Designators

A word designator indicates which word or words of a given command line are to be included in a history reference. A '`:`' usually separates the event specification from the word designator. It may be omitted only 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; *x* defaults to `0`.

`*` All the arguments, or a null value if there are none.

x`*` Abbreviates '*x*`-$`'.

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

Note that a '`%`' word designator works only when used in one of '`!%`', '`!:%`' or '`!?`*str*`?:%`', and only when used after a `!?` expansion (possibly in an earlier command). Anything else results in an error, although the error may not be the most obvious one.

13.1.4 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 generation* and *parameter expansion*, except where noted.

`h` Remove a trailing pathname component, leaving the head. This works like '`dirname`'.

`r` Remove a filename extension of the form '`.`*xxx*', leaving the root name.

`e` Remove all but the extension.

`t` Remove all leading pathname components, leaving the tail. This works like '`basename`'.

`p` Print the new command but do not execute it. Only works with history expansion.

`q` Quote the substituted words, escaping further substitutions. Works with history expansion and parameter expansion, though for parameters it is only useful if the resulting text is to be re-evaluated such as by `eval`.

`Q` Remove one level of quotes from the substituted words.

`x` Like `q`, but break into words at whitespace. Does not work with parameter expansion.

`l` Convert the words to all lowercase.

`u` Convert the words to all uppercase.

`s/`*l*`/`*r*`[/]` Substitute *r* for *l* as described below. Unless preceded immediately by a `g`, with no colon between, the substitution is done only for the first string that matches *l*. For arrays and for filename generation, this applies to each word of the expanded text.

`&` Repeat the previous `s` substitution. Like `s`, may be preceded immediately by a `g`. In parameter expansion the `&` must appear inside braces, and in filename generation it must be quoted with a backslash.

The `s/l/r/` substitution works as follows. 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 *r*, is replaced by the text from the left-hand-side *l*. The '`&`' can be quoted with a backslash. A null *l* uses the previous string either from the previous *l* or from the contextual scan string *s* from '`!?`*s*'. You can omit the rightmost delimiter if a newline immediately follows *r*; the rightmost '`?`' in a context scan can similarly be omitted. Note the same record of the last *l* and *r* is maintained across all forms of expansion.

The following `f`, `F`, `w` and `W` modifiers work only with parameter expansion and filename generation. They are listed here to provide a single point of reference for all modifiers.

`f` Repeats the immediately (without a colon) following modifier until the resulting word doesn't change any more.

`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 '`(`', '`[`', or '`{`' is used as the opening delimiter, the closing delimiter should 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.

13.2 Process Substitution

Each command argument of the form '`<(`*list*`)`', '`>(`*list*`)`' or '`=(`*list*`)`' is subject to process substitution. In the case of the `<` or `>` forms, the shell runs process *list* asynchronously. If the system supports the `/dev/fd` mechanism, the command argument is the name of the device file corresponding to a file descriptor; otherwise, if the system supports named pipes (FIFOs), the command argument will be a named pipe. If the form with `>` is selected then writing on this special file will provide input for *list*. If `<` is used, then the file passed as an argument will be connected to the output of the *list* process. For example,

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

cuts fields 1 and 3 from the files *file1* and *file2* respectively, pastes the results together, and sends it to the processes *process1* and *process2*.

Both the `/dev/fd` and the named pipe implementation have drawbacks. In the former case, some programmes may automatically close the file descriptor in question before examining the file on the command line, particularly if this is necessary for security reasons such as when the programme is running setuid. In the second case, if the programme does not actually open the file, the subshell attempting to read from or write to the pipe will (in a typical implementation, different operating systems may have different behaviour) block for ever and have to be killed explicitly. In both cases, the shell actually supplies the information using a pipe, so that programmes that expect to lseek (see man page lseek(2)) on the file will not work.

Also note that the previous example can be more compactly and efficiently written (provided the `MULTIOS` option is set) 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 (see man page lseek(2)) on the input file.

13.3 Parameter Expansion

The character '`$`' is used to introduce parameter expansions. See Chapter 14 [Parameters], page 46 for a description of parameters, including arrays, associative arrays, and subscript notation to access individual array elements.

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 13.8 [Filename Generation], page 38. Note that these patterns, along with the replacement text of any substitutions, are themselves subject to parameter expansion, command substitution, and arithmetic expansion. In addition to the following operations, the colon modifiers described in Section 13.1.4 [Modifiers], page 27 in Section 13.1 [History Expansion], page 26 can be applied: for example, `${i:s/foo/bar/}` performs string substitution on the expansion of parameter `$i`.

`${`*name*`}` The value, if any, of the parameter *name* is substituted. The braces are required if the expansion is to be followed by a letter, digit, or underscore that is not to be interpreted as part of *name*. In addition, more complicated forms of substitution usually require the braces to be present; exceptions, which only apply if the option `KSH_ARRAYS` is not set, are a single subscript or any colon modifiers appearing after the name, or any of the characters '`^`', '`=`', '`~`', '`#`' or '`+`' appearing before the name, all of which work with or without braces.

If *name* is an array parameter, and the `KSH_ARRAYS` option is not set, then the value of each element of *name* is substituted, one element per word. Otherwise, the expansion results in one word only; with `KSH_ARRAYS`, this is the first element of an array. No field splitting is done on the result unless the `SH_WORD_SPLIT` option is set.

`${+`*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*.

`${`*name*`:=`*word*`}`
`${`*name*`::=`*word*`}`
: In the first form, if *name* is unset or is null then set it to *word*; in the second form, unconditionally set *name* to *word*. In both forms, 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. Interactive shells instead return to the prompt. 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.

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

In the following expressions, when *name* is an array and the substitution is not quoted, or if the '`(@)`' flag or the *name*`[@]` syntax is used, matching and replacement is performed on each array element separately.

`${`*name*`#`*pattern*`}`
`${`*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.

`${`*name*`%`*pattern*`}`
`${`*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.

`${`*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 the matching array elements are removed (use the '`(M)`' flag to remove the non-matched elements).

`${`*name*`/`*pattern*`/`*repl*`}`
`${`*name*`//`*pattern*`/`*repl*`}`
: Replace the longest possible match of *pattern* in the expansion of parameter *name* by string *repl*. The first form replaces just the first occurrence, the second form all occurrences. Both *pattern* and *repl* are subject to double-quoted substitution, so that expressions like `${name/$opat/$npat}` will work, but note the usual rule that pattern characters in `$opat` are not treated specially unless either the option `GLOB_SUBST` is set, or `$opat` is instead substituted as `${~opat}`.

  The *pattern* may begin with a '`#`', in which case the *pattern* must match at the start of the string, or '`%`', in which case it must match at the end of the string. The *repl* may be an empty string, in which case the final '`/`' may also be omitted. To quote the final '`/`' in other cases it should be preceded by two backslashes (i.e., a quoted backslash); this is not necessary if the '`/`' occurs inside a substituted parameter. Note also that the '`#`' and '`%`' are not active if they occur inside a substituted parameter, even at the start.

  The first '`/`' may be preceded by a '`:`', in which case the match will only succeed if it matches the entire word. Note also the effect of the `I` and `S` parameter expansion flags below; however, the flags `M`, `R`, `B`, `E` and `N` are not useful.

For example,

```
foo="twinkle twinkle little star" sub="t*e" rep="spy"
print ${foo//${~sub}/$rep}
print ${(S)foo//${~sub}/$rep}
```

Here, the '`~`' ensures that the text of `$sub` is treated as a pattern rather than a plain string. In the first case, the longest match for `t*e` is substituted and the result is '`spy star`', while in the second case, the shortest matches are taken and the result is '`spy spy lispy star`'.

`${#`*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. Note that '`^`', '`=`', and '`~`', below, must appear to the left of '`#`' when these forms are combined.

`${^`*spec*`}` Turn on the `RC_EXPAND_PARAM` option for the evaluation of *spec*; if the '`^`' is doubled, turn it off. When this option is set, array expansions of the form *foo*`${`*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*'.

Internally, each such expansion is converted into the equivalent list for brace expansion. E.g., `${^var}` becomes `{$var[1],$var[2],...}`, and is processed as described in Section 13.6 [Brace Expansion], page 37 below. If word splitting is also in effect the `$var[`*N*`]` may themselves be split into different list elements.

`${=`*spec*`}` Perform word splitting using the rules for `SH_WORD_SPLIT` during the evaluation of *spec*, but regardless of whether the parameter appears in double quotes; if the '`=`' is doubled, turn it off. This forces parameter expansions to be split into separate words before substitution, using `IFS` as a delimiter. This is done by default in most other shells.

Note that splitting is applied to *word* in the assignment forms of *spec before* the assignment to *name* is performed. This affects the result of array assignments with the `A` flag.

`${~`*spec*`}` Turn on the `GLOB_SUBST` option for the evaluation of *spec*; if the '`~`' is doubled, turn it off. When this option is set, the string resulting from the expansion will be interpreted as a pattern anywhere that is possible, such as in filename expansion and filename generation and pattern-matching contexts like the right hand side of the '`=`' and '`!=`' operators in conditions.

If a `${...}` type parameter expression or a `$(...)` type command substitution is used in place of *name* above, it is expanded first and the result is used as if it were the value of *name*. Thus it is possible to perform nested operations: `${${foo#head}%tail}` substitutes the value of `$foo` with both '`head`' and '`tail`' deleted. The form with `$(...)` is often useful in combination with the flags described next; see the examples below. Each *name* or nested `${...}` in a parameter expansion may also be followed by a subscript expression as described in Section 14.2 [Array Parameters], page 46.

Note that double quotes may appear around nested expressions, in which case only the part inside is treated as quoted; for example, `${(f)"$(foo)"}` quotes the result of `$(foo)`, but the flag '`(f)`' (see below) is applied using the rules for unquoted expansions. Note further that quotes are themselves nested in this context; for example, in `"${(@f)"$(foo)"}"`, there are two sets of quotes, one surrounding the whole expression, the other (redundant) surrounding the `$(foo)` as before.

13.3.1 Parameter Expansion Flags

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. In cases where repeating a flag is meaningful,

the repetitions need not be consecutive; for example, '(q%q%q)' means the same thing as the more readable '(%%qqq)'. The following flags are supported:

`%` Expand all `%` escapes in the resulting words in the same way as in in prompts (see Chapter 12 [Prompt Expansion], page 22). If this flag is given twice, full prompt expansion is done on the resulting words, depending on the setting of the `PROMPT_PERCENT`, `PROMPT_SUBST` and `PROMPT_BANG` options.

`@` In double quotes, array elements are put into separate words. E.g., '`"${(@)foo}"`' is equivalent to '`"${foo[@]}"`' and '`"${(@)foo[1,2]}"`' is the same as '`"$foo[1]" "$foo[2]"`'. This is distinct from *field splitting* by the the `f`, `s` or `z` flags, which still applies within each array element.

`A` Create an array parameter with '`${...=...}`', '`${...:=...}`' or '`${...::=...}`'. If this flag is repeated (as in '`AA`'), create an associative array parameter. Assignment is made before sorting or padding. The *name* part may be a subscripted range for ordinary arrays; the *word* part *must* be converted to an array, for example by using '`${(AA)=`*name*`=...}`' to activate field splitting, when creating an associative array.

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

`C` Capitalize the resulting words. 'Words' in this case refers to sequences of alphanumeric characters separated by non-alphanumerics, *not* to words that result from field splitting.

`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.

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

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

`i` With `o` or `O`, sort case-independently.

`k` If *name* refers to an associative array, substitute the *keys* (element names) rather than the values of the elements. Used with subscripts (including ordinary arrays), force indices or keys to be substituted even if the subscript form refers to values. However, this flag may not be combined with subscript ranges.

`L` Convert all letters in the result to lower case.

`o` Sort the resulting words in ascending order.

`O` Sort the resulting words in descending order.

`P` This forces the value of the parameter *name* to be interpreted as a further parameter name, whose value will be used where appropriate. If used with a nested parameter or command substitution, the result of that will be taken as a parameter name in the same way. For example, if you have '`foo=bar`' and '`bar=baz`', the strings `${(P)foo}`, `${(P)${foo}}`, and `${(P)$(echo bar)}` will be expanded to '`baz`'.

`q` Quote the resulting words with backslashes. If this flag is given twice, the resulting words are quoted in single quotes and if it is given three times, the words are quoted in double quotes. If it is given four times, the words are quoted in single quotes preceded by a `$`.

`Q` Remove one level of quotes from the resulting words.

`t` Use a string describing the type of the parameter where the value of the parameter would usually appear. This string consists of keywords separated by hyphens ('`-`'). The first keyword in the string describes the main type, it can be one of '`scalar`', '`array`', '`integer`', '`float`' or '`association`'. The other keywords describe the type in more detail:

- `local` for local parameters
- `left` for left justified parameters
- `right_blanks` for right justified parameters with leading blanks
- `right_zeros` for right justified parameters with leading zeros
- `lower` for parameters whose value is converted to all lower case when it is expanded
- `upper` for parameters whose value is converted to all upper case when it is expanded
- `readonly` for readonly parameters
- `tag` for tagged parameters
- `export` for exported parameters
- `unique` for arrays which keep only the first occurrence of duplicated values
- `hide` for parameters with the 'hide' flag
- `special` for special parameters defined by the shell

`U` Convert all letters in the result to upper case.

`v` Used with `k`, substitute (as two consecutive words) both the key and the value of each associative array element. Used with subscripts, force values to be substituted even if the subscript form refers to indices or keys.

`V` Make any special characters in the resulting words visible.

`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.

`X` With this flag parsing errors occurring with the `Q` and `e` flags or the pattern matching forms such as '`${`*name*`#`*pattern*`}`' are reported. Without the flag they are silently ignored.

`z` Split the result of the expansion into words using shell parsing to find the words, i.e. taking into account any quoting in the value.

Note that this is done very late, as for the '`(s)`' flag. So to access single words in the result, one has to use nested expansions as in '`${${(z)foo}[2]}`'. Likewise, to remove the quotes in the resulting words one would do: '`${(Q)${(z)foo}}`'.

The following flags (except `p`) are followed by one or more arguments as shown. Any character, or the matching pairs '`(...)`', '`{...}`', '`[...]`', or '`<...>`', may be used in place of a colon as delimiters, but note that when a flag takes more than one argument, a matched pair of delimiters must surround each argument.

`p` Recognize the same escape sequences as the `print` builtin in string arguments to any of the flags described below.

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

`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 is inserted once directly to the left of each word, before padding.

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

`s:`*string*`:` Force field splitting (see the option `SH_WORD_SPLIT`) at the separator *string*. Note that a *string* of two or more characters means all must all match in sequence; this differs from the treatment of two or more characters in the `IFS` parameter.

The following flags are meaningful with the `${...#...}` or `${...%...}` forms. The `S` and `I` flags may also be used with the `${.../...}` forms.

`S` Search substrings as well as beginnings or ends; with `#` start from the beginning and with `%` start from the end of the string. With substitution via `${.../...}` or `${...//...}`, specifies non-greedy matching, i.e. that the shortest instead of the longest match should be replaced.

`I:`*expr*`:` Search the *expr*th match (where *expr* evaluates to a number). This only applies when searching for substrings, either with the `S` flag, or with `${.../...}` (only the *expr*th match is substituted) or `${...//...}` (all matches from the *expr*th on are substituted). The default is to take the first match.

The *expr*th match is counted such that there is either one or zero matches from each starting position in the string, although for global substitution matches overlapping previous replacements are ignored. With the `${...%...}` and `${...%%...}` forms, the starting position for the match moves backwards from the end as the index increases, while with the other forms it moves forward from the start.

Hence with the string

```
which switch is the right switch for Ipswich?
```

substitutions of the form `${(SI:`N`:)string#w*ch}` as N increases from 1 will match and remove '`which`', '`witch`', '`witch`' and '`wich`'; the form using '`##`' will match and remove '`which switch is the right switch for Ipswich`', '`witch is the right switch for Ipswich`', '`witch for Ipswich`' and '`wich`'. The form using '`%`' will remove the same matches as for '`#`', but in reverse order, and the form using '`%%`' will remove the same matches as for '`##`' in reverse order.

`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.

`M` Include the matched portion in the result.

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

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

13.3.2 Rules

Here is a summary of the rules for substitution; this assumes that braces are present around the substitution, i.e. `${...}`. Some particular examples are given below. Note that the Zsh

Development Group accepts *no responsibility* for any brain damage which may occur during the reading of the following rules.

1. *Nested Substitution*
   If multiple nested `${...}` forms are present, substitution is performed from the inside outwards. At each level, the substitution takes account of whether the current value is a scalar or an array, whether the whole substitution is in double quotes, and what flags are supplied to the current level of substitution, just as if the nested substitution were the outermost. The flags are not propagated up to enclosing substitutions; the nested substitution will return either a scalar or an array as determined by the flags, possibly adjusted for quoting. All the following steps take place where applicable at all levels of substitution. Note that, unless the '`(P)`' flag is present, the flags and any subscripts apply directly to the value of the nested substitution; for example, the expansion `${${foo}}` behaves exactly the same as `${foo}`.

2. *Parameter Subscripting*
   If the value is a raw parameter reference with a subscript, such as `${`*var*`[3]}`, the effect of subscripting is applied directly to the parameter. Subscripts are evaluated left to right; subsequent subscripts apply to the scalar or array value yielded by the previous subscript. Thus if `var` is an array, `${var[1][2]}` is the second character of the first word, but `${var[2,4][2]}` is the entire third word (the second word of the range of words two through four of the original array). Any number of subscripts may appear.

3. *Parameter Name Replacement*
   The effect of any `(P)` flag, which treats the value so far as a parameter name and replaces it with the corresponding value, is applied.

4. *Double-Quoted Joining*
   If the value after this process is an array, and the substitution appears in double quotes, and no `(@)` flag is present at the current level, the words of the value are joined with the first character of the parameter `$IFS`, by default a space, between each word (single word arrays are not modified). If the `(j)` flag is present, that is used for joining instead of `$IFS`.

5. *Nested Subscripting*
   Any remaining subscripts (i.e. of a nested substitution) are evaluated at this point, based on whether the value is an array or a scalar. As with `2.`, multiple subscripts can appear. Note that `${foo[2,4][2]}` is thus equivalent to `${${foo[2,4]}[2]}` and also to `"${${(@)foo[2,4]}[2]}"` (the nested substitution returns an array in both cases), but not to `"${${foo[2,4]}[2]}"` (the nested substitution returns a scalar because of the quotes).

6. *Modifiers*
   Any modifiers, as specified by a trailing '`#`', '`%`', '`/`' (possibly doubled) or by a set of modifiers of the form `:...` (see Section 13.1.4 [Modifiers], page 27 in Section 13.1 [History Expansion], page 26), are applied to the words of the value at this level.

7. *Forced Joining*
   If the '`(j)`' flag is present, or no '`(j)`' flag is present but the string is to be split as given by rules `8.` or `9.`, and joining did not take place at step `4.`, any words in the value are joined together using the given string or the first character of `$IFS` if none. Note that the '`(F)`' flag implicitly supplies a string for joining in this manner.

8. *Forced Splitting*
   If one of the '`(s)`', '`(f)`' or '`(z)`' flags are present, or the '`=`' specifier was present (e.g. `${=`*var*`}`), the word is split on occurrences of the specified string, or (for `=` with neither of the two flags present) any of the characters in `$IFS`.

9. *Shell Word Splitting*
   If no '`(s)`', '`(f)`' or '`=`' was given, but the word is not quoted and the option `SH_WORD_SPLIT` is set, the word is split on occurrences of any of the characters in `$IFS`. Note this step, too, takes place at all levels of a nested substitution.

10. *Re-Evaluation*
    Any '`(e)`' flag is applied to the value, forcing it to be re-examined for new parameter substitutions, but also for command and arithmetic substitutions.

11. *Padding*
    Any padding of the value by the '`(l.`*fill*`.)`' or '`(r.`*fill*`.)`' flags is applied.

12. *Semantic Joining*
    In contexts where expansion semantics requires a single word to result, all words are rejoined with the first character of `IFS` between. So in '`${(P)${(f)lines}}`' the value of `${lines}` is split at newlines, but then must be joined again before the `P` flag can be applied.

    If a single word is not required, this rule is skipped.

13.3.3 Examples

The flag `f` is useful to split a double-quoted substitution line by line. For example, `${(f)"$(<`*file*`)"}` substitutes the contents of *file* divided so that each line is an element of the resulting array. Compare this with the effect of `$(<`*file*`)` alone, which divides the file up by words, or the same inside double quotes, which makes the entire content of the file a single string.

The following illustrates the rules for nested parameter expansions. Suppose that `$foo` contains the array `(bar baz)`:

`"${(@)${foo}[1]}"`
: This produces the result `b`. First, the inner substitution `"${foo}"`, which has no array (`@`) flag, produces a single word result `"bar baz"`. The outer substitution `"${(@)...[1]}"` detects that this is a scalar, so that (despite the '`(@)`' flag) the subscript picks the first character.

`"${${(@)foo}[1]}"`
: This produces the result '`bar`'. In this case, the inner substitution `"${(@)foo}"` produces the array '`(bar baz)`'. The outer substitution `"${...[1]}"` detects that this is an array and picks the first word. This is similar to the simple case `"${foo[1]}"`.

As an example of the rules for word splitting and joining, suppose `$foo` contains the array '`(ax1 bx1)`'. Then

`${(s/x/)foo}`
: produces the words '`a`', '`1 b`' and '`1`'.

`${(j/x/s/x/)foo}`
: produces '`a`', '`1`', '`b`' and '`1`'.

`${(s/x/)foo%%1*}`
: produces '`a`' and '` b`' (note the extra space). As substitution occurs before either joining or splitting, the operation first generates the modified array `(ax bx)`, which is joined to give `"ax bx"`, and then split to give '`a`', '` b`' and ''. The final empty string will then be elided, as it is not in double quotes.

13.4 Command Substitution

A command enclosed in parentheses preceded by a dollar sign, like '`$(...)`', or quoted with grave accents, like '`...`', is replaced with its standard output, with any trailing newlines deleted. If the substitution is not enclosed in double quotes, the output is broken into words using the `IFS` parameter. The substitution '`$(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.

13.5 Arithmetic Expansion

A string of the form '`$[`*exp*`]`' or '`$((`*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 10 [Arithmetic Evaluation], page 18.

13.6 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.

Note that brace expansion is not part of filename generation (globbing); an expression such as `*/{foo,bar}` is split into two separate words `*/foo` and `*/bar` before filename generation takes place. In particular, note that this is liable to produce a 'no match' error if *either* of the two expressions does not match; this is to be contrasted with `*/(foo|bar)`, which is treated as a single pattern but otherwise has similar effects.

13.7 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 `$HOME`. A '`~`' followed by a '`+`' or a '`-`' is replaced by the value of `$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 / itself. The parameters $PWD and $OLDPWD are never abbreviated in this fashion.

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 the PATH parameter, so that a '~' or an '=' following a ':' is eligible for expansion. All such behaviour 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.

13.8 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 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.

13.8.1 Glob Operators

`*` 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. There are also several named classes of characters, in the form '[:*name*:]' with the following meanings: '[:alnum:]' alphanumeric, '[:alpha:]' alphabetic, '[:blank:]' space or tab, '[:cntrl:]' control character, '[:digit:]' decimal digit, '[:graph:]' printable character except whitespace, '[:lower:]' lowercase letter, '[:print:]' printable character, '[:punct:]' printable character neither alphanumeric nor whitespace, '[:space:]' whitespace character, '[:upper:]' uppercase letter, '[:xdigit:]' hexadecimal digit. These use the macros provided by the operating system to test for the given character combinations, including any modifications due to local language settings: see man page ctype(3). Note that the square brackets are additional to those enclosing the whole set of characters, so to

test for a single alphanumeric character you need '`[[:alnum:]]`'. Named character sets can be used alongside other types, e.g. '`[[:alpha:]0-9]`'.

`[^...]`
`[!...]` 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. Either of the numbers may be omitted to make the range open-ended; hence '`<->`' matches any number. To match individual digits, the `[...]` form is more efficient.

Be careful when using other wildcards adjacent to patterns of this form; for example, `<0-9>*` will actually match any number whatsoever at the start of the string, since the '`<0-9>`' will match the first digit, and the '`*`' will match any others. This is a trap for the unwary, but is in fact an inevitable consequence of the rule that the longest possible match always succeeds. Expressions such as '`<0-9>[^[:digit:]]*`' can be used instead.

`(...)` Matches the enclosed pattern. This is used for grouping. If the `KSH_GLOB` option is set, then a '`@`', '`*`', '`+`', '`?`' or '`!`' immediately preceding the '`(`' is treated specially, as detailed below. The option `SH_GLOB` prevents bare parentheses from being used in this way, though the `KSH_GLOB` option is still available.

Note that grouping cannot extend over multiple directories: it is an error to have a '`/`' within a group (this only applies for patterns used in filename generation). There is one exception: a group of the form `(`*pat*`/)#` appearing as a complete path segment can match a sequence of directories. For example, `foo/(a*/)#bar` matches `foo/bar`, `foo/any/bar`, `foo/any/anyother/bar`, and so on.

x`|`*y* Matches either *x* or *y*. This operator has lower precedence than any other. The '`|`' character must be within parentheses, to avoid interpretation as a pipeline.

`^`*x* (Requires `EXTENDED_GLOB` to be set.) Matches anything except the pattern *x*. This has a higher precedence than '`/`', so '`^foo/bar`' will search directories in '`.`' except '`./foo`' for a file named '`bar`'.

x`~`*y* (Requires `EXTENDED_GLOB` to be set.) Match anything that matches the pattern *x* but does not match *y*. This has lower precedence than any operator except '`|`', so '`*/*~foo/bar`' will search for all files in all directories in '`.`' and then exclude '`foo/bar`' if there was such a match. Multiple patterns can be excluded by '*foo*`~`*bar*`~`*baz*'. In the exclusion pattern (*y*), '`/`' and '`.`' are not treated specially the way they usually are in globbing.

x`#` (Requires `EXTENDED_GLOB` to be set.) Matches zero or more occurrences of the pattern *x*. This operator has high precedence; '`12#`' is equivalent to '`1(2#)`', rather than '`(12)#`'. It is an error for an unquoted '`#`' to follow something which cannot be repeated; this includes an empty string, a pattern already followed by '`##`', or parentheses when part of a `KSH_GLOB` pattern (for example, '`!(`*foo*`)#`' is invalid and must be replaced by '`*(!(`*foo*`))`').

x`##` (Requires `EXTENDED_GLOB` to be set.) Matches one or more occurrences of the pattern *x*. This operator has high precedence; '`12##`' is equivalent to '`1(2##)`', rather than '`(12)##`'. No more than two active '`#`' characters may appear together.

13.8.2 ksh-like Glob Operators

If the `KSH_GLOB` option is set, the effects of parentheses can be modified by a preceding '`@`', '`*`', '`+`', '`?`' or '`!`'. This character need not be unquoted to have special effects, but the '`(`' must be.

`@(...)` Match the pattern in the parentheses. (Like '`(...)`'.)

`*(...)` Match any number of occurrences. (Like '`(...)#`'.)

`+(...)` Match at least one occurrence. (Like '`(...)##`'.)

`?(...)` Match zero or one occurrence. (Like '`(|...)`'.)

`!(...)` Match anything but the expression in parentheses. (Like '`(^(...))`'.)

13.8.3 Precedence

The precedence of the operators given above is (highest) '`^`', '`/`', '`~`', '`|`' (lowest); the remaining operators are simply treated from left to right as part of a string, with '`#`' and '`##`' applying to the shortest possible preceding unit (i.e. a character, '`?`', '`[...]`', '`<...>`', or a parenthesised expression). As mentioned above, a '`/`' used as a directory separator may not appear inside parentheses, while a '`|`' must do so; in patterns used in other contexts than filename generation (for example, in `case` statements and tests within '`[[...]]`'), a '`/`' is not special; and '`/`' is also not special after a '`~`' appearing outside parentheses in a filename pattern.

13.8.4 Globbing Flags

There are various flags which affect any text to their right up to the end of the enclosing group or to the end of the pattern; they require the `EXTENDED_GLOB` option. All take the form `(#`*X*`)` where *X* may have one of the following forms:

i Case insensitive: upper or lower case characters in the pattern match upper or lower case characters.

l Lower case characters in the pattern match upper or lower case characters; upper case characters in the pattern still only match upper case characters.

I Case sensitive: locally negates the effect of `i` or `l` from that point on.

b Activate backreferences for parenthesised groups in the pattern; this does not work in filename generation. When a pattern with a set of active parentheses is matched, the strings matched by the groups are stored in the array `$match`, the indices of the beginning of the matched parentheses in the array `$mbegin`, and the indices of the end in the array `$mend`, with the first element of each array corresponding to the first parenthesised group, and so on. These arrays are not otherwise special to the shell. The indices use the same convention as does parameter substitution, so that elements of `$mend` and `$mbegin` may be used in subscripts; the `KSH_ARRAYS` option is respected. Sets of globbing flags are not considered parenthesised groups; only the first nine active parentheses can be referenced.

For example,

```
foo="a string with a message"
if [[ $foo = (a|an)' '(#b)(*)' '* ]]; then
  print ${foo[$mbegin[1],$mend[1]]}
fi
```

prints '`string with a`'. Note that the first parenthesis is before the `(#b)` and does not create a backreference.

Backreferences work with all forms of pattern matching other than filename generation, but note that when performing matches on an entire array, such as `${`*array*`#`*pattern*`}`, or a global substitution, such as `${`*param*`//`*pat*`/`*repl*`}`, only the data for the last match remains available. In the case of global replacements this may still be useful. See the example for the `m` flag below.

The numbering of backreferences strictly follows the order of the opening parentheses from left to right in the pattern string, although sets of parentheses may be nested. There are special rules for parentheses followed by '`#`' or '`##`'. Only the last match

of the parenthesis is remembered: for example, in '`[[ abab = (#b)([ab])# ]]`', only the final '`b`' is stored in `match[1]`. Thus extra parentheses may be necessary to match the complete segment: for example, use '`X((ab|cd)#)Y`' to match a whole string of either '`ab`' or '`cd`' between '`X`' and '`Y`', using the value of `$match[1]` rather than `$match[2]`.

If the match fails none of the parameters is altered, so in some cases it may be necessary to initialise them beforehand. If some of the backreferences fail to match — which happens if they are in an alternate branch which fails to match, or if they are followed by `#` and matched zero times — then the matched string is set to the empty string, and the start and end indices are set to -1.

Pattern matching with backreferences is slightly slower than without.

B — Deactivate backreferences, negating the effect of the `b` flag from that point on.

m — Set references to the match data for the entire string matched; this is similar to backreferencing and does not work in filename generation. The flag must be in effect at the end of the pattern, i.e. not local to a group. The parameters `$MATCH`, `$MBEGIN` and `$MEND` will be set to the string matched and to the indices of the beginning and end of the string, respectively. This is most useful in parameter substitutions, as otherwise the string matched is obvious.

For example,

```
arr=(veldt jynx grimps waqf zho buck)
print ${arr//(#m)[aeiou]/${(U)MATCH}}
```

forces all the matches (i.e. all vowels) into uppercase, printing '`vEldt jynx grImps wAqf zhO bUck`'.

Unlike backreferences, there is no speed penalty for using match references, other than the extra substitutions required for the replacement strings in cases such as the example shown.

M — Deactivate the `m` flag, hence no references to match data will be created.

a*num* — Approximate matching: *num* errors are allowed in the string matched by the pattern. The rules for this are described in the next subsection.

`s`, `e` — Unlike the other flags, these have only a local effect, and each must appear on its own: '`(#s)`' and '`(#e)`' are the only valid forms. The '`(#s)`' flag succeeds only at the start of the test string, and the '`(#e)`' flag succeeds only at the end of the test string; they correspond to '`^`' and '`$`' in standard regular expressions. They are useful for matching path segments in patterns other than those in filename generation (where path segments are in any case treated separately). For example, '`*((#s)|/)test((#e)|/)*`' matches a path segment '`test`' in any of the following strings: `test`, `test/at/start`, `at/end/test`, `in/test/middle`.

Another use is in parameter substitution; for example '`${array/(#s)A*Z(#e)}`' will remove only elements of an array which match the complete pattern '`A*Z`'. There are other ways of performing many operations of this type, however the combination of the substitution operations '`/`' and '`//`' with the '`(#s)`' and '`(#e)`' flags provides a single simple and memorable method.

Note that assertions of the form '`(^(#s))`' also work, i.e. match anywhere except at the start of the string, although this actually means 'anything except a zero-length portion at the start of the string'; you need to use '`(""~(#s))`' to match a zero-length portion of the string not at the start.

For example, the test string `fooxx` can be matched by the pattern `(#i)FOOXX`, but not by `(#l)FOOXX`, `(#i)FOO(#I)XX` or `((#i)FOOX)X`. The string `(#ia2)readme` specifies case-insensitive matching of `readme` with up to two errors.

When using the ksh syntax for grouping both `KSH_GLOB` and `EXTENDED_GLOB` must be set and the left parenthesis should be preceded by `@`. Note also that the flags do not affect letters inside `[...]` groups, in other words `(#i)[a-z]` still matches only lowercase letters. Finally, note that when examining whole paths case-insensitively every directory must be searched for all files which match, so that a pattern of the form `(#i)/foo/bar/...` is potentially slow.

13.8.5 Approximate Matching

When matching approximately, the shell keeps a count of the errors found, which cannot exceed the number specified in the `(#a`*num*`)` flags. Four types of error are recognised:

1. Different characters, as in `fooxbar` and `fooybar`.
2. Transposition of characters, as in `banana` and `abnana`.
3. A character missing in the target string, as with the pattern `road` and target string `rod`.
4. An extra character appearing in the target string, as with `stove` and `strove`.

Thus, the pattern `(#a3)abcd` matches `dcba`, with the errors occurring by using the first rule twice and the second once, grouping the string as `[d][cb][a]` and `[a][bc][d]`.

Non-literal parts of the pattern must match exactly, including characters in character ranges: hence `(#a1)???` matches strings of length four, by applying rule 4 to an empty part of the pattern, but not strings of length two, since all the `?` must match. Other characters which must match exactly are initial dots in filenames (unless the `GLOB_DOTS` option is set), and all slashes in filenames, so that `a/bc` is two errors from `ab/c` (the slash cannot be transposed with another character). Similarly, errors are counted separately for non-contiguous strings in the pattern, so that `(ab|cd)ef` is two errors from `aebf`.

When using exclusion via the `~` operator, approximate matching is treated entirely separately for the excluded part and must be activated separately. Thus, `(#a1)README~READ_ME` matches `READ.ME` but not `READ_ME`, as the trailing `READ_ME` is matched without approximation. However, `(#a1)README~(#a1)READ_ME` does not match any pattern of the form `READ?ME` as all such forms are now excluded.

Apart from exclusions, there is only one overall error count; however, the maximum errors allowed may be altered locally, and this can be delimited by grouping. For example, `(#a1)cat((#a0)dog)fox` allows one error in total, which may not occur in the `dog` section, and the pattern `(#a1)cat(#a0)dog(#a1)fox` is equivalent. Note that the point at which an error is first found is the crucial one for establishing whether to use approximation; for example, `(#a1)abc(#a0)xyz` will not match `abcdxyz`, because the error occurs at the '`x`', where approximation is turned off.

Entire path segments may be matched approximately, so that '`(#a1)/foo/d/is/available/at/the/bar`' allows one error in any path segment. This is much less efficient than without the `(#a1)`, however, since every directory in the path must be scanned for a possible approximate match. It is best to place the `(#a1)` after any path segments which are known to be correct.

13.8.6 Recursive Globbing

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 '`(*/)#`'; note that this therefore matches files in the current directory as well as subdirectories. Thus:

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

or

```
ls **/bar
```

does a recursive directory search for files named '`bar`' (potentially including the file '`bar`' in the current directory). This form does not follow symbolic links; the alternative form '`***/`' does, but is otherwise identical. Neither of these can be combined with other forms of globbing within the same path segment; in that case, the '`*`' operators revert to their usual effect.

13.8.7 Glob Qualifiers

Patterns used for filename generation may 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.

If the option `BARE_GLOB_QUAL` is set, then a trailing set of parentheses containing no '`|`' or '`(`' characters (or '`~`' if it is special) is taken as a set of glob qualifiers. A glob subexpression that would normally be taken as glob qualifiers, for example '`(^x)`', can be forced to be treated as part of the glob pattern by doubling the parentheses, in this case producing '`((^x))`'.

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)
`f`*spec*	files with access rights matching *spec*. This *spec* may be a octal number optionally preceded by a '`=`', a '`+`', or a '`-`'. If none of these characters is given, the behavior is the same as for '`=`'. The octal number describes the mode bits to be expected, if combined with a '`=`', the value given must match the file-modes exactly, with a '`+`', at least the bits in the given number must be set in the file-modes, and with

a '-', the bits in the number must not be set. Giving a '?' instead of a octal digit anywhere in the number ensures that the corresponding bits in the file-modes are not checked, this is only useful in combination with '='.

If the qualifier 'f' is followed by any other character anything up to the next matching character ('[', '{', and '<' match ']', '}', and '>' respectively, any other character matches itself) is taken as a list of comma-separated *sub-spec*s. Each *sub-spec* may be either a octal number as described above or a list of any of the characters 'u', 'g', 'o', and 'a', followed by a '=', a '+', or a '-', followed by a list of any of the characters 'r', 'w', 'x', 's', and 't', or a octal digit. The first list of characters specify which access rights are to be checked. If a 'u' is given, those for the owner of the file are used, if a 'g' is given, those of the group are checked, a 'o' means to test those of other users, and the 'a' says to test all three groups. The '=', '+', and '-' again says how the modes are to be checked and have the same meaning as described for the first form above. The second list of characters finally says which access rights are to be expected: 'r' for read access, 'w' for write access, 'x' for the right to execute the file (or to search a directory), 's' for the setuid and setgid bits, and 't' for the sticky bit.

Thus, '*(f70?)' gives the files for which the owner has read, write, and execute permission, and for which other group members have no rights, independent of the permissions for other users. The pattern '*(f-100)' gives all files for which the owner does not have execute permission, and '*(f:gu+w,o-rx:)' gives the files for which the owner and the other members of the group have at least write permission, and for which other users don't have read or execute permission.

e*string* The *string* will be executed as shell code. The filename will be included in the list if and only if the code returns a zero status (usually the status of the last command). The first character after the 'e' will be used as a separator and anything up to the next matching separator will be taken as the *string*; '[', '{', and '<' match ']', '}', and '>', respectively, while any other character matches itself. Note that expansions must be quoted in the *string* to prevent them from being expanded before globbing is done.

During the execution of *string* the filename currently being tested is available in the parameter REPLY; the parameter may be altered to a string to be inserted into the list instead of the original filename. In addition, the parameter reply may be set to an array or a string, which overrides the value of REPLY. If set to an array, the latter is inserted into the command line word by word.

For example, suppose a directory contains a single file 'lonely'. Then the expression '*(e:'reply=(${REPLY}{1,2})':)' will cause the words 'lonely1 lonely2' to be inserted into the command line. Note the quotation marks.

d*dev* files on the device *dev*

l[-|+]*ct* files having a link count less than *ct* (-), greater than *ct* (+), or 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 *id* if it is a number, if not, than 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 the user ID of this user will be taken (e.g. 'u:foo:' or 'u[foo]' for user 'foo')

g*id* like u*id* but with group IDs or names

`a[Mwhms][-|+]`*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`', '`m`' or '`s`' (e.g. '`ah5`') cause the check to be performed with months (of 30 days), weeks, hours, minutes or seconds instead of days, respectively. For instance, '`echo *(ah-5)`' would echo files accessed within the last five hours.

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

`c[Mwhms][-|+]`*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 filenames, analogous to the `LIST_TYPES` option, 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

`n`
: sets the `NUMERIC_GLOB_SORT` option for the current pattern

`o`*c*
: specifies how the names of the files should be sorted. If *c* is `n` they are sorted by name (the default); if it is `L` they are sorted depending on the size (length) of the files; if `l` they are sorted by the number of links; if `a`, `m`, or `c` they are sorted by the time of the last access, modification, or inode change respectively; if `d`, files in subdirectories appear before those in the current directory at each level of the search — this is best combined with other criteria, for example '`odon`' to sort on names for files within the same directory. Note that `a`, `m`, and `c` compare the age against the current time, hence the first name in the list is the youngest file. Also note that the modifiers `^` and `-` are used, so '`*(^-oL)`' gives a list of all files sorted by file size in descending order, following any symbolic links.

`O`*c*
: like '`o`', but sorts in descending order; i.e. '`*(^oc)`' is the same as '`*(Oc)`' and '`*(^Oc)`' is the same as '`*(oc)`'; '`Od`' puts files in the current directory before those in subdirectories at each level of the search.

`[`*beg*`[,`*end*`]]`
: specifies which of the matched filenames should be included in the returned list. The syntax is the same as for array subscripts. *beg* and the optional *end* may be mathematical expressions. As in parameter subscripting they may be negative to make them count from the last match backward. E.g.: '`*(-OL[1,3])`' gives a list of the names of the three largest files.

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). Some qualifiers, however, affect all matches generated, independent of the sublist in which they are given. These are the qualifiers '`M`', '`T`', '`N`', '`D`', '`n`', '`o`', '`O`' and the subscripts given in brackets ('`[...]`').

If a '`:`' appears in a qualifier list, the remainder of the expression in parenthesis is interpreted as a modifier (see Section 13.1.4 [Modifiers], page 27 in Section 13.1 [History Expansion], page 26). 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 actual 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`.

14 Parameters

14.1 Description

A parameter has a name, a value, and a number of attributes. A name may be any sequence of alphanumeric characters and underscores, or the single characters '`*`', '`@`', '`#`', '`?`', '`-`', '`$`', or '`!`'. The value may be a *scalar* (a string), an integer, an array (indexed numerically), or an *associative* array (an unordered set of name-value pairs, indexed by name). To declare the type of a parameter, or to assign a scalar or integer value to a parameter, use the `typeset` builtin.

The value of a scalar or integer 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. See Section 14.2 [Array Parameters], page 46 for additional forms of assignment.

To refer to the value of a parameter, write '`$`*name*' or '`${`*name*`}`'. See Section 13.3 [Parameter Expansion], page 29 for complete details.

In the parameter lists that follow, the mark '<S>' indicates that the parameter is special. Special parameters cannot have their type changed, and they stay special even if unset. '<Z>' indicates that the parameter does not exist when the shell initializes in `sh` or `ksh` emulation mode.

14.2 Array Parameters

To assign an array value, write one of:

`set -A` *name value ...*

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

If no parameter *name* exists, an ordinary array parameter is created. If the parameter *name* exists and is a scalar, it is replaced by a new array. Ordinary array parameters may also be explicitly declared with:

`typeset -a` *name*

Associative arrays *must* be declared before assignment, by using:

`typeset -A` *name*

When *name* refers to an associative array, the list in an assignment is interpreted as alternating keys and values:

set -A *name key value* ...

name`=(`*key value* ...`)`

Every *key* must have a *value* in this case. Note that this assigns to the entire array, deleting any elements that do not appear in the list.

To create an empty array (including associative arrays), use one of:

`set -A` *name*

name`=()`

14.2.1 Array Subscripts

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 in which case they are numbered from zero.

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 works, as bracketed expressions otherwise are not treated as subscripts.

The same subscripting syntax is used for associative arrays, except that no arithmetic expansion is applied to *exp*. However, the parsing rules for arithmetic expressions still apply, which affects the way that certain special characters must be protected from interpretation. See *Subscript Parsing* below for details.

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] $foo[2] ..."`', whereas '`"$foo[@]"`' evaluates to '`"$foo[1]" "$foo[2]" ...`'. For associative arrays, '`[*]`' or '`[@]`' evaluate to all the values (not the keys, but see *Subscript Flags* below), in no particular order. When an array parameter is referenced as '$*name*' (with no subscript) it evaluates to '$*name*`[*]`', unless the `KSH_ARRAYS` option is set in which case it evaluates to '`${`*name*`[0]}`' (for an associative array, this means the value of the key '`0`', which may not exist even if there are values for other keys).

A subscript of the form '`[`*exp1*`,`*exp2*`]`' selects all elements in the range *exp1* to *exp2*, inclusive. (Associative arrays are unordered, and so do not support ranges.) 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`'.

14.2.2 Array Element Assignment

A subscript may be used on the left side of an assignment like so:

name`[`*exp*`]=`*value*

In this form of assignment the element or range specified by *exp* is replaced by the expression on the right side. An array (but not an associative array) may be created by assignment to a range or element. Arrays do not nest, so assigning a parenthesized list of values to an element or range changes the number of elements in the array, shifting the other elements to accommodate the new values. (This is not supported for associative arrays.)

This syntax also works as an argument to the `typeset` command:

`typeset "`*name*`[`*exp*`]"=`*value*

The *value* may *not* be a parenthesized list in this case; only single-element assignments may be made with `typeset`. Note that quotes are necessary in this case to prevent the brackets from being interpreted as filename generation operators. The `noglob` precommand modifier could be used instead.

To delete an element of an ordinary array, assign '`()`' to that element. To delete an element of an associative array, use the `unset` command:

`unset "`*name*`[`*exp*`]"`

14.2.3 Subscript Flags

If the opening bracket, or the comma in a range, in any subscript expression is directly followed by an opening parenthesis, the string up to the matching closing one is considered to be a list of flags, as in '*name*`[(`*flags*`)`*exp*`]`'. The flags currently understood are:

`w` If the parameter subscripted is a scalar than this flag makes subscripting work on words instead of characters. The default word separator is whitespace.

`s:`*string*`:` This gives 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 subscripting work on lines instead of characters, i.e. with elements separated by newlines. This is a shorthand for '`pws:\n:`'.

`r` Reverse subscripting: 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). The subscript used is the number of the matching element, so that pairs of subscripts such as '`$foo[(r)??,3]`' and '`$foo[(r)??,(r)f*]`' are possible. If the parameter is an associative array, only the value part of each pair is compared to the pattern, and the result is that value. Reverse subscripts may be used for assigning to ordinary array elements, but not for assigning to associative arrays.

`R` Like '`r`', but gives the last match. For associative arrays, gives all possible matches.

`i` Like '`r`', but gives the index of the match instead; this may not be combined with a second argument. On the left side of an assignment, behaves like '`r`'. For associative arrays, the key part of each pair is compared to the pattern, and the first matching key found is the result.

`I` Like '`i`', but gives the index of the last match, or all possible matching keys in an associative array.

`k` If used in a subscript on an associative array, this flag causes the keys to be interpreted as patterns, and returns the value for the first key found where *exp* is matched by the key. This flag does not work on the left side of an assignment to an associative array element. If used on another type of parameter, this behaves like '`r`'.

`K` On an associative array this is like '`k`' but returns all values where *exp* is matched by the keys. On other types of parameters this has the same effect as '`R`'.

`n:`*expr*`:` If combined with '`r`', '`R`', '`i`' or '`I`', makes them give the *n*th or *n*th last match (if *expr* evaluates to *n*). This flag is ignored when the array is associative.

`b:`*expr*`:` If combined with '`r`', '`R`', '`i`' or '`I`', makes them begin at the *n*th or *n*th last element, word, or character (if *expr* evaluates to *n*). This flag is ignored when the array is associative.

`e` This flag has no effect and for ordinary arrays is retained for backward compatibility only. For associative arrays, this flag can be used to force `*` or `@` to be interpreted as a single key rather than as a reference to all values. This flag may be used on the left side of an assignment.

See *Parameter Expansion Flags* (Section 13.3 [Parameter Expansion], page 29) for additional ways to manipulate the results of array subscripting.

14.2.4 Subscript Parsing

This discussion applies mainly to associative array key strings and to patterns used for reverse subscripting (the '`r`', '`R`', '`i`', etc. flags), but it may also affect parameter substitutions that appear as part of an arithmetic expression in an ordinary subscript.

The basic rule to remember when writing a subscript expression is that all text between the opening '`[`' and the closing '`]`' is interpreted *as if* it were in double quotes (Section 5.8 [Quoting], page 11). However, unlike double quotes which normally cannot nest, subscript expressions may appear inside double-quoted strings or inside other subscript expressions (or both!), so the rules have two important differences.

The first difference is that brackets ('`[`' and '`]`') must appear as balanced pairs in a subscript expression unless they are preceded by a backslash ('`\`'). Therefore, within a subscript expression (and unlike true double-quoting) the sequence '`\[`' becomes '`[`', and similarly '`\]`' becomes '`]`'. This applies even in cases where a backslash is not normally required; for example, the pattern '`[^[]`' (to match any character other than an open bracket) should be written '`[^\[]`' in a reverse-subscript pattern. However, note that '`\[^\[\]`' and even '`\[^[]`' mean the *same* thing, because backslashes are always stripped when they appear before brackets!

The same rule applies to parentheses ('`(`' and '`)`') and braces ('`{`' and '`}`'): they must appear either in balanced pairs or preceded by a backslash, and backslashes that protect parentheses or braces are removed during parsing. This is because parameter expansions may be surrounded balanced braces, and subscript flags are introduced by balanced parenthesis.

The second difference is that a double-quote ('`"`') may appear as part of a subscript expression without being preceded by a backslash, and therefore that the two characters '`\"`' remain as two characters in the subscript (in true double-quoting, '`\"`' becomes '`"`'). However, because of the standard shell quoting rules, any double-quotes that appear must occur in balanced pairs unless preceded by a backslash. This makes it more difficult to write a subscript expression that contains an odd number of double-quote characters, but the reason for this difference is so that when a subscript expression appears inside true double-quotes, one can still write '`\"`' (rather than '`\\\"`') for '`"`'.

To use an odd number of double quotes as a key in an assignment, use the `typeset` builtin and an enclosing pair of double quotes; to refer to the value of that key, again use double quotes:

```
typeset -A aa
typeset "aa[one\"two\"three\"quotes]"=QQQ
print "$aa[one\"two\"three\"quotes]"
```

It is important to note that the quoting rules do not change when a parameter expansion with a subscript is nested inside another subscript expression. That is, it is not necessary to use additional backslashes within the inner subscript expression; they are removed only once, from the innermost subscript outwards. Parameters are also expanded from the innermost subscript first, as each expansion is encountered left to right in the outer expression.

A further complication arises from a way in which subscript parsing is *not* different from double quote parsing. As in true double-quoting, the sequences '`\*`', and '`\@`' remain as two characters when they appear in a subscript expression. To use a literal '`*`' or '`@`' as an associative array key, the '`e`' flag must be used:

```
typeset -A aa
aa[(e)*]=star
print $aa[(e)*]
```

A last detail must be considered when reverse subscripting is performed. Parameters appearing in the subscript expression are first expanded and then the complete expression is interpreted as a pattern. This has two effects: first, parameters behave as if `GLOB_SUBST` were on (and it cannot be turned off); second, backslashes are interpreted twice, once when parsing the array subscript and again when parsing the pattern. In a reverse subscript, it's necessary to use *four* backslashes to cause a single backslash to match literally in the pattern. For complex patterns, it is often easiest to assign the desired pattern to a parameter and then refer to that parameter in the subscript, because then the backslashes, brackets, parentheses, etc., are seen only when the complete expression is converted to a pattern. To match the value of a parameter literally in a reverse subscript, rather than as a pattern, use '`${(q)`*name*`}`' (Section 13.3 [Parameter Expansion], page 29) to quote the expanded value.

Note that the '`k`' and '`K`' flags are reverse subscripting for an ordinary array, but are *not* reverse subscripting for an associative array! (For an associative array, the keys in the array itself are interpreted as patterns by those flags; the subscript is a plain string in that case.)

One final note, not directly related to subscripting: the numeric names of positional parameters (Section 14.3 [Positional Parameters], page 50) are parsed specially, so for example '`$2foo`' is equivalent to '`${2}foo`'. Therefore, to use subscript syntax to extract a substring from a positional parameter, the expansion must be surrounded by braces; for example, '`${2[3,5]}`' evaluates to the third through fifth characters of the second positional parameter, but '`$2[3,5]`' is the entire second parameter concatenated with the filename generation pattern '`[3,5]`'.

14.3 Positional Parameters

The positional parameters provide access to the command-line arguments of a shell function, shell script, or the shell itself; see Chapter 3 [Invocation], page 4, and also Chapter 8 [Functions], page 14. 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 '`$argv[`*n*`]`', etc., is equivalent to simply '`$`*n*'.

Positional parameters may be changed after the shell or function starts by using the `set` builtin, by assigning to the `argv` array, or by direct assignment of the form '*n*`=`*value*' where *n* is the number of the positional parameter to be changed. This also creates (with empty values) any of the positions from 1 to *n* that do not already have values. Note that, because the positional parameters form an array, an array assignment of the form '*n*`=(`*value* ...`)`' is allowed, and has the effect of shifting all the values at positions greater than *n* by as many positions as necessary to accommodate the new values.

14.4 Local Parameters

Shell function executions delimit scopes for shell parameters. (Parameters are dynamically scoped.) The `typeset` builtin, and its alternative forms `declare`, `integer`, `local` and `readonly` (but not `export`), can be used to declare a parameter as being local to the innermost scope.

When a parameter is read or assigned to, the innermost existing parameter of that name is used. (That is, the local parameter hides any less-local parameter.) However, assigning to a non-existent parameter, or declaring a new parameter with `export`, causes it to be created in the *outer*most scope.

Local parameters disappear when their scope ends. `unset` can be used to delete a parameter while it is still in scope; any outer parameter of the same name remains hidden.

Special parameters may also be made local; they retain their special attributes unless either the existing or the newly-created parameter has the `-h` (hide) attribute. This may have unexpected effects: there is no default value, so if there is no assignment at the point the variable is made local, it will be set to an empty value (or zero in the case of integers). The following:

```
typeset PATH=/new/directory:$PATH
```

is valid for temporarily allowing the shell or programmes called from it to find the programs in `/new/directory` inside a function.

Note that the restriction in older versions of zsh that local parameters were never exported has been removed.

14.5 Parameters Set By The Shell

The following parameters are automatically set by the shell:

`!` <S>
: The process ID of the last background command invoked.

`#` <S>
: The number of positional parameters in decimal. Note that some confusion may occur with the syntax `$#`*param* which substitutes the length of *param*. Use `${#}` to resolve ambiguities. In particular, the sequence '`$#-`...' in an arithmetic expression is interpreted as the length of the parameter `-`, q.v.

`ARGC` <S> <Z>
: Same as `#`.

`$` <S>
: The process ID of this shell.

`-` <S>
: Flags supplied to the shell on invocation or by the `set` or `setopt` commands.

`*` <S>
: An array containing the positional parameters.

`argv` <S> <Z>
: Same as `*`. Assigning to `argv` changes the local positional parameters, but `argv` is *not* itself a local parameter. Deleting `argv` with `unset` in any function deletes it everywhere, although only the innermost positional parameter array is deleted (so `*` and `@` in other scopes are not affected).

`@` <S>
: Same as `argv[@]`, even when `argv` is not set.

`?` <S>
: The exit value returned by the last command.

`0` <S>
: The name used to invoke the current shell. If the `FUNCTION_ARGZERO` option is set, this is set temporarily within a shell function to the name of the function, and within a sourced script to the name of the script.

`status` <S> <Z>
: Same as `?`.

`pipestatus <S> <Z>`
: An array containing the exit values returned by all commands in the last pipeline.

`_ <S>`
: 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.

`CPUTYPE`
: The machine type (microprocessor class or machine model), as determined at run time.

`EGID <S>`
: 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=`*gid*`; command)`'

`EUID <S>`
: 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=`*uid*`; command)`'

`ERRNO <S>`
: The value of errno (see man page errno(3)) as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes.

`GID <S>`
: The real 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 <S>`
: The line number of the current line within the current script, sourced file, or shell function being executed, whichever was started most recently. Note that in the case of shell functions the line number refers to the function as it appeared in the original definition, not necessarily as displayed by the `functions` builtin.

`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. This is set when the shell initializes and whenever the directory changes.

`OPTARG <S>`
: The value of the last option argument processed by the `getopts` command.

`OPTIND <S>`
: The index of the last option argument processed by the `getopts` command.

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

`PPID <S>`
: The process ID of the parent of the shell.

`PWD`
: The present working directory. This is set when the shell initializes and whenever the directory changes.

`RANDOM <S>`
: 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` <S>
: 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` <S>
: 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` <S>
: The idle time of the tty associated with the shell in seconds or -1 if there is no such tty.

`UID` <S>
: The real 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` <S>
: The username corresponding to the real 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.

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

`ZSH_VERSION`
: The version number of this zsh.

14.6 Parameters Used By The Shell

The following parameters are used by the shell.

In cases where there are two parameters with an upper- and lowercase form of the same name, such as `path` and `PATH`, the lowercase form is an array and the uppercase form is a scalar with the elements of the array joined together by colons. These are similar to tied parameters created via '`typeset -T`'. The normal use for the colon-separated form is for exporting to the environment, while the array form is easier to manipulate within the shell. Note that unsetting either of the pair will unset the other; they retain their special properties when recreated, and recreating one of the pair will recreate the other.

`ARGV0`
: If exported, its value is used as the `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` <S> <Z> (`CDPATH` <S>)
: An array (colon-separated list) of directories specifying the search path for the `cd` command.

`COLUMNS` <S>
: 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` <S> <Z> (`FIGNORE` <S>)
: 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` <S> <Z> (`FPATH` <S>)
: 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` <S>
: Three characters used by the shell's history and lexical analysis mechanism. The first character signals the start of a history expansion (default '`!`'). The second character signals the start of a quick history substitution (default '`^`'). The third character is the comment character (default '`#`').

`HISTCHARS` <S> <Z>
: Same as `histchars`. (Deprecated.)

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

`HISTSIZE` <S>
: The maximum number of events stored in the internal history list. If you use the `HIST_EXPIRE_DUPS_FIRST` option, setting this value larger than the `SAVEHIST` size will give you the difference as a cushion for saving duplicated history events.

`HOME` <S> The default argument for the `cd` command.

`IFS` <S> Internal field separators (by default space, tab, newline and NUL), that are used to separate words which result from command or parameter expansion 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` <S> This variable determines the locale category for any category not specifically selected via a variable starting with '`LC_`'.

`LC_ALL` <S>
: This variable overrides the value of the '`LANG`' variable and the value of any of the other variables starting with '`LC_`'.

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

`LC_CTYPE` <S>
: This variable determines the locale category for character handling functions.

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

`LC_NUMERIC` <S>
: This variable affects the decimal point character and thousands separator character for the formatted input/output functions and string conversion functions. Note that zsh ignores this setting when parsing floating point mathematical expressions.

`LC_TIME` <S>
: This variable determines the locale category for date and time formatting in prompt escape sequences.

`LINES` <S> 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 matches to list without asking first. If the value is negative, the list will be shown if it spans at most as many lines as given by the absolute value. If set to zero, the shell asks only if the top of 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` <S> <Z> (`MAILPATH` <S>)
: 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 expansion 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` <S> <Z> (`MANPATH` <S> <Z>)
: 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.

`module_path` <S> <Z> (`MODULE_PATH` <S>)
: An array (colon-separated list) of directories that `zmodload` searches for dynamically loadable modules. This is initialized to a standard pathname, usually '`/usr/local/lib/zsh/$ZSH_VERSION`'. (The '`/usr/local/lib`' part varies from installation to installation.) For security reasons, any value set in the environment when the shell is started will be ignored.

  These parameters only exist if the installation supports dynamic module loading.

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

`path` <S> <Z> (`PATH` <S>)
: 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 <S>`
: This string is output whenever the line editor exits. It usually contains termcap strings to reset the terminal.

`PROMPT <S> <Z>`
`PROMPT2 <S> <Z>`
`PROMPT3 <S> <Z>`
`PROMPT4 <S> <Z>`
: Same as `PS1`, `PS2`, `PS3` and `PS4`, respectively.

`prompt <S> <Z>`
: Same as `PS1`.

`PS1 <S>`
: The primary prompt string, printed before a command is read. the default is '`%m%# `'. It undergoes a special form of expansion before being displayed; see Chapter 12 [Prompt Expansion], page 22.

`PS2 <S>`
: The secondary prompt, printed when the shell needs more information to complete a command. It is expanded in the same way as `PS1`. The default is '`%_> `', which displays any shell constructs or quotation marks which are currently being processed.

`PS3 <S>`
: Selection prompt used within a `select` loop. It is expanded in the same way as `PS1`. The default is '`?# `'.

`PS4 <S>`
: The execution trace prompt. Default is '`+%N:%i> `', which displays the name of the current shell structure and the line number within it. In sh or ksh emulation, the default is '`+ `'.

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

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

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

`REPLY`
: This parameter is reserved by convention to pass string values between shell scripts and shell builtins in situations where a function call or redirection are impossible or undesirable. The `read` builtin and the `select` complex command may set `REPLY`, and filename generation both sets and examines its value when evaluating certain expressions. Some modules also employ `REPLY` for similar purposes.

`reply`
: As `REPLY`, but for array values rather than strings.

`RPROMPT <S>`
`RPS1 <S>`
: 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. It is expanded in the same way as `PS1`.

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

`SPROMPT <S>`
: 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.

`TERM` <S> The type of terminal in use. This is used when looking up termcap sequences. An assignment to `TERM` causes zsh to re-initialize the terminal, even if the value does not change (e.g., '`TERM=$TERM`'). It is necessary to make such an assignment upon any change to the terminal definition database or terminal type in order for the new settings to take effect.

`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` <S> <Z> (`WATCH` <S>)
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 events are reported as with '`all`' except `$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 rules 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 be present in any case.

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

`ZBEEP` If set, this gives a string of characters, which can use all the same codes as the `bindkey` command as described in Section 21.19 [The zsh/zle Module], page 208, that will be output to the terminal instead of beeping. This may have a visible instead of an audible effect; for example, the string '`\e[?5h\e[?5l`' on a vt100 or xterm will have the effect of flashing reverse video on and off (if you usually use reverse video, you should use the string '`\e[?5l\e[?5h`' instead). This takes precedence over the `NOBEEP` option.

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

15 Options

15.1 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 used to emulate *sh/ksh* (used when the `SH_OPTION_LETTERS` option is set). 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`'.

In strings of single letter options supplied to the shell at startup, trailing whitespace will be ignored; for example the string '`-f `' will be treated just as '`-f`', but the string '`-f i`' is an error. This is because many systems which implement the '`#!`' mechanism for calling scripts do not strip trailing whitespace.

15.2 Description of Options

In the following list, options set by default in all emulations are marked <D>; those set by default only in csh, ksh, sh, or zsh emulations are marked <C>, <K>, <S>, <Z> as appropriate. When listing options (by '`setopt`', '`unsetopt`', '`set -o`' or '`set +o`'), those turned on by default appear in the list prefixed with '`no`'. Hence (unless `KSH_OPTION_PRINT` is set), '`setopt`' shows all options whose settings are changed from the default.

`ALIASES` <D>
: Expand aliases.

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

`ALWAYS_LAST_PROMPT` <D>
: 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 is performed with the cursor within a word, and a full completion is inserted, the cursor is moved to the end of the word. That is, the cursor is moved to the end of the word if either a single match is inserted or menu completion is performed.

`APPEND_HISTORY` <D>
: 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.

`AUTO_CD` (`-J`)
: If a command is issued that can't be executed as a normal command, and the command is the name of a directory, perform the `cd` command to that directory.

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

`AUTO_MENU` <D>
: 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, that will be used by the '`%~`' and related prompt sequences, and will be available when completion is performed on a word starting with '`~`'. (Otherwise, the parameter must be used in the form '`~`*param*' first.)

`AUTO_PARAM_KEYS` <D>
: If a parameter name was completed and a following character (normally a space) automatically inserted, and the next character typed is one of those that have to come directly after the name (like '`}`', '`:`', etc.), the automatically added character is deleted, so that the character typed comes immediately after the parameter name. Completion in a brace expansion is affected similarly: the added character is a '`,`', which will be removed if '`}`' is typed next.

`AUTO_PARAM_SLASH` <D>
: If a parameter is completed whose content is the name of a directory, then add a trailing slash instead of a space.

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

`AUTO_REMOVE_SLASH` <D>
: When the last character resulting from a completion is a slash and the next character typed is a word delimiter, a slash, or a character that ends a command (such as a semicolon or an ampersand), remove the slash.

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

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

`BANG_HIST` (`+K`) <C> <Z>
: Perform textual history expansion, *csh*-style, treating the character '`!`' specially.

`BARE_GLOB_QUAL` <Z>
: In a glob pattern, treat a trailing set of parentheses as a qualifier list, if it contains no '`|`', '`(`' or (if special) '`~`' characters. See Section 13.8 [Filename Generation], page 38.

`BASH_AUTO_LIST`
: On an ambiguous completion, automatically list choices when the completion function is called twice in succession. This takes precedence over `AUTO_LIST`. The setting of `LIST_AMBIGUOUS` is respected. If `AUTO_MENU` is set, the menu behaviour will then start with the third press. Note that this will not work with `MENU_COMPLETE`, since repeated completion calls immediately cycle through the list in that case.

BEEP (+B) <D>
: Beep on error in ZLE.

BG_NICE (-6) <C> <Z>
: 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 13.6 [Brace Expansion], page 37.

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

C_BASES
: Output hexadecimal numbers in the standard C format, for example '`0xFF`' instead of the usual '`16#FF`'. If the option `OCTAL_ZEROES` is also set (it is not by default), octal numbers will be treated similarly and hence appear as '`077`' instead of '`8#77`'. This option has no effect on the choice of the output base, nor on the output of bases other than hexadecimal and octal. Note that these formats will be understood on input irrespective of the setting of `C_BASES`.

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 13.7 [Filename Expansion], page 37).

CHASE_DOTS
: When changing to a directory containing a path segment '`..`' which would otherwise be treated as canceling the previous segment in the path (in other words, '`foo/..`' would be removed from the path, or if '`..`' is the first part of the path, the last part of `$PWD` would be deleted), instead resolve the path to the physical directory. This option is overridden by `CHASE_LINKS`.

  For example, suppose `/foo/bar` is a link to the directory `/alt/rod`. Without this option set, '`cd /foo/bar/..`' changes to `/foo`; with it set, it changes to `/alt`. The same applies if the current directory is `/foo/bar` and '`cd ..`' is used. Note that all other symbolic links in the path will also be resolved.

CHASE_LINKS (-w)
: Resolve symbolic links to their true values when changing directory. This also has the effect of `CHASE_DOTS`, i.e. a '`..`' path segment will be treated as referring to the physical parent, even if the preceding path segment is a symbolic link.

CHECK_JOBS <Z>
: Report the status of background and suspended jobs before exiting a shell with job control; a second attempt to exit the shell will succeed. `NO_CHECK_JOBS` is best used only in combination with `NO_HUP`, else such jobs will be killed automatically.

  The check is omitted if the commands run from the previous command line included a '`jobs`' command, since it is assumed the user is aware that there are background or suspended jobs. A '`jobs`' command run from the `precmd` function is not counted for this purpose.

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

`COMPLETE_ALIASES`
: Prevents aliases on the command line from being internally substituted before completion is attempted. The effect is to make the alias a distinct command for completion purposes.

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

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

`CORRECT_ALL` (`-0`)
: Try to correct the spelling of all arguments in a line.

`CSH_JUNKIE_HISTORY` <C>
: A history reference without an event specifier will always refer to the previous command. Without this option, such a history reference refers to the same event as the previous history reference, defaulting to the previous command.

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

`CSH_JUNKIE_QUOTES` <C>
: Changes the rules for single- and double-quoted text to match that of *csh*. These require that embedded newlines be preceded by a backslash; unescaped newlines will cause an error message. In double-quoted strings, it is made impossible to escape '`$`', '`` ` ``' or '`"`' (and '`\`' itself no longer needs escaping). Command substitutions are only expanded once, and cannot be nested.

`CSH_NULLCMD` <C>
: Do not use the values of `NULLCMD` and `READNULLCMD` when running redirections with no command. This make such redirections fail (see Chapter 6 [Redirection], page 11).

`CSH_NULL_GLOB` <C>
: 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 `NOMATCH`.

`DVORAK`
: Use the Dvorak keyboard instead of the standard qwerty keyboard as a basis for examining spelling mistakes for the `CORRECT` and `CORRECT_ALL` options and the `spell-word` editor command.

`EQUALS` <Z>
: Perform `=` filename expansion. (See Section 13.7 [Filename Expansion], page 37.)

`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`) <D>
: Do execute commands. Without this option, commands are read and checked for syntax errors, but not executed. This option cannot be turned off in an interactive shell, except when '`-n`' is supplied to the shell at startup.

`EXTENDED_GLOB`
: Treat the '`#`', '`~`' and '`^`' characters as part of patterns for filename generation, etc. (An initial unquoted '`~`' always produces named directory expansion.)

`EXTENDED_HISTORY` <C>
: Save each command's beginning timestamp (in seconds since the epoch) and the duration (in seconds) to the history file. The format of this prefixed data is:

'`:<`*beginning time*`>:<`*elapsed seconds*`>:<`*command*`>`'.

`FLOW_CONTROL` <D>
: 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` <C> <Z>
: When executing a shell function or sourcing a script, set `$0` temporarily to the name of the function/script.

`GLOB` (`+F`, ksh: `+f`) <D>
: Perform filename generation (globbing). (See Section 13.8 [Filename Generation], page 38.)

`GLOBAL_EXPORT` (`<Z>`)
: If this option is set, passing the `-x` flag to the builtins `declare`, `float`, `integer`, `readonly` and `typeset` (but not `local`) will also set the `-g` flag; hence parameters exported to the environment will not be made local to the enclosing function, unless they were already or the flag `+g` is given explicitly. If the option is unset, exported parameters will be made local in just the same way as any other parameter.

  This option is set by default for backward compatibility; it is not recommended that its behaviour be relied upon. Note that the builtin `export` always sets both the `-x` and `-g` flags, and hence its effect extends beyond the scope of the enclosing function; this is the most portable way to achieve this behaviour.

`GLOBAL_RCS` (`-d`) <D>
: If this option is unset, the startup files `/etc/zprofile`, `/etc/zshrc`, `/etc/zlogin` and `/etc/zlogout` will not be run. It can be disabled and re-enabled at any time, including inside local startup files (`.zshrc`, etc.).

`GLOB_ASSIGN` <C>
: 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. '`foo=*`'). 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. '`foo=(*)`') 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 generate matches as for completion and cycle through them like `MENU_COMPLETE`. The matches are generated as if a '`*`' was added to the end of the word, or inserted at the cursor when `COMPLETE_IN_WORD` is set. This actually uses pattern matching, not globbing, so it works not only for files but for any completion, such as options, user names, etc.

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

`GLOB_SUBST` <C> <K> <S>
: Treat any characters resulting from parameter expansion as being eligible for file expansion and filename generation, and any characters resulting from command substitution as being eligible for filename generation. Braces (and commas in between) do not become eligible for expansion.

`HASH_CMDS` <D>
: Note the location of each command the first time it is executed. Subsequent invocations of the same command will use the saved location, avoiding a path search. If

this option is unset, no path hashing is done at all. However, when `CORRECT` is set, commands whose names do not appear in the functions or aliases hash tables are hashed in order to avoid reporting them as spelling errors.

`HASH_DIRS` <D>
: Whenever a command name is hashed, hash the directory containing it, as well as all directories that occur earlier in the path. Has no effect if neither `HASH_CMDS` nor `CORRECT` is set.

`HASH_LIST_ALL` <D>
: 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` <D>
: Beep when an attempt is made to access a history entry which isn't there.

`HIST_EXPIRE_DUPS_FIRST`
: If the internal history needs to be trimmed to add the current command line, setting this option will cause the oldest history event that has a duplicate to be lost before losing a unique event from the list. You should be sure to set the value of `HISTSIZE` to a larger number than `SAVEHIST` in order to give you some room for the duplicated events, otherwise this option will behave just like `HIST_IGNORE_ALL_DUPS` once the history fills up with unique events.

`HIST_FIND_NO_DUPS`
: When searching for history entries in the line editor, do not display duplicates of a line previously found, even if the duplicates are not contiguous.

`HIST_IGNORE_ALL_DUPS`
: If a new command line being added to the history list duplicates an older one, the older command is removed from the list (even if it is not the previous event).

`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`)
: Remove command lines from the history list when the first character on the line is a space, or when one of the expanded aliases contains a leading space. Note that the command lingers in the internal history until the next command is entered before it vanishes, allowing you to briefly reuse or edit the line. If you want to make it vanish right away without entering another command, type a space and press return.

`HIST_NO_FUNCTIONS`
: Remove function definitions from the history list. Note that the function lingers in the internal history until the next command is entered before it vanishes, allowing you to briefly reuse or edit the definition.

`HIST_NO_STORE`
: Remove the `history` (`fc -l`) command from the history list when invoked. Note that the command lingers in the internal history until the next command is entered before it vanishes, allowing you to briefly reuse or edit the line.

`HIST_REDUCE_BLANKS`
: Remove superfluous blanks from each command line being added to the history list.

`HIST_SAVE_NO_DUPS`
: When writing out the history file, older commands that duplicate newer ones are omitted.

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

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

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

`IGNORE_EOF` (`-7`)
: Do not exit on end-of-file. Require the use of `exit` or `logout` instead. However, ten consecutive EOFs will cause the shell to exit anyway, to avoid the shell hanging if its tty goes away.

  Also, if this option is set and the Zsh Line Editor is used, widgets implemented by shell functions can be bound to EOF (normally Control-D) without printing the normal warning message. This works only for normal widgets, not for completion widgets.

`INC_APPEND_HISTORY`
: This options works like `APPEND_HISTORY` except that new history lines are added to the `$HISTFILE` incrementally (as soon as they are entered), rather than waiting until the shell is killed. The file is periodically trimmed to the number of lines specified by `$SAVEHIST`, but can exceed this value between trimmings.

`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`) <K> <S>
: Allow comments even in interactive shells.

`KSH_ARRAYS` <K> <S>
: 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_AUTOLOAD` <K> <S>
: Emulate *ksh* function autoloading. This means that when a function is autoloaded, the corresponding file is merely executed, and must define the function itself. (By default, the function is defined to the contents of the file. However, the most common *ksh*-style case - of the file containing only a simple definition of the function - is always handled in the *ksh*-compatible manner.)

`KSH_GLOB` <K>
: In pattern matching, the interpretation of parentheses is affected by a preceding '`@`', '`*`', '`+`', '`?`' or '`!`'. See Section 13.8 [Filename Generation], page 38.

`KSH_OPTION_PRINT` <K>
: Alters the way options settings are printed: instead of separate lists of set and unset options, all options are shown, marked 'on' if they are in the non-default state, 'off' otherwise.

KSH_TYPESET <K>
: Alters the way arguments to the `typeset` family of commands, including `declare`, `export`, `float`, `integer`, `local` and `readonly`, are processed. Without this option, zsh will perform normal word splitting after command and parameter expansion in arguments of an assignment; with it, word splitting does not take place in those cases.

LIST_AMBIGUOUS <D>
: This option works when `AUTO_LIST` or `BASH_AUTO_LIST` is also set. If there is an unambiguous prefix to insert on the command line, that is done without a completion list being displayed; in other words, auto-listing behaviour only takes place when nothing would be inserted. In the case of `BASH_AUTO_LIST`, this means that the list will be delayed to the third call of the function.

LIST_BEEP <D>
: Beep on an ambiguous completion. More accurately, this forces the completion widgets to return status 1 on an ambiguous completion, which causes the shell to beep if the option `BEEP` is also set; this may be modified if completion is called from a user-defined widget.

LIST_PACKED
: Try to make the completion list smaller (occupying less lines) by printing the matches in columns with different widths.

LIST_ROWS_FIRST
: Lay out the matches in completion lists sorted horizontally, that is, the second match is to the right of the first one, not under it as usual.

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

LOCAL_OPTIONS <K>
: 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. A shell function can also guarantee itself a known shell configuration with a formulation like '`emulate -L zsh`'; the `-L` activates `LOCAL_OPTIONS`.

LOCAL_TRAPS <K>
: If this option is set when a signal trap is set inside a function, then the previous status of the trap for that signal will be restored when the function exits. Note that this option must be set *prior* to altering the trap behaviour in a function; unlike `LOCAL_OPTIONS`, the value on exit from the function is irrelevant. However, it does not need to be set before any global trap for that to be correctly restored by a function. For example,

```
unsetopt localtraps
trap - INT
fn() { setopt localtraps; trap '' INT; sleep 3; }
```

  will restore normally handling of `SIGINT` after the function exits.

LOGIN (`-l`, ksh: `-l`)
: This is a login shell. If this option is not explicitly set, the shell is a login shell if the first character of the `argv[0]` passed to the shell is a '`-`'.

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

`MAGIC_EQUAL_SUBST`
: All unquoted arguments of the form '*anything*`=`*expression*' appearing after the command name have filename 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; it is passed to the command as a single argument, and not used as an actual parameter assignment. For example, in `echo foo=~/bar:~/rod`, both occurrences of `~` would be replaced. Note that this happens anyway with `typeset` and similar statements.

  This option respects the setting of the `KSH_TYPESET` option. In other words, if both options are in effect, arguments looking like assignments will not undergo wordsplitting.

`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` <Z>
: Perform implicit *tee*s or *cat*s when multiple redirections are attempted (see Chapter 6 [Redirection], page 11).

`NOMATCH` (`+3`) <C> <Z>
: 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`) <Z>
: 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.

`OCTAL_ZEROES` <S>
: Interpret any integer constant beginning with a 0 as octal, per IEEE Std 1003.2-1992 (ISO 9945-2:1993). This is not enabled by default as it causes problems with parsing of, for example, date and time strings with leading zeroes.

`OVERSTRIKE`
: 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 or she types '`X11/xinit`', the command '`/usr/local/bin/X11/xinit`' will be executed (assuming it exists). Commands explicitly beginning with '`/`', '`./`' or '`../`' are not subject to the path search. This also applies to the `.` builtin.

  Note that subdirectories of the current directory are always searched for executables specified in this form. This takes place before any search indicated by this option, and regardless of whether '`.`' or the current directory appear in the command search path.

`POSIX_BUILTINS` <K> <S>
: 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_EIGHT_BIT`
: Print eight bit characters literally in completion lists, etc. This option is not necessary if your system correctly returns the printability of eight bit characters (see man page ctype(3)).

`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_BANG` <K>
: If set, '`!`' is treated specially in prompt expansion. See Chapter 12 [Prompt Expansion], page 22.

`PROMPT_CR` (`+V`) <D>
: Print a carriage return just before printing a prompt in the line editor. This is on by default as multi-line editing is only possible if the editor knows where the start of the line appears.

`PROMPT_PERCENT` <C> <Z>
: If set, '`%`' is treated specially in prompt expansion. See Chapter 12 [Prompt Expansion], page 22.

`PROMPT_SUBST` <K>
: If set, *parameter expansion*, *command substitution* and *arithmetic expansion* are performed in prompts.

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

PUSHD_MINUS
: Exchanges the meanings of '+' and '-' when used with a number to specify a directory in the stack.

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. Note this does not apply in quoted strings using the format $'...', where a backslashed single quote can be used.

RCS (+f) <D>
: After **/etc/zshenv** is sourced on startup, source the **.zshenv**, **/etc/zprofile**, **.zprofile**, **/etc/zshrc**, **.zshrc**, **/etc/zlogin**, **.zlogin**, and **.zlogout** files, as described in Chapter 4 [Files], page 6. If this option is unset, the **/etc/zshenv** file is still sourced, but any of the others will not be; it can be set at any time to prevent the remaining startup files after the currently executing one from being sourced.

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

RESTRICTED (-r)
: Enables restricted mode. This option cannot be changed using **unsetopt**, and setting it inside a function always changes it globally regardless of the LOCAL_OPTIONS option. See Section 3.3 [Restricted Shell], page 5.

RM_STAR_SILENT (-H) <K> <S>
: Do not query the user before executing '**rm ***' or '**rm path/***'.

RM_STAR_WAIT
: If querying the user before executing '**rm ***' or '**rm path/***', first wait ten seconds and ignore anything typed in that time. This avoids the problem of reflexively answering 'yes' to the query when one didn't really mean it. The wait and query can always be avoided by expanding the '*' in ZLE (with tab).

SHARE_HISTORY <K>
: This option both imports new commands from the history file, and also causes your typed commands to be appended to the history file (the latter is like specifying INC_APPEND_HISTORY). The history lines are also output with timestamps ala EXTENDED_HISTORY (which makes it easier to find the spot where we left off reading the file after it gets re-written).

  By default, history movement commands visit the imported lines as well as the local lines, but you can toggle this on and off with the set-local-history zle binding. It is also possible to create a zle widget that will make some commands ignore imported commands, and some include them.

  If you find that you want more control over when commands get imported, you may wish to turn SHARE_HISTORY off, INC_APPEND_HISTORY on, and then manually import commands whenever you need them using '**fc -RI**'.

`SH_FILE_EXPANSION` <K> <S>
: 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` <K> <S>
: 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 by default 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 unsetting 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 on not commands are *actually* being read from standard input. The value of this option cannot be changed anywhere other than the command line.

`SH_NULLCMD` <K> <S>
: Do not use the values of `NULLCMD` and `READNULLCMD` when doing redirections, use '`:`' instead (see Chapter 6 [Redirection], page 11).

`SH_OPTION_LETTERS` <K> <S>
: 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` <C> <Z>
: Allow the short forms of `for`, `select`, `if`, and `function` constructs.

`SH_WORD_SPLIT` (`-y`) <K> <S>
: Causes field splitting to be performed on unquoted parameter expansions. Note that this option has nothing to do with word splitting. (See Section 13.3 [Parameter Expansion], page 29.)

`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`) <K>
: Use single-line command line editing instead of multi-line.

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

`UNSET` (`+u`, ksh: `+u`) <K> <S> <Z>
: 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. Set by default in interactive shells connected to a terminal.

15.3 Option Aliases

Some options have alternative names. These aliases are never used for output, but can be used just like normal option names when specifying options to the shell.

`BRACE_EXPAND`
: *NO*`_IGNORE_BRACES` (ksh and bash compatibility)

`DOT_GLOB`
: `GLOB_DOTS` (bash compatibility)

`HASH_ALL`
: `HASH_CMDS` (bash compatibility)

`HIST_APPEND`
: `APPEND_HISTORY` (bash compatibility)

`HIST_EXPAND`
: `BANG_HIST` (bash compatibility)

`LOG`
: *NO*`_HIST_NO_FUNCTIONS` (ksh compatibility)

`MAIL_WARN`
: `MAIL_WARNING` (bash compatibility)

`ONE_CMD`
: `SINGLE_COMMAND` (bash compatibility)

`PHYSICAL`
: `CHASE_LINKS` (ksh and bash compatibility)

`PROMPT_VARS`
: `PROMPT_SUBST` (bash compatibility)

`STDIN`
: `SHIN_STDIN` (ksh compatibility)

`TRACK_ALL`
: `HASH_CMDS` (ksh compatibility)

15.4 Single Letter Options

15.4.1 Default set

`-0`
: CORRECT

`-1`
: PRINT_EXIT_VALUE

`-2`
: *NO*_BAD_PATTERN

`-3`
: *NO*_NOMATCH

`-4`
: GLOB_DOTS

`-5`
: NOTIFY

`-6`
: BG_NICE

`-7`
: IGNORE_EOF

`-8`
: MARK_DIRS

`-9`	AUTO_LIST
`-B`	*NO*_BEEP
`-C`	*NO*_CLOBBER
`-D`	PUSHD_TO_HOME
`-E`	PUSHD_SILENT
`-F`	*NO*_GLOB
`-G`	NULL_GLOB
`-H`	RM_STAR_SILENT
`-I`	IGNORE_BRACES
`-J`	AUTO_CD
`-K`	*NO*_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`	*NO*_PROMPT_CR
`-W`	AUTO_RESUME
`-X`	LIST_TYPES
`-Y`	MENU_COMPLETE
`-Z`	ZLE
`-a`	ALL_EXPORT
`-e`	ERR_EXIT
`-f`	*NO*_RCS
`-g`	HIST_IGNORE_SPACE
`-h`	HIST_IGNORE_DUPS
`-i`	INTERACTIVE
`-k`	INTERACTIVE_COMMENTS
`-l`	LOGIN
`-m`	MONITOR
`-n`	*NO*_EXEC
`-p`	PRIVILEGED

`-r` RESTRICTED
`-s` SHIN_STDIN
`-t` SINGLE_COMMAND
`-u` *NO*_UNSET
`-v` VERBOSE
`-w` CHASE_LINKS
`-x` XTRACE
`-y` SH_WORD_SPLIT

15.4.2 sh/ksh emulation set

`-C` *NO*_CLOBBER
`-X` MARK_DIRS
`-a` ALL_EXPORT
`-b` NOTIFY
`-e` ERR_EXIT
`-f` *NO*_GLOB
`-i` INTERACTIVE
`-l` LOGIN
`-m` MONITOR
`-n` *NO*_EXEC
`-p` PRIVILEGED
`-r` RESTRICTED
`-s` SHIN_STDIN
`-t` SINGLE_COMMAND
`-u` *NO*_UNSET
`-v` VERBOSE
`-x` XTRACE

15.4.3 Also note

`-A` Used by `set` for setting arrays
`-b` Used on the command line to specify end of option processing
`-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
`-s` Used by `set` to sort positional parameters

16 Shell Builtin Commands

- *simple command*
See Section 5.2 [Precommand Modifiers], page 8.

. *file* [*arg* ...]
Read 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 a file named '*file*`.zwc`' is found, is newer than *file*, and is the compiled form (created with the `zcompile` builtin) of *file*, then commands are read from that file instead of *file*.

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 does nothing, although normal argument expansions is performed which may have effects on shell parameters. A zero exit code is returned.

alias [`{+|-}gmrL`] [*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 expansion. 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. Using '`+`' instead of '`-`', or ending the option list with a single '`+`', prevents the values of the aliases from being printed.

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 [`{+|-}UXmt`] [`-wkz`] [*name* ...]
Equivalent to `functions -u`, with the exception of `-X`/`+X`, `-w`, `-k` and `-z`.

The flag `-X` may be used only inside a shell function, and may not be followed by a *name*. It causes the calling function to be marked for autoloading and then immediately loaded and executed, with the current array of positional parameters as arguments. This replaces the previous definition of the function. If no function definition is found, an error is printed and the function remains undefined and marked for autoloading.

The flag `+X` attempts to load each *name* as an autoloaded function, but does *not* execute it. The exit status is zero (success) if the function was not previously defined *and* a definition for it was found. This does *not* replace any existing definition of the function. The exit status is nonzero (failure) if the function was already defined or when no definition was found. In the latter case the function remains undefined and marked for autoloading.

The flag `+X` may be combined with either `-k` or `-z` to make the function be loaded using ksh-style or zsh-style autoloading, respectively. If neither is given, the current

setting of the `KSH_AUTOLOAD` options determines how the function is loaded. With ksh-style autoloading, the contents of the file will not be executed immediately. Instead, the function created will contain the contents of the file plus a call to the function itself appended to it, thus given normal ksh autoloading behaviour on the first call to the function.

With the `-w` flag, the *name*s are taken as names of files compiled with the `zcompile` builtin, and all functions defined in them are marked for autoloading.

`bg` [*job* ...]
job ... `&` Put each specified *job* in the background, or the current job if none is specified.

`bindkey` See Section 17.3 [Zle Builtins], page 96.

`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`.

`cap` See Section 21.2 [The zsh/cap Module], page 188.

`cd` [`-sLP`] [*arg*]
`cd` [`-sLP`] *old new*
`cd` [`-sLP`] `{+|-}`*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. Otherwise, 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 no directory is found and the option `CDABLE_VARS` is set, and a parameter named *arg* exists whose value begins with a slash, treat its value as the directory. In that case, the parameter is added to the named directory hash table.

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.

If the `-s` option is specified, `cd` refuses to change the current directory if the given pathname contains symlinks. If the `-P` option is given or the `CHASE_LINKS` option is set, symbolic links are resolved to their true values. If the `-L` option is given symbolic links are followed regardless of the state of the `CHASE_LINKS` option.

`chdir` Same as `cd`.

`clone` See Section 21.3 [The zsh/clone Module], page 189.

`command` *simple command*
See Section 5.2 [Precommand Modifiers], page 8.

`comparguments`
See Section 21.7 [The zsh/computil Module], page 193.

`compcall` See Section 21.4 [The zsh/compctl Module], page 189.

`compctl` See Section 21.4 [The zsh/compctl Module], page 189.

`compdescribe`
: See Section 21.7 [The zsh/computil Module], page 193.

`compfiles`
: See Section 21.7 [The zsh/computil Module], page 193.

`compgroups`
: See Section 21.7 [The zsh/computil Module], page 193.

`compquote`
: See Section 21.7 [The zsh/computil Module], page 193.

`comptags` See Section 21.7 [The zsh/computil Module], page 193.

`comptry` See Section 21.7 [The zsh/computil Module], page 193.

`compvalues`
: See Section 21.7 [The zsh/computil Module], page 193.

`continue` [*n*]
: 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` 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`] *name* ...
: Temporarily disable the *name*d hash table elements. 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` options 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 prevent them from undergoing filename expansion), 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 *job*s 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, disown 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 character
`\b`	backspace
`\c`	suppress final newline
`\e`	escape

`\f`	form feed
`\n`	linefeed (newline)
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\0`*NNN*	character code in octal
`\x`*NN*	character code in hexadecimal

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

`echotc` See Section 21.16 [The zsh/termcap Module], page 202.

`echoti` See Section 21.17 [The zsh/terminfo Module], page 202.

`emulate` [`-LR`] `{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; more precisely, the tests performed on the argument are the same as those used to determine the emulation at startup based on the shell name, see Section 3.2 [Compatibility], page 5 . If the `-R` option is given, all options are reset to their default value corresponding to the specified emulation mode, except for certain options describing the interactive environment; otherwise, only those options likely to cause portability problems in scripts and functions are altered. If the `-L` option is given, the options `LOCAL_OPTIONS` and `LOCAL_TRAPS` will be set as well, causing the effects of the `emulate` command and any `setopt` and `trap` commands to be local to the immediately surrounding shell function, if any; normally these options are turned off in all emulation modes except `ksh`.

`enable` [`-afmr`] *name* ...
Enable the *name*d hash table elements, 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 enabled 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 in the current shell process.

`exec` *simple command*
See Section 5.2 [Precommand Modifiers], page 8.

`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 *name*s are marked for automatic export to the environment of subsequently executed commands. Equivalent to `typeset -gx`. If a parameter specified does not already exist, it is created in the global scope.

`false` [*arg* ...]
: 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 (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 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, and `-f` prints full time-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 in ISO8601 '*yyyy-mm-dd*' 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. If no filename is specified, the `$HISTFILE` is assumed. If the `-I` option is added to `-R`, only those events that are not already contained within the internal history list are added. If the `-I` option is added to `-A` or `-W`, only those events that are new since last incremental append/write to the history file are appended/written. In any case, the created file will have no more than `$SAVEHIST` entries.

`fg` [*job* ...]
job ...
: Bring each specified *job* in turn to the foreground. If no *job* is specified, resume the current job.

`float` [`{+|-}EFghlrtux`] [*name*[=*value*] ...]
: Equivalent to `typeset -E`, except that options irrelevant to floating point numbers are not permitted.

`functions` [`{+|-}UXmtu`] [*name* ...]
: Equivalent to `typeset -f`.

`getcap`
: See Section 21.2 [The zsh/cap Module], page 188.

`getln` [`-AclneE`] *name* ...
: Read the top value from the buffer stack and put it in the shell parameter `name`. Equivalent to `read -zr`.

`getopts` *optstring name* [*arg* ...]
: Checks the *arg*s for legal options. If the *arg*s are 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`.

The first option to be examined may be changed by explicitly assigning to `OPTIND`. `OPTIND` has an initial value of `1`, and is normally reset to `1` upon exit from a shell function. `OPTARG` is not reset and retains its value from the most recent call to `getopts`. If either of `OPTIND` or `OPTARG` is explicitly unset, it remains unset, and the index or option argument is not stored. The option itself is still stored in *name* in this case.

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

`hash` [`-Ldfmrv`] [*name*[=*value*]] ...

`hash` can be used to directly modify the contents of the command hash table, and the named directory hash table. Normally one would modify these tables by modifying one's `PATH` (for the command hash table) or by creating appropriate shell parameters (for the named directory hash table). The choice of hash table to work on is determined by the `-d` option; without the option the command hash table is used, and with the option the named directory hash table is used.

Given no arguments, and neither the `-r` or `-f` options, the selected hash table will be listed in full.

The `-r` option causes the selected hash table to be emptied. It will be subsequently rebuilt in the normal fashion. The `-f` option causes the selected hash table to be fully rebuilt immediately. For the command hash table this hashes all the absolute directories in the `PATH`, and for the named directory hash table this adds all users' home directories. These two options cannot be used with any arguments.

The `-m` option causes the arguments to be taken as patterns (which should be quoted) and the elements of the hash table matching those patterns are printed. This is the only way to display a limited selection of hash table elements.

For each *name* with a corresponding *value*, put '*name*' in the selected hash table, associating it with the pathname '*value*'. In the command hash table, this means that whenever '*name*' is used as a command argument, the shell will try to execute the file given by '*value*'. In the named directory hash table, this means that '*value*' may be referred to as '`~`*name*'.

For each *name* with no corresponding *value*, attempt to add *name* to the hash table, checking what the appropriate `value` is in the normal manner for that hash table. If an appropriate `value` can't be found, then the hash table will be unchanged.

The `-v` option causes hash table entries to be listed as they are added by explicit specification. If has no effect if used with `-f`.

If the `-L` flag is present, then each hash table entry is printed in the form of a call to hash.

`history` Same as `fc -l`.

`integer` [`{+|-}ghilrtux`] [*name*[=*value*] ...]

Equivalent to `typeset -i`, except that options irrelevant to integers are not permitted.

`jobs` [`-dlprs`] [*job* ...]
`jobs -Z` *string*

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 specified only

running jobs will be listed and if the `-s` flag is given only stopped jobs are shown. If the `-d` flag is given, the directory from which the job was started (which may not be the current directory of the job) will also be shown.

The `-Z` option replaces the shell's argument and environment space with the given string, truncated if necessary to fit. This will normally be visible in `ps` (man page ps(1)) listings. This feature is typically used by daemons, to indicate their state.

`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 10 [Arithmetic Evaluation], page 18 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 can be one of:

`addressspace`
Maximum amount of address space used.

`aiomemorylocked`
Maximum amount of memory locked in RAM for AIO operations.

`aiooperations`
Maximum number of AIO operations.

`cachedthreads`
Maximum number of cached threads.

`coredumpsize`
Maximum size of a core dump.

`cputime` Maximum CPU seconds per process.

`datasize` Maximum data size (including stack) for each process.

`descriptors`
Maximum value for a file descriptor.

`filesize` Largest single file allowed.

`maxproc` Maximum number of processes.

`maxpthreads`
Maximum number of threads per process.

`memorylocked`
: Maximum amount of memory locked in RAM.

`memoryuse`
: Maximum resident set size.

`resident` Maximum resident set size.

`sockbufsize`
: Maximum size of all socket buffers.

`stacksize`
: Maximum stack size for each process.

`vmemorysize`
: Maximum amount of virtual memory.

Which of these resource limits are available depends on the system. *resource* can be abbreviated to any unambiguous prefix.

limit is a number, with an optional scaling factor, as follows:

n`h` hours

n`k` kilobytes (default)

n`m` megabytes or minutes

[*mm*`:`]*ss* minutes and seconds

`local` [`{+|-}AEFLRUZahilrtux` [*n*]] [*name*[`=`*value*]] ...
: Same as `typeset`, except that the options `-g`, and `-f` are not permitted. In this case the `-x` option does not force the use of `-g`, i.e. exported variables will be local to functions.

`log` List all users currently logged in who are affected by the current setting of the `watch` parameter.

`logout` [*n*]
: Same as `exit`, except that it only works in a login shell.

`noglob` *simple command*
: See Section 5.2 [Precommand Modifiers], page 8.

`popd` [`{+|-}`*n*]
: Remove an entry from the directory stack, and perform 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` [`-bnrslzpNDPoOicm`] [`-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 NUL 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.

`-b` Recognize all the escape sequences defined for the `bindkey` command, see Section 17.3 [Zle Builtins], page 96.

`-m` Take the first 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`, sorting is performed case-independently.

`-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.

`-D` Treat the arguments as directory names, replacing prefixes with `~` expressions, as appropriate.

`-P` Perform prompt expansion (see Chapter 12 [Prompt Expansion], page 22).

`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 `$HOME` if the `PUSHD_TO_HOME` option is set or if there is only one entry on the stack. Otherwise, *arg* is interpreted as it would be by `cd`. The meaning of *old* and *new* in the second form is also the same as for `cd`.

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.

If the option `PUSHD_SILENT` is not set, the directory stack will be printed after a `pushd` is performed.

`pushln` [*arg* ...]

Equivalent to `print -nz`.

`pwd` [`-rLP`]

Print the absolute pathname of the current working directory. If the `-r` or the `-P` flag is specified, or the `CHASE_LINKS` option is set and the `-L` flag is not given, the printed path will not contain symbolic links.

`r` Same as `fc -e -`.

`read` [`-rzpqAclneEt`] [`-k` [*num*]] [`-u`*n*] [*name*[`?`*prompt*]] [*name* ...]
Read one line and break it into fields using the characters in `$IFS` as separators, except as noted below. 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*. If *name* is omitted then `REPLY` is used for scalars and `reply` for arrays.

`-r` Raw mode: a '\' at the end of a line does not signify line continuation and backslashes in the line don't quote the following character and are not removed.

`-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`'. Note that this always reads from the terminal, even if used with the `-p` or `-u` or `-z` flags or with redirected input. This option may also be used within zle widgets.

`-k` [*num*] Read only one (or *num*) characters. All are assigned to the first *name*, without word splitting. This flag is ignored when `-q` is present. Input is read from the terminal unless one of `-u` or `-p` is present. This option may also be used within zle widgets.

Note that *num* must be in the argument word that follows `-k`, not in the same word. See `-u`.

`-z` Read one entry from the editor buffer stack and assign it to the first *name*, without word splitting. Text is pushed onto the stack with '`print -z`' or with `push-line` from the line editor (see Chapter 17 [Zsh Line Editor], page 94). This flag is ignored when the `-k` or `-q` flags are present.

`-e`
`-E` The input read is printed (echoed) to the standard output. If the `-e` flag is used, no input is 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 both flags are present, `-l` is used and `-c` is ignored.

`-n` Together with `-c`, the number of the word the cursor is on is read. With `-l`, the index of the character the cursor is on is read. Note that the command name is word number 1, not word 0, and that when the cursor is at the end of the line, its character index is the length of the line plus one.

`-u`*n* Input is read from file descriptor *n*, where *n* is a single digit and must *not* be separated from `-u` by any whitespace.

`-p` Input is read from the coprocess.

`-t` Test if input is available before attempting to read; if none is, return status 1 and do not set any variables. This is not available when reading from the editor buffer with `-z`, when called from within completion with

`-c` or `-l`, with `-q` which clears the input queue before reading, or within zle where other mechanisms should be used to test for input.

Note that read does not attempt to alter the input processing mode. The default mode is canonical input, in which an entire line is read at a time, so usually '`read -t`' will not read anything until an entire line has been typed. However, when reading from the terminal with `-k` this is automatically handled; note that only availability of the first character is tested, so that e.g. '`read -t -k 2`' can still block on the second character.

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 value (exit status) of `read` is 1 when an end-of-file is encountered, or when `-c` or `-l` is present and the command is not called from a `compctl` function, or as described for `-q`. Otherwise the value is 0.

The behavior of some combinations of the `-k`, `-p`, `-q`, `-u` and `-z` flags is undefined. Presently `-q` cancels all the others, `-p` cancels `-u`, `-k` cancels `-z`, and otherwise `-z` cancels both `-p` and `-u`.

The `-c` or `-l` flags cancel any and all of `-kpquz`.

`readonly` Same as `typeset -r`.

`rehash` Same as `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, the return status is that of the last command executed.

If `return` was executed from a trap in a `TRAP`*NAL* function, the effect is different for zero and non-zero 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 numeric value of 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` See Section 21.14 [The zsh/sched Module], page 200.

`set` [`{+|-}`*options* | `{+|-}o` *option_name*] ... [`{+|-}A` [*name*]] [*arg* ...]
Set the options for the shell and/or set the positional parameters, or declare and set 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. For the meaning of the other flags, see Chapter 15 [Options], page 59. 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 *arg*s. if `+A` is used and *name* is an array, the given 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.

`setcap` See Section 21.2 [The zsh/cap Module], page 188.

`setopt` [`{+|-}`*options* | `{+|-}o` *option_name*] [*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.

If the `-m` flag is given the arguments are taken as patterns (which should be quoted to protect them from filename expansion), and all options with names matching these patterns are set.

`shift` [*n*] [*name* ...]
: The positional parameters `${`*n*`+1}` ... are renamed to `$1` ..., where *n* is an arithmetic expression that defaults to 1. If any *name*s are given then the arrays with these names are shifted instead of the positional parameters.

`source` *file* [*arg* ...]
: Same as `.`, except that the current directory is always searched and is always searched first, before directories in `$path`.

`stat`
: See Section 21.15 [The zsh/stat Module], page 200.

`suspend` [`-f`]
: Suspend the execution of the shell (send it a `SIGTSTP`) until it receives a `SIGCONT`. Unless the `-f` option is given, this will refuse to suspend a login shell.

`test` [*arg* ...]
`[` [*arg* ...] `]`
: Like the system version of `test`. Added for compatibility; use conditional expressions instead (see Chapter 11 [Conditional Expressions], page 20).

`times`
: Print the accumulated user and system times for the shell and for processes run from the shell.

`trap` [*arg* [*sig* ...]]
: *arg* is a series of commands (usually quoted to protect it from immediate evaluation by the shell) 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 empty 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.

  Note that traps defined with the `trap` builtin are slightly different from those defined as '`TRAP`*NAL* `() { ... }`', as the latter have their own function environment (line numbers, local variables, etc.) while the former use the environment of the command in which they were called. For example,

  ```
  trap 'print $LINENO' DEBUG
  ```

  will print the line number of a command executed after it has run, while

  ```
  TRAPDEBUG() { print $LINENO; }
  ```

  will always print the number zero.

`true` [*arg* ...]
: Do nothing and return an exit code of 0.

`ttyctl -fu`
: The `-f` option freezes the tty, and `-u` unfreezes it. When the tty is frozen, no changes made to the tty settings by external programs will be honored 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 or is suspended. 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` [`-wfpams`] *name* ...
: Equivalent to `whence -v`.

`typeset` [`{+|-}AEFLRUZafghilrtuxm` [*n*]] [*name*[=*value*] ...]
`typeset` -T [`{+|-}LRUZrux`] *SCALAR*[=*value*] *array*
: Set or display attributes and values for shell parameters.

A parameter is created for each *name* that does not already refer to one. When inside a function, a new parameter is created for every *name* (even those that already exist), and is unset again when the function completes. See Section 14.4 [Local Parameters], page 51. The same rules apply to special shell parameters, which retain their special attributes when made local.

For each *name*=*value* assignment, the parameter *name* is set to *value*. Note that arrays currently cannot be assigned in `typeset` expressions, only scalars and integers.

For each remaining *name* that refers to a parameter that is set, the name and value of the parameter are printed in the form of an assignment. Nothing is printed for newly-created parameters, or when any attribute flags listed below are given along with the *name*. Using '`+`' instead of minus to introduce an attribute turns it off.

If the `-T` option is given, exactly two (or zero) *name* arguments must be present. They represent a scalar and an array (in that order) that will be tied together in the manner of `$PATH` and `$path`. In other words, an array present in the latter variable appears as a scalar with the elements of the array joined by colons in the former. Only the scalar may have an initial value. Both the scalar and the array may otherwise be manipulated as normal. If one is unset, the other will automatically be unset too. There is no way of untying the variables without unsetting them, or converting the type of one of them with another `typeset` command; `+T` does not work, assigning an array to *SCALAR* is an error, and assigning a scalar to *array* sets it to be a single-element array. Note that both '`typeset -xT ...`' and '`export -T ...`' work, but only the scalar will be marked for export.

The `-g` (global) flag is treated specially: it means that any resulting parameter will not be restricted to local scope. Note that this does not necessarily mean that the parameter will be global, as the flag will apply to any existing parameter (even if unset) from an enclosing function. This flag does not affect the parameter after creation, hence it has no effect when listing existing parameters, nor does the flag `+g` have any effect except in combination with `-m` (see below).

If no *name* is present, the names and values of all parameters are printed. In this case the attribute flags restrict the display to only those parameters that have the specified attributes, and using '`+`' rather than '`-`' to introduce the flag suppresses printing of the values of parameters when there is no parameter name. Also, if the last option is the word '`+`', then names are printed but values are not.

If the `-m` flag is given the *name* arguments are taken as patterns (which should be quoted). With no attribute flags, all parameters (or functions with the `-f` flag) with matching names are printed. Note that `-m` is ignored if no patterns are given. If the `+g` flag is combined with `-m`, a new local parameter is created for every matching parameter that is not already local. Otherwise `-m` applies all other flags or assignments to the existing parameters. Except when assignments are made with *name*=*value*, using `+m` forces the matching parameters to be printed, even inside a function.

If no attribute flags are given and either no `-m` flag is present or the `+m` form was used, each parameter name printed is preceded by a list of the attributes of that

parameter (`array`, `association`, `exported`, `integer`, `readonly`). If `+m` is used with attribute flags, and all those flags are introduced with `+`, the matching parameter names are printed but their values are not.

The following attribute flags may be specified:

`-A` The names refer to associative array parameters; see Section 14.2 [Array Parameters], page 46.

`-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 expanded, 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 if defines the width of the field; otherwise it is determined by the width of the value of the first assignment. When the parameter is expanded, the field is left filled with blanks or truncated from the end.

`-U` For arrays (but not for associative arrays), keep only the first occurrence of each duplicated value. This may also be set for colon-separated special parameters like `PATH` or `FIGNORE`, etc. This flag has a different meaning when used with `-f`; see below.

`-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.

`-a` The names refer to array parameters. An array parameter may be created this way, but it may not be assigned to in the `typeset` statement. When displaying, both normal and associative arrays are shown.

`-f` The names refer to functions rather than parameters. No assignments can be made, and the only other valid flags are `-t`, `-u` and `-U`. The flag `-t` turns on execution tracing for this function. The `-u` and `-U` flags cause the function to be marked for autoloading; `-U` also causes alias expansion to be suppressed when the function is loaded. The `fpath` parameter will be searched to find the function definition when the function is first referenced; see Chapter 8 [Functions], page 14.

`-h` Hide: only useful for special parameters (those marked '<S>' in the table in Section 14.5 [Parameters Set By The Shell], page 51), and for local parameters with the same name as a special parameter, though harmless for others. A special parameter with this attribute will not retain its special effect when made local. Thus after '`typeset -h PATH`', a function containing '`typeset PATH`' will create an ordinary local parameter without the usual behaviour of `PATH`. Alternatively, the local parameter may itself be given this attribute; hence inside a function '`typeset -h PATH`' creates an ordinary local parameter and the special `PATH` parameter is not altered in any way. It is also possible to create a local parameter using '`typeset +h` *special*', where the local copy of *special* will retain its special properties regardless of having the `-h` attribute. Global special parameters loaded from shell modules (currently those in `zsh/mapfile` and `zsh/parameter`) are automatically given the `-h` attribute to avoid name clashes.

`-H` Hide value: specifies that `typeset` will not display the value of the parameter when listing parameters; the display for such parameters is

always as if the '+' flag had been given. Use of the parameter is in other respects normal, and the option does not apply if the parameter is specified by name, or by pattern with the `-m` option. This is on by default for the parameters in the `zsh/parameter` and `zsh/mapfile` modules. Note, however, that unlike the `-h` flag this is also useful for non-special parameters.

`-i` Use an internal integer representation. If *n* is nonzero it defines the output arithmetic base, otherwise it is determined by the first assignment.

`-E` Use an internal double-precision floating point representation. On output the variable will be converted to scientific notation. If *n* is nonzero it defines the number of significant figures to display; the default is ten.

`-F` Use an internal double-precision floating point representation. On output the variable will be converted to fixed-point decimal notation. If *n* is nonzero it defines the number of digits to display after the decimal point; the default is ten.

`-l` Convert the result to lower case whenever the parameter is expanded. The value is *not* converted when assigned.

`-r` The given *name*s are marked readonly.

`-t` Tags the named parameters. Tags have no special meaning to the shell. This flag has a different meaning when used with `-f`; see above.

`-u` Convert the result to upper case whenever the parameter is expanded. The value is *not* converted when assigned. This flag has a different meaning when used with `-f`; see above.

`-x` Mark for automatic export to the environment of subsequently executed commands. If the option `GLOBAL_EXPORT` is set, this implies the option `-g`, unless `+g` is also explicitly given; in other words the parameter is not made local to the enclosing function. This is for compatibility with previous versions of zsh.

`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` 512-byte blocks on the size of core dumps.

`-d` K-bytes on the size of the data segment.

`-f` 512-byte blocks on the size of files written.

`-l` K-bytes on the size of locked-in memory.

`-m` K-bytes on the size of physical memory.

`-n` open file descriptors.

`-s` K-bytes on the size of the stack.

`-t` CPU seconds to be used.

`-u` processes available to the user.

`-v` K-bytes on the size of virtual memory.

`umask` [`-S`] [*mask*]
: The umask is set to *mask*. *mask* can be either an octal number or a symbolic value as described in man page 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` Same as `unhash -a`.

`unfunction`
: Same as `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 has appropriate privileges, 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. Local parameters remain local even if unset; they appear unset within scope, but the previous value will still reappear when the scope ends.

  Individual elements of associative array parameters may be unset by using subscript syntax on *name*, which should be quoted (or the entire command prefixed with `noglob`) to protect the subscript from filename generation.

  If the `-m` flag is specified the arguments are taken as patterns (should be quoted) and all parameters with matching names are unset. Note that this cannot be used when unsetting associative array elements, as the subscript will be treated as part of the pattern.

  `unset -f` is equivalent to `unfunction`.

`unsetopt` [`{+|-}`*options* | `{+|-}o` *option_name*] [*name* ...]
: Unset the options for the shell. All options specified either with flags or by name are unset. If no arguments are supplied, the names of all options currently unset are printed. 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 unset.

`vared` See Section 17.3 [Zle Builtins], page 96.

`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` [`-vcwfpams`] *name* ...
: For each name, indicate how it would be interpreted if used as a command name.

`-v` Produce a more verbose report.

`-c` Print the results in a *csh*-like format. This takes precedence over `-v`.

`-w` For each *name*, print '*name*`:` *word*' where *word* is one of `alias`, `builtin`, `command`, `function`, `hashed`, `reserved` or `none`, according as *name* corresponds to an alias, a built-in command, an external command, a shell function, a command defined with the `hash` builtin, a reserved word, or is not recognised. This takes precedence over `-v` and `-c`.

`-f` Causes the contents of a shell function to be displayed, which would otherwise not happen unless the `-c` flag were used.

`-p` Do a path search for *name* even if it is an alias, reserved word, shell function or builtin.

`-a` Do a search for all occurrences of *name* throughout the command path. Normally only the first occurrence is printed.

`-m` The arguments are taken as patterns (should be quoted), and the information is displayed for each command matching one of these patterns.

`-s` If a pathname contains symlinks, print the symlink-free pathname as well.

`where` [`-wpms`] *name* ...
Equivalent to `whence -ca`.

`which` [`-wpams`] *name* ...
Equivalent to `whence -c`.

`zcompile` [`-U`] [`-z` | `-k`] [`-R` | `-M`] *file* [*name* ...]
`zcompile -ca` [`-m`] [`-R` | `-M`] *file* [*name* ...]
`zcompile -t` *file* [*name* ...]
This builtin command can be used to compile functions or scripts, storing the compiled form in a file, and to examine files containing the compiled form. This allows faster autoloading of functions and execution of scripts by avoiding parsing of the text when the files are read.

The first form (without the `-c`, `-a` or `-t` options) creates a compiled file. If only the *file* argument is given, the output file has the name '*file*`.zwc`' and will be placed in the same directory as the *file*. The shell will load the compiled file instead of the normal function file when the function is autoloaded; see Chapter 8 [Functions], page 14 for a description of how autoloaded functions are searched. The extension `.zwc` stands for 'zsh word code'.

If there is at least one *name* argument, all the named files are compiled into the output *file* given as the first argument. If *file* does not end in `.zwc`, this extension is automatically appended. Files containing multiple compiled functions are called 'digest' files, and are intended to be used as elements of the `FPATH`/`fpath` special array.

The second form, with the `-c` or `-a` options, writes the compiled definitions for all the named functions into *file*. For `-c`, the names must be functions currently defined in the shell, not those marked for autoloading. Undefined functions that are marked for autoloading may be written by using the `-a` option, in which case the `fpath` is searched and the contents of the definition files for those functions, if found, are compiled into *file*. If both `-c` and `-a` are given, names of both defined functions and functions marked for autoloading may be given. In either case, the functions in files

written with the `-c` or `-a` option will be autoloaded as if the `KSH_AUTOLOAD` option were unset.

The reason for handling loaded and not-yet-loaded functions with different options is that some definition files for autoloading define multiple functions, including the function with the same name as the file, and, at the end, call that function. In such cases the output of '`zcompile -c`' does not include the additional functions defined in the file, and any other initialization code in the file is lost. Using '`zcompile -a`' captures all this extra information.

If the `-m` option is combined with `-c` or `-a`, the *name*s are used as patterns and all functions whose names match one of these patterns will be written. If no *name* is given, the definitions of all functions currently defined or marked as autoloaded will be written.

The third form, with the `-t` option, examines an existing compiled file. Without further arguments, the names of the original files compiled into it are listed. The first line of output shows the version of the shell which compiled the file and how the file will be used (i.e. by reading it directly or by mapping it into memory). With arguments, nothing is output and the return value is set to zero if definitions for *all* *name*s were found in the compiled file, and non-zero if the definition for at least one *name* was not found.

Other options:

`-U` Aliases are not expanded when compiling the *name*d files.

`-R` When the compiled file is read, its contents are copied into the shell's memory, rather than memory-mapped (see `-M`). This happens automatically on systems that do not support memory mapping.

When compiling scripts instead of autoloadable functions, it is often desirable to use this option; otherwise the whole file, including the code to define functions which have already been defined, will remain mapped, consequently wasting memory.

`-M` The compiled file is mapped into the shell's memory when read. This is done in such a way that multiple instances of the shell running on the same host will share this mapped file. If neither `-R` nor `-M` is given, the `zcompile` builtin decides what to do based on the size of the compiled file.

`-k`
`-z` These options are used when the compiled file contains functions which are to be autoloaded. If `-z` is given, the function will be autoloaded as if the `KSH_AUTOLOAD` option is *not* set, even if it is set at the time the compiled file is read, while if the `-k` is given, the function will be loaded as if `KSH_AUTOLOAD` *is* set. If neither of these options is given, the function will be loaded as determined by the setting of the `KSH_AUTOLOAD` option at the time the compiled file is read.

These options may also appear as many times as necessary between the listed *name*s to specify the loading style of all following functions, up to the next `-k` or `-z`.

The created file always contains two versions of the compiled format, one for big-endian machines and one for small-endian machines. The upshot of this is that the compiled file is machine independent and if it is read or mapped, only one half of the file is actually used (and mapped).

`zformat` See Section 21.23 [The zsh/zutil Module], page 210.

`zftp` See Section 21.18 [The zsh/zftp Module], page 202.

`zle` See Section 17.3 [Zle Builtins], page 96.

`zmodload` [`-dL`] [...]
`zmodload -e` [`-A`] [...]
`zmodload` [`-a` [`-bcpf` [`-I`]]] [`-iL`] ...
`zmodload -u` [`-abcdpf` [`-I`]] [`-iL`] ...
`zmodload -A` [`-L`] [*modalias*[=*module*] ...]
`zmodload -R` *modalias* ...

Performs operations relating to zsh's loadable modules. Loading of modules while the shell is running ('dynamical loading') is not available on all operating systems, or on all installations on a particular operating system, although the `zmodload` command itself is always available and can be used to manipulate modules built into versions of the shell executable without dynamical loading.

Without arguments the names of all currently loaded binary modules are printed. The `-L` option causes this list to be in the form of a series of `zmodload` commands. Forms with arguments are:

`zmodload` [`-i`] *name* ...
`zmodload -u` [`-i`] *name* ...

In the simplest case, `zmodload` loads a binary module. The module must be in a file with a name consisting of the specified *name* followed by a standard suffix, usually '`.so`' ('`.sl`' on HPUX). If the module to be loaded is already loaded and the `-i` option is given, the duplicate module is ignored. Otherwise `zmodload` prints an error message.

The *name*d module is searched for in the same way a command is, using `$module_path` instead of `$path`. However, the path search is performed even when the module name contains a '`/`', which it usually does. There is no way to prevent the path search.

With `-u`, `zmodload` unloads modules. The same *name* must be given that was given when the module was loaded, but it is not necessary for the module to exist in the filesystem. The `-i` option suppresses the error if the module is already unloaded (or was never loaded).

Each module has a boot and a cleanup function. The module will not be loaded if its boot function fails. Similarly a module can only be unloaded if its cleanup function runs successfully.

`zmodload -d` [`-L`] [*name*]
`zmodload -d` *name dep* ...
`zmodload -ud` *name* [*dep* ...]

The `-d` option can be used to specify module dependencies. The modules named in the second and subsequent arguments will be loaded before the module named in the first argument.

With `-d` and one argument, all dependencies for that module are listed. With `-d` and no arguments, all module dependencies are listed. This listing is by default in a Makefile-like format. The `-L` option changes this format to a list of `zmodload -d` commands.

If `-d` and `-u` are both used, dependencies are removed. If only one argument is given, all dependencies for that module are removed.

`zmodload -ab` [`-L`]
`zmodload -ab` [`-i`] *name* [*builtin* ...]
`zmodload -ub` [`-i`] *builtin* ...

The `-ab` option defines autoloaded builtins. It defines the specified *builtin*s. When any of those builtins is called, the module specified in

the first argument is loaded. If only the *name* is given, one builtin is defined, with the same name as the module. `-i` suppresses the error if the builtin is already defined or autoloaded, regardless of which module it came from.

With `-ab` and no arguments, all autoloaded builtins are listed, with the module name (if different) shown in parentheses after the builtin name. The `-L` option changes this format to a list of `zmodload -a` commands.

If `-b` is used together with the `-u` option, it removes builtins previously defined with `-ab`. This is only possible if the builtin is not yet loaded. `-i` suppresses the error if the builtin is already removed (or never existed).

`zmodload -ac` [`-IL`]
`zmodload -ac` [`-iI`] *name* [*cond* ...]
`zmodload -uc` [`-iI`] *cond* ...

The `-ac` option is used to define autoloaded condition codes. The *cond* strings give the names of the conditions defined by the module. The optional `-I` option is used to define infix condition names. Without this option prefix condition names are defined.

If given no condition names, all defined names are listed (as a series of `zmodload` commands if the `-L` option is given).

The `-uc` option removes definitions for autoloaded conditions.

`zmodload -ap` [`-L`]
`zmodload -ap` [`-i`] *name* [*parameter* ...]
`zmodload -up` [`-i`] *parameter* ...

The `-p` option is like the `-b` and `-c` options, but makes `zmodload` work on autoloaded parameters instead.

`zmodload -af` [`-L`]
`zmodload -af` [`-i`] *name* [*function* ...]
`zmodload -uf` [`-i`] *function* ...

The `-f` option is like the `-b`, `-p`, and `-c` options, but makes `zmodload` work on autoloaded math functions instead.

`zmodload -a` [`-L`]
`zmodload -a` [`-i`] *name* [*builtin* ...]
`zmodload -ua` [`-i`] *builtin* ...

Equivalent to `-ab` and `-ub`.

`zmodload -e` [`-A`] [*string* ...]

The `-e` option without arguments lists all loaded modules; if the `-A` option is also given, module aliases corresponding to loaded modules are also shown. With arguments only the return status is set to zero if all *string*s given as arguments are names of loaded modules and to one if at least on *string* is not the name of a loaded module. This can be used to test for the availability of things implemented by modules. In this case, any aliases are automatically resolved and the `-A` flag is not used.

`zmodload -A` [`-L`] [*modalias*[=*module*] ...]

For each argument, if both *modalias* and *module* are given, define *modalias* to be an alias for the module *module*. If the module *modalias* is ever subsequently requested, either via a call to `zmodload` or implicitly, the shell will attempt to load *module* instead. If *module* is not given, show the definition of *modalias*. If no arguments are given, list all defined module aliases. When listing, if the `-L` flag was also given, list the definition as a `zmodload` command to recreate the alias.

The existence of aliases for modules is completely independent of whether the name resolved is actually loaded as a module: while the alias exists, loading and unloading the module under any alias has exactly the same effect as using the resolved name, and does not affect the connection between the alias and the resolved name which can be removed either by `zmodload -R` or by redefining the alias. Chains of aliases (i.e. where the first resolved name is itself an alias) are valid so long as these are not circular. As the aliases take the same format as module names, they may include path separators: in this case, there is no requirement for any part of the path named to exist as the alias will be resolved first. For example, '`any/old/alias`' is always a valid alias.

Dependencies added to aliased modules are actually added to the resolved module; these remain if the alias is removed. It is valid to create an alias whose name is one of the standard shell modules and which resolves to a different module. However, if a module has dependencies, it will not be possible to use the module name as an alias as the module will already be marked as a loadable module in its own right.

Apart from the above, aliases can be used in the `zmodload` command anywhere module names are required. However, aliases will not be shown in lists of loaded modules with a bare '`zmodload`'.

`zmodload -R` *modalias ...*
: For each *modalias* argument that was previously defined as a module alias via `zmodload -A`, delete the alias. If any was not defined, an error is caused and the remainder of the line is ignored.

Note that `zsh` makes no distinction between modules that were linked into the shell and modules that are loaded dynamically. In both cases this builtin command has to be used to make available the builtins and other things defined by modules (unless the module is autoloaded on these definitions). This is true even for systems that don't support dynamic loading of modules.

`zparseopts`
: See Section 21.23 [The zsh/zutil Module], page 210.

`zprof` See Section 21.21 [The zsh/zprof Module], page 209.

`zpty` See Section 21.22 [The zsh/zpty Module], page 209.

`zregexparse`
: See Section 21.23 [The zsh/zutil Module], page 210.

`zstyle` See Section 21.23 [The zsh/zutil Module], page 210.

17 Zsh Line Editor

17.1 Description

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

There are two display modes. The first, multiline 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 default.

The parameters `BAUD`, `COLUMNS`, and `LINES` are also used by the line editor. Section 14.6 [Parameters Used By The Shell], page 53.

17.2 Keymaps

A keymap in ZLE contains a set of bindings between key sequences and ZLE commands. The empty key sequence cannot be bound.

There can be any number of keymaps at any time, and each keymap has one or more names. If all of a keymap's names are deleted, it disappears. `bindkey` can be used to manipulate keymap names.

Initially, there are four keymaps:

`emacs` EMACS emulation

`viins` vi emulation - insert mode

`vicmd` vi emulation - command mode

`.safe` fallback keymap

The '`.safe`' keymap is special. It can never be altered, and the name can never be removed. However, it can be linked to other names, which can be removed. In the future other special keymaps may be added; users should avoid using names beginning with '`.`' for their own keymaps.

In addition to these four names, either '`emacs`' or '`viins`' is also linked to the name '`main`'. If one of the `VISUAL` or `EDITOR` environment variables contain the string '`vi`' when the shell starts up then it will be '`viins`', otherwise it will be '`emacs`'. `bindkey`'s `-e` and `-v` options provide a convenient way to override this default choice.

When the editor starts up, it will select the '`main`' keymap. If that keymap doesn't exist, it will use '`.safe`' instead.

In the '`.safe`' keymap, each single key is bound to `self-insert`, except for `^J` (line feed) and `^M` (return) which are bound to `accept-line`. This is deliberately not pleasant to use; if you are using it, it means you deleted the main keymap, and you should put it back.

17.2.1 Reading Commands

When ZLE is reading a command from the terminal, it may read a sequence that is bound to some command and is also a prefix of a longer bound string. In this case ZLE will wait a certain time to see if more characters are typed, and if not (or they don't match any longer string) it will execute the binding. This timeout is defined by the `KEYTIMEOUT` parameter; its default is 0.4 sec. There is no timeout if the prefix string is not itself bound to a command.

As well as ZLE commands, key sequences can be bound to other strings, by using '`bindkey -s`'. When such a sequence is read, the replacement string is pushed back as input, and the command reading process starts again using these fake keystrokes. This input can itself invoke further replacement strings, but in order to detect loops the process will be stopped if there are twenty such replacements without a real command being read.

17.3 Zle Builtins

The ZLE module contains three related builtin commands. The `bindkey` command manipulates keymaps and key bindings; the `vared` command invokes ZLE on the value of a shell parameter; and the `zle` command manipulates editing widgets and allows command line access to ZLE commands from within shell functions.

`bindkey` [*options*] `-l`
`bindkey` [*options*] `-d`
`bindkey` [*options*] `-D` *keymap ...*
`bindkey` [*options*] `-A` *old-keymap new-keymap*
`bindkey` [*options*] `-N` *new-keymap* [*old-keymap*]
`bindkey` [*options*] `-m`
`bindkey` [*options*] `-r` *in-string ...*
`bindkey` [*options*] `-s` *in-string out-string ...*
`bindkey` [*options*] *in-string command ...*
`bindkey` [*options*] [*in-string*]

`bindkey`'s options can be divided into three categories: keymap selection, operation selection, and others. The keymap selection options are:

`-e` Selects keymap '`emacs`', and also links it to '`main`'.

`-v` Selects keymap '`viins`', and also links it to '`main`'.

`-a` Selects keymap '`vicmd`'.

`-M` The first non-option argument is used as a keymap name, and does not otherwise count as an argument.

If a keymap selection is required and none of the options above are used, the '`main`' keymap is used. Some operations do not permit a keymap to be selected, namely:

`-l` List all existing keymap names. If the `-L` option is also used, list in the form of `bindkey` commands to create the keymaps.

`-d` Delete all existing keymaps and reset to the default state.

`-D` *keymap ...*
Delete the named *keymap*s.

`-A` *old-keymap new-keymap*
Make the *new-keymap* name an alias for *old-keymap*, so that both names refer to the same keymap. The names have equal standing; if either is deleted, the other remains. If there is already a keymap with the *new-keymap* name, it is deleted.

`-N` *new-keymap* [*old-keymap*]
Create a new keymap, named *new-keymap*. If a keymap already has that name, it is deleted. If an *old-keymap* name is given, the new keymap is initialized to be a duplicate of it, otherwise the new keymap will be empty.

To use a newly created keymap, it should be linked to `main`. Hence the sequence of commands to create and use a new keymap '`mymap`' initialized from the `emacs` keymap (which remains unchanged) is:

```
bindkey -N mymap emacs
bindkey -A mymap main
```

Note that while '`bindkey -A` *newmap* `main`' will work when *newmap* is `emacs` or `viins`, it will not work for `vicmd`, as switching from vi insert to command mode becomes impossible.

The following operations act on the '`main`' keymap if no keymap selection option was given:

`-m` Add the built-in set of meta-key bindings to the selected keymap. Only keys that are unbound or bound to `self-insert` are affected.

`-r` *in-string ...*
Unbind the specified *in-string*s in the selected keymap. This is exactly equivalent to binding the strings to `undefined-key`.

When `-R` is also used, interpret the *in-string*s as ranges.

When `-p` is also used, the *in-string*s specify prefixes. Any binding that has the given *in-string* as a prefix, not including the binding for the *in-string* itself, if any, will be removed. For example,

```
bindkey -rpM viins '^['
```

will remove all bindings in the vi-insert keymap beginning with an escape character (probably cursor keys), but leave the binding for the escape character itself (probably `vi-cmd-mode`). This is incompatible with the option `-R`.

`-s` *in-string out-string ...*
Bind each *in-string* to each *out-string*. When *in-string* is typed, *out-string* will be pushed back and treated as input to the line editor. When `-R` is also used, interpret the *in-string*s as ranges.

in-string command ...
Bind each *in-string* to each *command*. When `-R` is used, interpret the *in-string*s as ranges.

[*in-string*]
List key bindings. If an *in-string* is specified, the binding of that string in the selected keymap is displayed. Otherwise, all key bindings in the selected keymap are displayed. (As a special case, if the `-e` or `-v` option is used alone, the keymap is *not* displayed - the implicit linking of keymaps is the only thing that happens.)

When the option `-p` is used, the *in-string* must be present. The listing shows all bindings which have the given key sequence as a prefix, not including any bindings for the key sequence itself.

When the `-L` option is used, the list is in the form of `bindkey` commands to create the key bindings.

When the `-R` option is used as noted above, a valid range consists of two characters, with an optional '`-`' between them. All characters between the two specified, inclusive, are bound as specified.

For either *in-string* or *out-string*, the following escape sequences are recognised:

`\a`	bell character
`\b`	backspace
`\e`, `\E`	escape
`\f`	form feed
`\n`	linefeed (newline)
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab

`\`*NNN* character code in octal

`\x`*NN* character code in hexadecimal

`\M[-]`*X* character with meta bit set

`\C[-]`*X* control character

`^`*X* control character

In all other cases, '`\`' escapes the following character. Delete is written as '`^?`'. Note that '`\M^?`' and '`^\M?`' are not the same, and that (unlike emacs), the bindings '`\M-`*X*' and '`\e`*X*' are entirely distinct, although they are initialized to the same bindings by '`bindkey -m`'.

`vared` [`-Aache`] [`-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. When the `-c` flag is given, the parameter is created if it doesn't already exist. The `-a` flag may be given with `-c` to create an array parameter, or the `-A` flag to create an associative array. If the type of an existing parameter does not match the type to be created, the parameter is unset and recreated.

If an array or array slice is being edited, separator characters as defined in `$IFS` will be shown quoted with a backslash, as will backslashes themselves. Conversely, when the edited text is split into an array, a backslash quotes an immediately following separator character or backslash; no other special handling of backslashes, or any handling of quotes, is performed.

Individual elements of existing array or associative array parameters may be edited by using subscript syntax on *name*. New elements are created automatically, even without `-c`.

If the `-p` flag is given, the following string will be taken as the prompt to display at the left. 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. If the `-e` flag is given, typing `^D` (Control-D) on an empty line causes `vared` to exit immediately with a non-zero return value.

`zle -l` [`-L | -a`] [*string* ...]
`zle -D` *widget* ...
`zle -A` *old-widget new-widget*
`zle -N` *widget* [*function*]
`zle -C` *widget completion-widget function*
`zle -R` [`-c`] [*display-string*] [*string* ...]
`zle -M` *string*
`zle -U` *string*
`zle -I`
`zle` *widget* `[ -n` *num* `] [ -N ]` *args* ...
`zle`

The `zle` builtin performs a number of different actions concerning ZLE. Which operation it performs depends on its options:

`-l` [`-L | -a`]

List all existing user-defined widgets. If the `-L` option is used, list in the form of `zle` commands to create the widgets.

When combined with the `-a` option, all widget names are listed, including the builtin ones. In this case the `-L` option is ignored.

If at least one *string* is given, nothing will be printed but the return status will be zero if all *string*s are names of existing widgets (or of user-defined widgets if the `-a` flag is not given) and non-zero if at least one *string* is not a name of an defined widget.

`-D` *widget ...*
: Delete the named *widget*s.

`-A` *old-widget new-widget*
: Make the *new-widget* name an alias for *old-widget*, so that both names refer to the same widget. The names have equal standing; if either is deleted, the other remains. If there is already a widget with the *new-widget* name, it is deleted.

`-N` *widget* [*function*]
: Create a user-defined widget. If there is already a widget with the specified name, it is overwritten. When the new widget is invoked from within the editor, the specified shell *function* is called. If no function name is specified, it defaults to the same name as the widget. For further information, see the section *Widgets* in Chapter 17 [Zsh Line Editor], page 94.

`-C` *widget completion-widget function*
: Create a user-defined completion widget named *widget*. The completion widget will behave like the built-in completion-widget whose name is given as *completion-widget*. To generate the completions, the shell function *function* will be called. For further information, see Chapter 18 [Completion Widgets], page 113.

`-R` [`-c`] [*display-string*] [*string ...*]
: Redisplay the command line; this is to be called from within a user-defined widget to allow changes to become visible. If a *display-string* is given and not empty, this is shown in the status line (immediately below the line being edited).

  If the optional *string*s are given they are listed below the prompt in the same way as completion lists are printed. If no *string*s are given but the `-c` option is used such a list is cleared.

  Note that this option is only useful for widgets that do not exit immediately after using it because the strings displayed will be erased immediately after return from the widget.

  This command can safely be called outside user defined widgets; if zle is active, the display will be refreshed, while if zle is not active, the command has no effect. In this case there will usually be no other arguments. The status is zero if zle was active, else one.

`-M` *string*
: As with the `-R` option, the *string* will be displayed below the command line; unlike the `-R` option, the string will not be put into the status line but will instead be printed normally below the prompt. This means that the *string* will still be displayed after the widget returns (until it is overwritten by subsequent commands).

`-U` *string*
: This pushes the characters in the *string* onto the input stack of ZLE. After the widget currently executed finishes ZLE will behave as if the characters in the *string* were typed by the user.

  As ZLE uses a stack, if this option is used repeatedly the last string pushed onto the stack will be processed first. However, the characters in each *string* will be processed in the order in which they appear in the string.

`-I`
: Unusually, this option is only useful *outside* ordinary widget functions. It invalidates the current zle display in preparation for output; usually this will be from a trap function. It has no effect if zle is not active.

When a trap exits, the shell checks to see if the display needs restoring, hence the following will print output in such a way as not to disturb the line being edited:

```
TRAPUSR1() {
    # Invalidate zle display
  zle -I
    # Show output
  print Hello
}
```

Note that there are better ways of manipulating the display from within zle widgets. In general, the trap function may need to test whether zle is loaded before using this method; if it is not, there is no point in loading it specially since the line editor will not be active.

The status is zero if zle was active, else one.

widget `[ -n` *num* `] [ -N ]` *args* ...
Invoke the specified widget. This can only be done when ZLE is active; normally this will be within a user-defined widget.

With the options `-n` and `-N`, the current numerical argument will be saved and then restored after the call to `widget`; '`-n` *num*' sets the numerical argument temporarily to *num*, while '`-N`' sets it to the default, i.e. as if there were none.

Any further arguments will be passed to the widget. If it is a shell function, these are passed down as positional parameters; for builtin widgets it is up to the widget in question what it does with them. Currently arguments are only handled by the incremental-search commands, the `history-search-forward` and `-backward` and the corresponding functions prefixed by `vi-`, and by `universal-argument`. No error is flagged if the command does not use the arguments, or only uses some of them.

The return status reflects the success or failure of the operation carried out by the widget, or if it is a user-defined widget the return status of the shell function.

A non-zero return status causes the shell to beep when the widget exits, unless the `BEEP` options was unset or the widget was called via the `zle` command. Thus if a user defined widget requires an immediate beep, it should call the `beep` widget directly.

With no options and no arguments, only the return status will be set. It is zero if ZLE is currently active and widgets could be invoked using this builtin command and non-zero if ZLE is not active.

17.4 Widgets

All actions in the editor are performed by 'widgets'. A widget's job is simply to perform some small action. The ZLE commands that key sequences in keymaps are bound to are in fact widgets. Widgets can be user-defined or built in.

The standard widgets built in to ZLE are listed in Standard Widgets below. Other built-in widgets can be defined by other modules (see Chapter 21 [Zsh Modules], page 187). Each built-in widget has two names: its normal canonical name, and the same name preceded by a '`.`'. The '`.`' name is special: it can't be rebound to a different widget. This makes the widget available even when its usual name has been redefined.

User-defined widgets are defined using '`zle -N`', and implemented as shell functions. When the widget is executed, the corresponding shell function is executed, and can perform editing (or other) actions. It is recommended that user-defined widgets should not have names starting with '`.`'.

17.5 User-Defined Widgets

User-defined widgets, being implemented as shell functions, can execute any normal shell command. They can also run other widgets (whether built-in or user-defined) using the `zle` builtin command. The standard input of the function is closed to prevent external commands from unintentionally blocking ZLE by reading from the terminal, but `read -k` or `read -q` can be used to read characters. Finally, they can examine and edit the ZLE buffer being edited by reading and setting the special parameters described below.

These special parameters are always available in widget functions, but are not in any way special outside ZLE. If they have some normal value outside ZLE, that value is temporarily inaccessible, but will return when the widget function exits. These special parameters in fact have local scope, like parameters created in a function using `local`.

Inside completion widgets and traps called while ZLE is active, these parameters are available read-only.

`BUFFER` (scalar)
: The entire contents of the edit buffer. If it is written to, the cursor remains at the same offset, unless that would put it outside the buffer.

`BUFFERLINES`
: The number of screen lines needed for the edit buffer currently displayed on screen (i.e. without any changes to the preceding parameters done after the last redisplay).

`CURSOR` (integer)
: The offset of the cursor, within the edit buffer. This is in the range 0 to `$#BUFFER`, and is by definition equal to `$#LBUFFER`. Attempts to move the cursor outside the buffer will result in the cursor being moved to the appropriate end of the buffer.

`HISTNO` (integer)
: The current history number.

`KEYS` (scalar)
: The keys typed to invoke this widget, as a literal string.

`LASTWIDGET` (scalar)
: The name of the last widget that was executed.

`LBUFFER` (scalar)
: The part of the buffer that lies to the left of the cursor position. If it is assigned to, only that part of the buffer is replaced, and the cursor remains between the new `$LBUFFER` and the old `$RBUFFER`.

`MARK` (integer)
: Like `CURSOR`, but for the mark.

`NUMERIC` (integer)
: The numeric argument. If no numeric argument was given, this parameter is unset. When this is set inside a widget function, builtin widgets called with the `zle` builtin command will use the value assigned. If it is unset inside a widget function, builtin widgets called behave as if no numeric argument was given.

`PENDING` (integer)
: The number of bytes pending for input, i.e. the number of bytes which have already been typed and can immediately be read. On systems where the shell is not able to get this information, this parameter will always have a value of zero.

`PREBUFFER` (scalar)
: In a multi-line input at the secondary prompt, this read-only parameter contains the contents of the lines before the one the cursor is currently in.

`RBUFFER` (scalar)
: The part of the buffer that lies to the right of the cursor position. If it is assigned to, only that part of the buffer is replaced, and the cursor remains between the old `$LBUFFER` and the new `$RBUFFER`.

`WIDGET` (scalar)
: The name of the widget currently being executed.

17.6 Standard Widgets

The following is a list of all the standard widgets, and their default bindings in emacs mode, vi command mode and vi insert mode (the '`emacs`', '`vicmd`' and '`viins`' keymaps, respectively).

Note that cursor keys are bound to movement keys in all three keymaps; the shell assumes that the cursor keys send the key sequences reported by the terminal-handling library (termcap or terminfo). The key sequences shown in the list are those based on the VT100, common on many modern terminals, but in fact these are not necessarily bound. In the case of the `viins` keymap, the initial escape character of the sequences serves also to return to the `vicmd` keymap: whether this happens is determined by the `KEYTIMEOUT` parameter, see Chapter 14 [Parameters], page 46.

17.6.1 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) (unbound)
: Move backward one character.

`vi-backward-char` (unbound) (^H h ^?) (ESC-[D)
: 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) (ESC-[C)
: 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 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.

17.6.2 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) (ESC-[B)
: 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.

  If called from a function by the `zle` command with arguments, the first argument is taken as the string for which to search, rather than 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 search is case-insensitive if the search string does not have uppercase letters and no numeric argument was given. 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-char`, `clear-screen`, `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 '`vicmd`' keymaps; the '`main`' keymap (insert mode) will be selected initially. `history-incremental-search-backward` will get the next occurrence of the contents of the mini-buffer. `history-incremental-search-forward` inverts the sense of the search. `vi-repeat-search` and `vi-rev-repeat-search` are similarly supported. The direction of the search is indicated in the mini-buffer.

Any multi-character string that is not bound to one of the above functions will beep and interrupt the search, leaving the last found line in the 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.

When called from a widget function by the `zle` command, the incremental search commands can take a string argument. This will be treated as a string of keys, as for arguments to the `bindkey` command, and used as initial input for the command. Any characters in the string which are unused by the incremental search will be silently ignored. For example,

```
zle history-incremental-search-backward forceps
```

will search backwards for `forceps`, leaving the minibuffer containing the string '`forceps`'.

`history-incremental-search-forward` (^S ^Xs) (unbound) (unbound)
: Search forward incrementally for a specified string. The search is case-insensitive if the search string does not have uppercase letters and no numeric argument was given. 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.

  If called from a function by the `zle` command with arguments, the first argument is taken as the string for which to search, rather than 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`, `backward-delete-char`, `vi-backward-delete-char`, `backward-kill-word`, `vi-backward-kill-word`, `clear-screen`, `redisplay`, `quoted-insert` and `vi-quoted-insert`.

  `vi-cmd-mode` is treated the same as accept-line, and `magic-space` is treated as a space. 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.

  If called from a function by the `zle` command with arguments, the first argument is taken as the string for which to search, rather than the first word in the buffer.

`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.

  If called from a function by the `zle` command with arguments, the first argument is taken as the string for which to search, rather than 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`. Argument handling is also the same as for that command.

`infer-next-history` (^X^N) (unbound) (unbound)
: Search in the history list 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). Repeating this command replaces the word just inserted with the last word from the history event prior to the one just used; numeric arguments can be used in the same way to pick a word from that event.

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

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

`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.

  If called from a function by the `zle` command with arguments, the first argument is taken as the string for which to search, rather than 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 in its original position.

17.6.3 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`, change 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 to the left of the cursor.

`copy-prev-shell-word` (ESC-^_) (unbound) (unbound)
: Like `copy-prev-word`, but the word is found by using shell parsing, whereas `copy-prev-word` looks for blanks. This makes a difference when the word is quoted and contains spaces.

`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 first non-blank character on 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. If already on the end of the line, kill the newline character.

`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 cursor 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)
: Insert a character into the buffer at the cursor position.

`self-insert-unmeta` (ESC-^I ESC-^J ESC-^M) (unbound) (unbound)
: Insert a character into 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.

17.6.4 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`. This only works if bound to a key sequence ending in a decimal digit.

  Inside a widget function, a call to this function treats the last key of the key sequence which called the widget as the digit.

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

`universal-argument`
: Multiply the argument of the next command by 4. Alternatively, if this command is followed by an integer (positive or negative), use that as the argument for the next command. Thus digits cannot be repeated using this command. For example, if this command occurs twice, followed immediately by `forward-char`, move forward sixteen spaces; if instead it is followed by `-2`, then `forward-char`, move backward two spaces.

  Inside a widget function, if passed an argument, i.e. '`zle universal-argument` *num*', the numerical argument will be set to *num*; this is equivalent to '`NUMERIC=`*num*'.

17.6.5 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 the `MENU_COMPLETE` option.

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

`reverse-menu-complete`
: Perform menu completion, like `menu-complete`, except that if a menu completion is already in progress, move to the *previous* completion rather than the next.

`end-of-list`
: When a previous completion displayed a list below the prompt, this widget can be used to move the prompt below the list.

17.6.6 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)
: Finish editing the buffer. Normally this causes the buffer to be executed as a shell command.

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

`beep`
: Beep, unless the `BEEP` option is unset.

`vi-cmd-mode` (^X^V) (unbound) (^[)
: Enter command mode; that is, select the '`vicmd`' 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`
: Reads a key sequence, then prints the function bound to that sequence.

`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 an 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`, `backward-kill-word`, `vi-backward-kill-word`, `kill-whole-line`, `vi-kill-line`, `backward-kill-line`, `list-choices`,

`delete-char-or-list`, `complete-word`, `accept-line`, `expand-or-complete` and `expand-or-complete-prefix`.

`kill-region` kills the last word, and vi-cmd-mode is treated the same as accept-line. The space 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 other character that is not bound to `self-insert` or `self-insert-unmeta` will beep and be ignored. 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 multiline 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 multiline construct into the editor buffer. The latter is equivalent to `push-input` 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`
: This command is executed when a key sequence that is not bound to any command is typed. By default it beeps.

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

`redo`
: Incrementally redo undone text modifications.

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

`what-cursor-position` (`^X=`) (unbound) (unbound)
: Print the character under the cursor, its code as an octal, decimal and hexadecimal number, the current cursor position within the buffer and the column of the cursor in the current line.

`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.

18 Completion Widgets

18.1 Description

The shell's programmable completion mechanism can be manipulated in two ways; here the low-level features supporting the newer, function-based mechanism are defined. A complete set of shell functions based on these features is described in the next chapter, Chapter 19 [Completion System], page 127, and users with no interest in adding to that system (or, potentially, writing their own — see dictionary entry for 'hubris') should skip this section. The older system based on the `compctl` builtin command is described in the chapter Chapter 20 [Completion Using compctl], page 179.

Completion widgets are defined by the `-C` option to the `zle` builtin command provided by the `zsh/zle` module (see Section 21.19 [The zsh/zle Module], page 208). For example,

```
zle -C complete expand-or-complete completer
```

defines a widget named '`complete`'. The second argument is the name of any of the builtin widgets that handle completions: `complete-word`, `expand-or-complete`, `expand-or-complete-prefix`, `menu-complete`, `menu-expand-or-complete`, `reverse-menu-complete`, `list-choices`, or `delete-char-or-list`. Note that this will still work even if the widget in question has been re-bound.

When this newly defined widget is bound to a key using the `bindkey` builtin command defined in the `zsh/zle` module (Chapter 17 [Zsh Line Editor], page 94), typing that key will call the shell function '`completer`'. This function is responsible for generating the possible matches using the builtins described below. As with other ZLE widgets, the function is called with its standard input closed.

Once the function returns, the completion code takes over control again and treats the matches in the same manner as the specified builtin widget, in this case `expand-or-complete`.

18.2 Special Parameters

Inside completion widgets, and any functions called from them, some parameters have special meaning; outside these functions they are not special to the shell in any way. These parameters are used to pass information between the completion code and the completion widget. Some of the builtin commands and the condition codes use or change the current values of these parameters. Any existing values will be hidden during execution of completion widgets; except for `compstate`, the parameters are reset on each function exit (including nested function calls from within the completion widget) to the values they had when the function was entered.

`CURRENT` This is the number of the current word, i.e. the word the cursor is currently on in the `words` array. Note that this value is only correct if the `ksharrays` option is not set.

`IPREFIX` Initially this will be set to the empty string. This parameter functions like `PREFIX`; it contains a string which precedes the one in `PREFIX` and is not considered part of the list of matches. Typically, a string is transferred from the beginning of `PREFIX` to the end of `IPREFIX`, for example:

```
IPREFIX=${PREFIX%%\=*}=
PREFIX=${PREFIX#*=}
```

causes the part of the prefix up to and including the first equal sign not to be treated as part of a matched string. This can be done automatically by the `compset` builtin, see below.

`ISUFFIX` As `IPREFIX`, but for a suffix that should not be considered part of the matches; note that the `ISUFFIX` string follows the `SUFFIX` string.

`PREFIX` Initially this will be set to the part of the current word from the beginning of the word up to the position of the cursor; it may be altered to give a common prefix for all matches.

`QIPREFIX` This parameter is read-only and contains the quoted string up to the word being completed. E.g. when completing '`"foo`', this parameter contains the double quote. If the `-q` option of `compset` is used (see below), and the original string was '`"foo bar`' with the cursor on the '`bar`', this parameter contains '`"foo `'.

`QISUFFIX` Like `QIPREFIX`, but containing the suffix.

`SUFFIX` Initially this will be set to the part of the current word from the cursor position to the end; it may be altered to give a common suffix for all matches. It is most useful when the option `COMPLETE_IN_WORD` is set, as otherwise the whole word on the command line is treated as a prefix.

`compstate`
: This is an associative array with various keys and values that the completion code uses to exchange information with the completion widget. The keys are:

  `all_quotes`
  : The `-q` option of the `compset` builtin command (see below) allows a quoted string to be broken into separate words; if the cursor is on one of those words, that word will be completed, possibly invoking '`compset -q`' recursively. With this key it is possible to test the types of quoted strings which are currently broken into parts in this fashion. Its value contains one character for each quoting level. The characters are a single quote or a double quote for strings quoted with these characters and a backslash for strings not starting with a quote character. The first character in the value always corresponds to the innermost quoting level.

  `context`
  : This will be set by the completion code to the overall context in which completion is attempted. Possible values are:

    `array_value`
    : when completing inside the value of an array parameter assignment; in this case the `words` array contains the words inside the parentheses.

    `brace_parameter`
    : when completing the name of a parameter in a parameter expansion beginning with `${`.

    `command`
    : when completing for a normal command (either in command position or for an argument of the command).

    `condition`
    : when completing inside a '`[[...]]`' conditional expression; in this case the `words` array contains only the words inside the conditional expression.

    `math`
    : when completing in a mathematical environment such as a '`((...))`' construct.

    `parameter`
    : when completing the name of a parameter in a parameter expansion beginning with `$` but not `${`.

    `redirect`
    : when completing after a redirection operator.

    `subscript`
    : when completing inside a parameter subscript.

    `value`
    : when completing the value of a parameter assignment.

  `exact`
  : Controls the behaviour when the `REC_EXACT` option is set. It will be set to `accept` if an exact match would be accepted, and will be unset otherwise.

    If it was set when at least one match equal to the string on the line was generated, the match is accepted.

  `exact_string`
  : The string of an exact match if one was found, otherwise unset.

  `ignored`
  : The number of words that were ignored because they matched one of the patterns given with the `-F` option to the `compadd` builtin command.

`insert` This controls the manner in which a match is inserted into the command line. On entry to the widget function, if it is unset the command line is not to be changed; if set to `unambiguous`, any prefix common to all matches is to be inserted; if set to `automenu-unambiguous`, the common prefix is to be inserted and the next invocation of the completion code may start menu completion (due to the `AUTO_MENU` option being set); if set to `menu` or `automenu` menu completion will be started for the matches currently generated (in the latter case this will happen because the `AUTO_MENU` is set). The value may also contain the string '`tab`' when the completion code would normally not really do completion, but only insert the TAB character.

On exit it may be set to any of the values above (where setting it to the empty string is the same as unsetting it), or to a number, in which case the match whose number is given will be inserted into the command line. Negative numbers count backward from the last match (with '`-1`' selecting the last match) and out-of-range values are wrapped around, so that a value of zero selects the last match and a value one more than the maximum selects the first. Unless the value of this key ends in a space, the match is inserted as in a menu completion, i.e. without automatically appending a space.

Both `menu` and `automenu` may also specify the the number of the match to insert, given after a colon. For example, '`menu:2`' says to start menu completion, beginning with the second match.

Note that a value containing the substring '`tab`' makes the matches generated be ignored and only the TAB be inserted.

Finally, it may also be set to `all`, which makes all matches generated be inserted into the line.

`insert_positions`
When the completion system inserts an unambiguous string into the line, there may be multiple places where characters are missing or where the character inserted differs from at least one match. The value of this key contains a colon separated list of all these positions, as indexes into the command line.

`last_prompt`
If this is set to a non-empty string for every match added, the completion code will move the cursor back to the previous prompt after the list of completions has been displayed. Initially this is set or unset according to the `ALWAYS_LAST_PROMPT` option.

`list` This controls whether or how the list of matches will be displayed. If it is unset or empty they will never be listed; if its value begins with `list`, they will always be listed; if it begins with `autolist` or `ambiguous`, they will be listed when the `AUTO_LIST` or `LIST_AMBIGUOUS` options respectively would normally cause them to be.

If the substring `force` appears in the value, this makes the list be shown even if there is only one match. Normally, the list would be shown only if there are at least two matches.

The value contains the substring `packed` if the `LIST_PACKED` option is set. If this substring is given for all matches added to a group, this group will show the `LIST_PACKED` behavior. The same is done for the `LIST_ROWS_FIRST` option with the substring `rows`.

Finally, if the value contains the string `explanations`, only the explanation strings, if any, will be listed and if it contains `messages`, only the messages (added with the `-x` option of `compadd`) will be listed. If it contains both `explanations` and `messages` both kinds of explanation strings will be listed. It will be set appropriately on entry to a completion widget and may be changed there.

`list_lines`
: This gives the number of lines that are needed to display the full list of completions. Note that to calculate the total number of lines to display you need to add the number of lines needed for the command line to this value, this is available as the value of the `BUFFERLINES` special parameter.

`list_max` Initially this is set to the value of the `LISTMAX` parameter. It may be set to any other value; when the widget exits this value will be used in the same way as the value of `LISTMAX`.

`nmatches` The number of matches generated and accepted by the completion code so far.

`old_insert`
: On entry to the widget this will be set to the number of the match of an old list of completions that is currently inserted into the command line. If no match has been inserted, this is unset.

  As with `old_list`, the value of this key will only be used if it is the string `keep`. If it was set to this value by the widget and there was an old match inserted into the command line, this match will be kept and if the value of the `insert` key specifies that another match should be inserted, this will be inserted after the old one.

`old_list` This is set to `yes` if there is still a valid list of completions from a previous completion at the time the widget is invoked. This will usually be the case if and only if the previous editing operation was a completion widget or one of the builtin completion functions. If there is a valid list and it is also currently shown on the screen, the value of this key is `shown`.

  After the widget has exited the value of this key is only used if it was set to `keep`. In this case the completion code will continue to use this old list. If the widget generated new matches, they will not be used.

`parameter`
: The name of the parameter when completing in a subscript or in the value of a parameter assignment.

`pattern_insert`
: Normally this is set to `menu`, which specifies that menu completion will be used whenever a set of matches was generated using pattern matching. If it is set to any other non-empty string by the user and menu completion is not selected by other option settings, the code will instead insert any common prefix for the generated matches as with normal completion.

`pattern_match`
: Locally controls the behaviour given by the `GLOB_COMPLETE` option. Initially it is set to '`*`' if and only if the option is set. The completion widget may set it to this value, to an empty string (which has the same effect as unsetting it), or to any other non-empty string. If it is non-empty,

unquoted metacharacters on the command line will be treated as patterns; if it is '`*`', then additionally a wildcard '`*`' is assumed at the cursor position; if it is empty or unset, metacharacters will be treated literally.

Note that the matcher specifications given to the `compadd` builtin command are not used if this is set to a non-empty string.

`quote` When completing inside quotes, this contains the quotation character (i.e. either a single quote, a double quote, or a backtick). Otherwise it is unset.

`quoting` When completing inside single quotes, this is set to the string `single`; inside double quotes, the string `double`; inside backticks, the string `backtick`. Otherwise it is unset.

`redirect` The redirection operator when completing in a redirection position, i.e. one of `<`, `>`, etc.

`restore` This is set to `auto` before a function is entered, which forces the special parameters mentioned above (`words`, `CURRENT`, `PREFIX`, `IPREFIX`, `SUFFIX`, and `ISUFFIX`) to be restored to their previous values when the function exits. If a function unsets it or sets it to any other string, they will not be restored.

`to_end` Specifies the occasions on which the cursor is moved to the end of a string when a match is inserted. On entry to a widget function, it may be `single` if this will happen when a single unambiguous match was inserted or `match` if it will happen any time a match is inserted (for example, by menu completion; this is likely to be the effect of the `ALWAYS_TO_END` option).

On exit, it may be set to `single` as above. It may also be set to `always`, or to the empty string or unset; in those cases the cursor will be moved to the end of the string always or never respectively. Any other string is treated as `match`.

`unambiguous`
This key is read-only and will always be set to the common (unambiguous) prefix the completion code has generated for all matches added so far.

`unambiguous_cursor`
This gives the position the cursor would be placed at if the common prefix in the `unambiguous` key were inserted, relative to the value of that key. The cursor would be placed before the character whose index is given by this key.

`unambiguous_positions`
This contains all positions where characters in the unambiguous string are missing or where the character inserted differs from at least one of the matches. The positions are given as indexes into the string given by the value of the `unambiguous` key.

`vared` If completion is called while editing a line using the `vared` builtin, the value of this key is set to the name of the parameter given as an argument to `vared`. This key is only set while a `vared` command is active.

`words` This array contains the words present on the command line currently being edited.

18.3 Builtin Commands

`compadd` [`-akqQfenUl12C`] [`-F` *array*]
[`-P` *prefix*] [`-S` *suffix*]
[`-p` *hidden-prefix*] [`-s` *hidden-suffix*]
[`-i` *ignored-prefix*] [`-I` *ignored-suffix*]
[`-W` *file-prefix*] [`-d` *array*]
[`-J` *name*] [`-V` *name*] [`-X` *explanation*] [`-x` *message*]
[`-r` *remove-chars*] [`-R` *remove-func*]
[`-D` *array*] [`-O` *array*] [`-A` *array*]
[`-M` *match-spec*] [`--`] [*words* ...]

This builtin command can be used to add matches directly and control all the information the completion code stores with each possible match. The return value is zero if at least one match was added and non-zero if no matches were added.

The completion code breaks the string to complete into seven fields in the order:

<*ipre*><*apre*><*hpre*><*word*><*hsuf*><*asuf*><*isuf*>

The first field is an ignored prefix taken from the command line, the contents of the `IPREFIX` parameter plus the string given with the `-i` option. With the `-U` option, only the string from the `-i` option is used. The field <*apre*> is an optional prefix string given with the `-P` option. The <*hpre*> field is a string that is considered part of the match but that should not be shown when listing completions, given with the `-p` option; for example, functions that do filename generation might specify a common path prefix this way. <*word*> is the part of the match that should appear in the list of completions, i.e. one of the *words* given at the end of the `compadd` command line. The suffixes <*hsuf*>, <*asuf*> and <*isuf*> correspond to the prefixes <*hpre*>, <*apre*> and <*ipre*> and are given by the options `-s`, `-S` and `-I`, respectively.

The supported flags are:

`-P` *prefix*
: This gives a string to be inserted before the given *words*. The string given is not considered as part of the match and any shell metacharacters in it will not be quoted when the string is inserted.

`-S` *suffix*
: Like `-P`, but gives a string to be inserted after the match.

`-p` *hidden-prefix*
: This gives a string that should be inserted into the command line before the match but that should not appear in the list of matches. Unless the `-U` option is given, this string must be matched as part of the string on the command line.

`-s` *hidden-suffix*
: Like '`-p`', but gives a string to insert after the match.

`-i` *ignored-prefix*
: This gives a string to insert into the command line just before any string given with the '`-P`' option. Without '`-P`' the string is inserted before the string given with '`-p`' or directly before the match.

`-I` *ignored-suffix*
: Like `-i`, but gives an ignored suffix.

`-a`
: With this flag the *words* are taken as names of arrays and the possible matches are their values. If only some elements of the arrays are needed, the *words* may also contain subscripts, as in '`foo[2,-1]`'.

`-k` With this flag the *words* are taken as names of associative arrays and the possible matches are their keys. As for `-a`, the *words* may also contain subscripts, as in '`foo[(R)*bar*]`'.

`-d` *array* This adds per-match display strings. The *array* should contain one element per *word* given. The completion code will then display the first element instead of the first *word*, and so on. The *array* may be given as the name of an array parameter or directly as a space-separated list of words in parentheses.

If there are fewer display strings than *words*, the leftover *words* will be displayed unchanged and if there are more display strings than *words*, the leftover display strings will be silently ignored.

`-l` This option only has an effect if used together with the `-d` option. If it is given, the display strings are listed one per line, not arrayed in columns.

`-J` *name* Gives the name of the group of matches the words should be stored in.

`-V` *name* Like `-J` but naming a unsorted group. These are in a different name space than groups created with the `-J` flag.

`-1` If given together with the `-V` option, makes only consecutive duplicates in the group be removed. If combined with the `-J` option, this has no visible effect. Note that groups with and without this flag are in different name spaces.

`-2` If given together with the `-J` or `-V` option, makes all duplicates be kept. Again, groups with and without this flag are in different name spaces.

`-X` *explanation*
The *explanation* string will be printed with the list of matches, above the group currently selected.

`-x` *message*
Like `-X`, but the *message* will be printed even if there are no matches in the group.

`-q` The suffix given with `-S` will be automatically removed if the next character typed is a blank or does not insert anything, or if the suffix consists of only one character and the next character typed is the same character.

`-r` *remove-chars*
This is a more versatile form of the `-q` option. The suffix given with `-S` or the slash automatically added after completing directories will be automatically removed if the next character typed inserts one of the characters given in the *remove-chars*. This string is parsed as a characters class and understands the backslash sequences used by the `print` command. For example, '`-r "a-z\t"`' removes the suffix if the next character typed inserts a lowercase character or a TAB, and '`-r "^0-9"`' removes the suffix if the next character typed inserts anything but a digit. One extra backslash sequence is understood in this string: '`\-`' stands for all characters that insert nothing. Thus '`-S "=" -q`' is the same as '`-S "=" -r "= \t\n\-"`'.

`-R` *remove-func*
This is another form of the `-r` option. When a suffix has been inserted and the completion accepted, the function *remove-func* will be called after the next character typed. It is passed the length of the suffix as an

argument and can use the special parameters available in ordinary (non-completion) zle widgets (see Chapter 17 [Zsh Line Editor], page 94) to analyse and modify the command line.

`-f` If this flag is given, all of the matches built from *words* are marked as being the names of files. They are not required to be actual filenames, but if they are, and the option `LIST_TYPES` is set, the characters describing the types of the files in the completion lists will be shown. This also forces a slash to be added when the name of a directory is completed.

`-e` This flag can be used to tell the completion code that the matches added are parameter names for a parameter expansion. This will make the `AUTO_PARAM_SLASH` and `AUTO_PARAM_KEYS` options be used for the matches.

`-W` *file-prefix*
This string is a pathname that will be prepended to each of the matches formed by the given *words* together with any prefix specified by the `-p` option to form a complete filename for testing. Hence it is only useful if combined with the `-f` flag, as the tests will not otherwise be performed.

`-F` *array* Specifies an array containing patterns. Words matching one of these patterns are ignored, i.e. not considered to be possible matches.

The *array* may be the name of an array parameter or a list of literal patterns enclosed in parentheses and quoted, as in '`-F "(*?.o *?.h)"`'. If the name of an array is given, the elements of the array are taken as the patterns.

`-Q` This flag instructs the completion code not to quote any metacharacters in the words when inserting them into the command line.

`-M` *match-spec*
This gives local match specifications as described below in Section 18.5 [Matching Control], page 124. This option may be given more than once. In this case all *match-spec*s given are concatenated with spaces between them to form the specification string to use. Note that they will only be used if the `-U` option is not given.

`-n` Specifies that the words added are to be used as possible matches, but are not to appear in the completion listing.

`-U` If this flag is given, all words given will be accepted and no matching will be done by the completion code. Normally this is used in functions that do the matching themselves.

`-O` *array* If this option is given, the *words* are *not* added to the set of possible completions. Instead, matching is done as usual and all of the *words* given as arguments that match the string on the command line will be stored in the array parameter whose name is given as *array*.

`-A` *array* As the `-O` option, except that instead of those of the *words* which match being stored in *array*, the strings generated internally by the completion code are stored. For example, with a matching specification of '`-M "L:|no="`', the string '`nof`' on the command line and the string '`foo`' as one of the *words*, this option stores the string '`nofoo`' in the array, whereas the `-O` option stores the '`foo`' originally given.

`-D` *array* As with `-O`, the *words* are not added to the set of possible completions. Instead, the completion code tests whether each *word* in turn matches

what is on the line. If the *n*'th *word* does not match, the *n*'th element of the *array* is removed. Elements for which the corresponding *word* is matched are retained.

`-C` This option adds a special match which expands to all other matches when inserted into the line, even those that are added after this option is used. Together with the `-d` option it is possible to specify a string that should be displayed in the list for this special match. If no string is given, it will be shown as a string containing the strings that would be inserted for the other matches, truncated to the width of the screen.

`-`
`--` This flag ends the list of flags and options. All arguments after it will be taken as the words to use as matches even if they begin with hyphens.

Except for the `-M` flag, if any of these flags is given more than once, the first one (and its argument) will be used.

`compset -p` *number*
`compset -P` [*number*] *pattern*
`compset -s` *number*
`compset -S` [*number*] *pattern*
`compset -n` *begin* [*end*]
`compset -N` *beg-pat* [*end-pat*]
`compset -q`

This command simplifies modification of the special parameters, while its return value allows tests on them to be carried out.

The options are:

`-p` *number*
If the contents of the `PREFIX` parameter is longer than *number* characters, the first *number* characters are removed from it and appended to the contents of the `IPREFIX` parameter.

`-P` [*number*] *pattern*
If the value of the `PREFIX` parameter begins with anything that matches the *pattern*, the matched portion is removed from `PREFIX` and appended to `IPREFIX`.

Without the optional *number*, the longest match is taken, but if *number* is given, anything up to the *number*'th match is moved. If the *number* is negative, the *number*'th longest match is moved. For example, if `PREFIX` contains the string '`a=b=c`', then `compset -P '*\='` will move the string '`a=b=`' into the `IPREFIX` parameter, but `compset -P 1 '*\='` will move only the string '`a=`'.

`-s` *number*
As `-p`, but transfer the last *number* characters from the value of `SUFFIX` to the front of the value of `ISUFFIX`.

`-S` [*number*] *pattern*
As `-P`, but match the last portion of `SUFFIX` and transfer the matched portion to the front of the value of `ISUFFIX`.

`-n` *begin* [*end*]
If the current word position as specified by the parameter `CURRENT` is greater than or equal to *begin*, anything up to the *begin*'th word is removed from the `words` array and the value of the parameter `CURRENT` is decremented by *begin*.

If the optional *end* is given, the modification is done only if the current word position is also less than or equal to *end*. In this case, the words from position *end* onwards are also removed from the `words` array.

Both *begin* and *end* may be negative to count backwards from the last element of the `words` array.

`-N` *beg-pat* [*end-pat*]
If one of the elements of the `words` array before the one at the index given by the value of the parameter `CURRENT` matches the pattern *beg-pat*, all elements up to and including the matching one are removed from the `words` array and the value of `CURRENT` is changed to point to the same word in the changed array.

If the optional pattern *end-pat* is also given, and there is an element in the `words` array matching this pattern, the parameters are modified only if the index of this word is higher than the one given by the `CURRENT` parameter (so that the matching word has to be after the cursor). In this case, the words starting with the one matching `end-pat` are also removed from the `words` array. If `words` contains no word matching *end-pat*, the testing and modification is performed as if it were not given.

`-q` The word currently being completed is split on spaces into separate words, respecting the usual shell quoting conventions. The resulting words are stored in the `words` array, and `CURRENT`, `PREFIX`, `SUFFIX`, `QIPREFIX`, and `QISUFFIX` are modified to reflect the word part that is completed.

In all the above cases the return value is zero if the test succeeded and the parameters were modified and non-zero otherwise. This allows one to use this builtin in tests such as:

```
if compset -P '*\='; then ...
```

This forces anything up to and including the last equal sign to be ignored by the completion code.

`compcall` [`-TD`]
This allows the use of completions defined with the `compctl` builtin from within completion widgets. The list of matches will be generated as if one of the non-widget completion function (`complete-word`, etc.) had been called, except that only `compctl`s given for specific commands are used. To force the code to try completions defined with the `-T` option of `compctl` and/or the default completion (whether defined by `compctl -D` or the builtin default) in the appropriate places, the `-T` and/or `-D` flags can be passed to `compcall`.

The return value can be used to test if a matching `compctl` definition was found. It is non-zero if a `compctl` was found and zero otherwise.

Note that this builtin is defined by the `zsh/compctl` module.

18.4 Condition Codes

The following additional condition codes for use within the `[[ ... ]]` construct are available in completion widgets. These work on the special parameters. All of these tests can also be performed by the `compset` builtin, but in the case of the condition codes the contents of the special parameters are not modified.

`-prefix` [*number*] *pattern*
: true if the test for the `-P` option of `compset` would succeed.

`-suffix` [*number*] *pattern*
: true if the test for the `-S` option of `compset` would succeed.

`-after` *beg-pat*
: true if the test of the `-N` option with only the *beg-pat* given would succeed.

`-between` *beg-pat end-pat*
: true if the test for the `-N` option with both patterns would succeed.

18.5 Matching Control

It is possible by use of the `-M` option of the `compadd` builtin command to specify how the characters in the string to be completed (referred to here as the command line) map onto the characters in the list of matches produced by the completion code (referred to here as the trial completions). Note that this is not used if the command line contains a glob pattern and the `GLOB_COMPLETE` option is set or the `pattern_match` of the `compstate` special association is set to a non-empty string.

The *match-spec* given as the argument to the `-M` option (see Section 18.3 [Builtin Commands], page 119) consists of one or more matching descriptions separated by whitespace. Each description consists of a letter followed by a colon and then the patterns describing which character sequences on the line match which character sequences in the trial completion. Any sequence of characters not handled in this fashion must match exactly, as usual.

The forms of *match-spec* understood are as follows. In each case, the form with an uppercase initial character retains the string already typed on the command line as the final result of completion, while with a lowercase initial character the string on the command line is changed into the corresponding part of the trial completion.

`m:`*lpat*`=`*tpat*
`M:`*lpat*`=`*tpat*
: Here, *lpat* is a pattern that matches on the command line, corresponding to *tpat* which matches in the trial completion.

`l:`*lanchor*`|`*lpat*`=`*tpat*
`L:`*lanchor*`|`*lpat*`=`*tpat*
`l:`*lanchor*`||`*ranchor*`=`*tpat*
`L:`*lanchor*`||`*ranchor*`=`*tpat*
`b:`*lpat*`=`*tpat*
`B:`*lpat*`=`*tpat*
: These letters are for patterns that are anchored by another pattern on the left side. Matching for *lpat* and *tpat* is as for `m` and `M`, but the pattern *lpat* matched on the command line must be preceded by the pattern *lanchor*. The *lanchor* can be blank to anchor the match to the start of the command line string; otherwise the anchor can occur anywhere, but must match in both the command line and trial completion strings.

  If no *lpat* is given but a *ranchor* is, this matches the gap between substrings matched by *lanchor* and *ranchor*. Unlike *lanchor*, the *ranchor* only needs to match the trial completion string.

  The `b` and `B` forms are similar to `l` and `L` with an empty anchor, but need to match only the beginning of the trial completion or the word on the command line, respectively.

`r:`*lpat*`|`*ranchor*`=`*tpat*
`R:`*lpat*`|`*ranchor*`=`*tpat*
`r:`*lanchor*`||`*ranchor*`=`*tpat*
`R:`*lanchor*`||`*ranchor*`=`*tpat*
`e:`*lpat*`=`*tpat*
`E:`*lpat*`=`*tpat*

As `l`, `L`, `b` and `B`, with the difference that the command line and trial completion patterns are anchored on the right side. Here an empty *ranchor* and the `e` and `E` forms force the match to the end of the trial completion or command line string.

Each *lpat*, *tpat* or *anchor* is either an empty string or consists of a sequence of literal characters (which may be quoted with a backslash), question marks, character classes, and correspondence classes; ordinary shell patterns are not used. Literal characters match only themselves, question marks match any character, and character classes are formed as for globbing and match any character in the given set.

Correspondence classes are defined like character classes, but with two differences: they are delimited by a pair of braces, and negated classes are not allowed, so the characters `!` and `^` have no special meaning directly after the opening brace. They indicate that a range of characters on the line match a range of characters in the trial completion, but (unlike ordinary character classes) paired according to the corresponding position in the sequence. For example, to make any lowercase letter on the line match the corresponding uppercase letter in the trial completion, you can use '`m:{a-z}={A-Z}`'. More than one pair of classes can occur, in which case the first class before the `=` corresponds to the first after it, and so on. If one side has more such classes than the other side, the superfluous classes behave like normal character classes. In anchor patterns correspondence classes also behave like normal character classes.

The pattern *tpat* may also be one or two stars, '`*`' or '`**`'. This means that the pattern on the command line can match any number of characters in the trial completion. In this case the pattern must be anchored (on either side); in the case of a single star, the *anchor* then determines how much of the trial completion is to be included — only the characters up to the next appearance of the anchor will be matched. With two stars, substrings matched by the anchor can be matched, too.

Examples:

The keys of the `options` association defined by the `parameter` module are the option names in all-lowercase form, without underscores, and without the optional `no` at the beginning even though the builtins `setopt` and `unsetopt` understand option names with uppercase letters, underscores, and the optional `no`. The following alters the matching rules so that the prefix `no` and any underscore are ignored when trying to match the trial completions generated and uppercase letters on the line match the corresponding lowercase letters in the words:

```
compadd -M 'L:|[nN][oO]= M:_= M:{A-Z}={a-z}' - \
  ${(k)options}
```

The first part says that the pattern '`[nN][oO]`' at the beginning (the empty anchor before the pipe symbol) of the string on the line matches the empty string in the list of words generated by completion, so it will be ignored if present. The second part does the same for an underscore anywhere in the command line string, and the third part uses correspondence classes so that any uppercase letter on the line matches the corresponding lowercase letter in the word. The use of the uppercase forms of the specification characters (`L` and `M`) guarantees that what has already been typed on the command line (in particular the prefix `no`) will not be deleted.

Note that the use of `L` in the first part means that it matches only when at the beginning of both the command line string and the trial completion. I.e., the string '`_NO_f`' would not be completed to '`_NO_foo`', nor would '`NONO_f`' be completed to '`NONO_foo`' because of the leading underscore or the second '`NO`' on the line which makes the pattern fail even though they are otherwise ignored. To fix this, one would use '`B:[nN][oO]=`' instead of the first part. As described above, this matches at the beginning of the trial completion, independent of other

characters or substrings at the beginning of the command line word which are ignored by the same or other *match-spec*s.

The second example makes completion case insensitive. This is just the same as in the option example, except here we wish to retain the characters in the list of completions:

```
compadd -M 'm:{a-z}={A-Z}' ...
```

This makes lowercase letters match their uppercase counterparts. To make uppercase letters match the lowercase forms as well:

```
compadd -M 'm:{a-zA-Z}={A-Za-z}' ...
```

A nice example for the use of `*` patterns is partial word completion. Sometimes you would like to make strings like '`c.s.u`' complete to strings like '`comp.source.unix`', i.e. the word on the command line consists of multiple parts, separated by a dot in this example, where each part should be completed separately — note, however, that the case where each part of the word, i.e. '`comp`', '`source`' and '`unix`' in this example, is to be completed from separate sets of matches is a different problem to be solved by the implementation of the completion widget. The example can be handled by:

```
compadd -M 'r:|.=* r:|=*' \
  - comp.sources.unix comp.sources.misc ...
```

The first specification says that *lpat* is the empty string, while *anchor* is a dot; *tpat* is `*`, so this can match anything except for the '`.`' from the anchor in the trial completion word. So in '`c.s.u`', the matcher sees '`c`', followed by the empty string, followed by the anchor '`.`', and likewise for the second dot, and replaces the empty strings before the anchors, giving '`c`[`omp`]`.s`[`ources`]`.u`[`nix`]', where the last part of the completion is just as normal.

With the pattern shown above, the string '`c.u`' could not be completed to '`comp.sources.unix`' because the single star means that no dot (matched by the anchor) can be skipped. By using two stars as in '`r:|.=**`', however, '`c.u`' could be completed to '`comp.sources.unix`'. This also shows that in some cases, especially if the anchor is a real pattern, like a character class, the form with two stars may result in more matches than one would like.

The second specification is needed to make this work when the cursor is in the middle of the string on the command line and the option `COMPLETE_IN_WORD` is set. In this case the completion code would normally try to match trial completions that end with the string as typed so far, i.e. it will only insert new characters at the cursor position rather then at the end. However in our example we would like the code to recognise matches which contain extra characters after the string on the line (the '`nix`' in the example). Hence we say that the empty string at the end of the string on the line matches any characters at the end of the trial completion.

More generally, the specification

```
compadd -M 'r:|[.,_-]=* r:|=*' ...
```

allows one to complete words with abbreviations before any of the characters in the square brackets. For example, to complete `veryverylongfile.c` rather than `veryverylongheader.h` with the above in effect, you can just type `very.c` before attempting completion.

The specifications with both a left and a right anchor are useful to complete partial words whose parts are not separated by some special character. For example, in some places strings have to be completed that are formed '`LikeThis`' (i.e. the separate parts are determined by a leading uppercase letter) or maybe one has to complete strings with trailing numbers. Here one could use the simple form with only one anchor as in:

```
compadd -M 'r:|[A-Z0-9]=* r:|=*' LikeTHIS FooHoo 5foo123 5bar234
```

But with this, the string '`H`' would neither complete to '`FooHoo`' nor to '`LikeTHIS`' because in each case there is an uppercase letter before the '`H`' and that is matched by the anchor. Likewise, a '`2`' would not be completed. In both cases this could be changed by using '`r:|[A-Z0-9]=**`', but then '`H`' completes to both '`LikeTHIS`' and '`FooHoo`' and a '`2`' matches the other strings

because characters can be inserted before every uppercase letter and digit. To avoid this one would use:

```
compadd -M 'r:[^A-Z0-9]||[A-Z0-9]=** r:|=*' \
    LikeTHIS FooHoo foo123 bar234
```

By using these two anchors, a 'H' matches only uppercase 'H's that are immediately preceded by something matching the left anchor '[^A-Z0-9]'. The effect is, of course, that 'H' matches only the string 'FooHoo', a '2' matches only 'bar234' and so on.

When using the completion system (see Chapter 19 [Completion System], page 127), users can define match specifications that are to be used for specific contexts by using the `matcher` and `matcher-list` styles. The values for the latter will be used everywhere.

18.6 Completion Widget Example

The first step is to define the widget:

```
zle -C complete complete-word complete-files
```

Then the widget can be bound to a key using the `bindkey` builtin command:

```
bindkey '^X\t' complete
```

After that the shell function `complete-files` will be invoked after typing control-X and TAB. The function should then generate the matches, e.g.:

```
complete-files () { compadd - * }
```

This function will complete files in the current directory matching the current word.

19 Completion System

19.1 Description

This describes the shell code for the new completion system. It consists of various shell functions; those beginning 'comp' are to be called directly by the user, while those beginning '_' are called by the completion code. The shell functions of the second set which implement completion behaviour and which may be bound to keystrokes, are referred to as 'widgets'.

19.2 Initialization

If the system was installed completely, it should be enough to call the shell function `compinit` from your initialization file; see the next section. However, the function `compinstall` can be run by a user to configure various aspects of the completion system.

Usually, `compinstall` will insert code into `.zshrc`, although if that is not writable it will save it in another file and tell you that file's location. Note that it is up to you to make sure that the lines added to `.zshrc` are actually run; you may, for example, need to move them to an earlier place in the file if `.zshrc` usually returns early. So long as you keep them all together (including the comment lines at the start and finish), you can rerun `compinstall` and it will correctly locate and modify these lines. Note, however, that any code you add to this section by hand is likely to be lost if you rerun `compinstall`, although lines using the command 'zstyle' should be gracefully handled.

The new code will take effect next time you start the shell, or run `.zshrc` by hand; there is also an option to make them take effect immediately. However, if `compinstall` has removed definitions, you will need to restart the shell to see the changes.

To run `compinstall` you will need to make sure it is in a directory mentioned in your `fpath` parameter, which should already be the case if zsh was properly configured as long as your startup files do not remove the appropriate directories from `fpath`. Then it must be autoloaded ('`autoload -U compinstall`' is recommended). You can abort the installation any time you are being prompted for information, and your `.zshrc` will not be altered at all; changes only take place right at the end, where you are specifically asked for confirmation.

19.2.1 Use of compinit

This section describes the use of `compinit` to initialize completion for the current session when run directly by the user; if you have run `compinstall` it will be called automatically from your `.zshrc`.

To initialize the system, the function `compinit` should be in a directory mentioned in the `fpath` parameter, and should be autoloaded ('`autoload -U compinit`' is recommended), and then run simply as '`compinit`'. This will define a few utility functions, arrange for all the necessary shell functions to be autoloaded, and will then re-define all widgets that do completion to use the new system. If you use the `menu-select` widget, which is part of the `zsh/complist` module, you should make sure that that module is loaded before the call to `compinit` so that that widget is also re-defined. If completion styles (see below) are set up to perform expansion as well as completion by default, and the TAB key is bound to `expand-or-complete`, `compinit` will rebind it to `complete-word`; this is necessary to use the correct form of expansion.

Should you need to use the original completion commands, you can still bind keys to the old widgets by putting a '`.`' in front of the widget name, e.g. '`.expand-or-complete`'.

To speed up the running of `compinit`, it can be made to produce a dumped configuration which will be read in on future invocations; this is the default, although it can be turned off by calling `compinit` with the option `-D`. The dumped file is `.zcompdump` in the same directory as the startup files (i.e. `$ZDOTDIR` or `$HOME`); alternatively, an explicit file name can be given by '`compinit -d` *dumpfile*'. On the next call to `compinit`, it will read the dumped file instead of performing a full initialization.

If the number of completion files changes, `compinit` will recognise this and produce a new dump file. However, if the name of a function or the arguments in the first line of a `#compdef` function (as described below) change, it is easiest to delete the dump file by hand so that `compinit` will re-create it the next time it is run. The check performed to see if there are new functions can be omitted by giving the option `-C`. In this case the dump file will only be created if there isn't one already.

The dumping is actually done by another function, `compdump`, but you will only need to run this yourself if you change the configuration (e.g. using `compdef`) and then want to dump the new one. The name of the old dumped file will be remembered for this purpose.

If the parameter `_compdir` is set, `compinit` uses it as a directory where completion functions can be found; this is only necessary if they are not already in the function search path.

For security reasons `compinit` also checks if the completion system would use files not owned by root or by the current user, or files in directories that are world- or group-writable or that are not owned by root or by the current user. If such files or directories are found, `compinit` will ask if the completion system should really be used. To avoid these tests and make all files found be used without asking, use the option `-u`, and to make `compinit` silently ignore all insecure files and directories use the option `-i`. This security check is skipped entirely when the `-C` option is given.

The security check can be retried at any time by running the function `compaudit`. This is the same check used by `compinit`, but when it is executed directly any changes to `fpath` are made local to the function so they do not persist. The directories to be checked may be passed as arguments; if none are given, `compaudit` uses `fpath` and `_compdir` to find completion system directories, adding missing ones to `fpath` as necessary. To force a check of exactly the directories currently named in `fpath`, set `_compdir` to an empty string before calling `compaudit` or `compinit`.

19.2.2 Autoloaded files

The convention for autoloaded functions used in completion is that they start with an underscore; as already mentioned, the `fpath/FPATH` parameter must contain the directory in which they are stored. If `zsh` was properly installed on your system, then `fpath/FPATH` automatically contains the required directories for the standard functions.

For incomplete installations, if `compinit` does not find enough files beginning with an underscore (fewer than twenty) in the search path, it will try to find more by adding the directory `_compdir` to the search path. If that directory has a subdirectory named `Base`, all subdirectories will be added to the path. Furthermore, if the subdirectory `Base` has a subdirectory named `Core`, `compinit` will add all subdirectories of the subdirectories is to the path: this allows the functions to be in the same format as in the `zsh` source distribution.

When `compinit` is run, it searches all such files accessible via `fpath/FPATH` and reads the first line of each of them. This line should contain one of the tags described below. Files whose first line does not start with one of these tags are not considered to be part of the completion system and will not be treated specially.

The tags are:

`#compdef` *names...*
: The file will be made autoloadable and the function defined in it will be called when completing *names*, each of which is either the name of a command whose arguments are to be completed or one of a number of special contexts in the form *-context-* described below for the `_complete` function.

  Each *name* may also be of the form '*cmd=service*'. This is used by functions that offer multiple services, i.e. different completion behaviour for multiple commands. Such a string makes the completion system call the function when completing arguments for the command '`cmd`', setting the parameter `$service` to the string '`service`'. The function can then use that parameter to decide what to complete.

`#compdef -p` *pattern*
: The file will be made autoloadable and the function defined in it will be called when completing for a command whose name matches the given *pattern* (a standard globbing pattern). Note that only one *pattern* may be given.

`#compdef -P` *pattern*
: Like the previous one, but the function will be called only if no completion function for the command on the line could be found.

`#compdef -k` *style key-sequences...*
: This can be used to bind special completion functions to the *key-sequences* specified. It creates a widget behaving like the builtin widget *style*, which must be one of those that perform completion, namely `complete-word`, `delete-char-or-list`, `expand-or-complete`, `expand-or-complete-prefix`, `list-choices`, `menu-complete`, `menu-expand-or-complete`, or `reverse-menu-complete`. If the `zsh/complist` module is loaded (see Section 21.6 [The zsh/complist Module], page 189), the same happens to the `menu-select` widget.

The widget is then bound to all the *key-sequences* given, if any: when one of the *key-sequences* is typed, the function in the file will be invoked to generate the matches. Note that a key will not be re-bound if if it already was (that is, was bound to something other than `undefined-key`). The widget created has the same name as the file and can be bound to any other keys using `bindkey` as usual.

`#compdef -K` *widget-name style key-sequences ...*
This is similar to `-k`, with the same *style* and *key-sequences* arguments, preceded by a string giving the name of a widget. In this case only one *key-sequences* argument may be given, but the entire set of three arguments may be repeated with a different set of arguments. In particular, the *widget-name* must be distinct in each set. It should begin with '`_`', else one will be added, and should not clash with the name of any existing widget: names based on the name of the function are most useful. For example,

```
#compdef -K _foo_complete complete-word "^X^C" \
  _foo_list list-choices "^X^D"
```

(all on one line) defines a widget `_foo_complete` for completion, bound to '`^X^C`', and a widget `_foo_list` for listing, bound to '`^X^D`'.

`#autoload` [*options*]
This is used for files defining utility functions that are not to be called directly as completion functions but should be loaded automatically when invoked. Typically they are to be called from within one of the completion functions.

The *options* will be given to the `autoload` builtin command when making the function autoloaded. Most often, this will be `+X` to force the function to be loaded immediately. Note that the `-U` flag is always implicitly added.

The `#` is part of the tag name and no white space is allowed after it. The `#compdef` tags use the `compdef` function described below; the main difference is that the name of the function is supplied implicitly.

Note also that the functions for the completion system assume that the `KSH_AUTOLOAD` option is not set and cannot be loaded when it is set. To avoid having to unset `KSH_AUTOLOAD`, you can instead use one or more `zwc` file(s) which have been created with the command `zcompile -z` to load the functions for the completion system; see Chapter 16 [Shell Builtin Commands], page 74. This forces the functions to be autoloaded the way zsh normally loads functions.

19.2.3 Functions

The `compinit` file defines the following function, which may also be called directly by the user.

`compdef` [`-an`] *function names...*
`compdef -d` *names...*
`compdef -p` [`-a`] *function pattern*
`compdef -P` [`-a`] *function pattern*
`compdef -k` [`-an`] *function style key-sequences...*
`compdef -K` [`-an`] *function name style key-sequences ...*
The first form tells the completion system to call the given *function* when completing for the contexts or commands whose *names* are given: this is like the `#compdef` tag unless the first word contains an equal sign. In this case all words have to be of the form '*cmd=service*' where *service* is the name of a command or of a service defined by an autoloaded function with the `#compdef` tag and an argument of the form '*cmd=service*'. This kind of use makes the arguments of the *cmd*s be completed as those for the *service*s.

If the `-n` option is given, any existing completion behaviour for particular contexts or commands will not be altered. These definitions can be deleted by giving the `-d` option as in the second form.

The form with `-p` is similar to the first, but *function* will be called for all commands whose name matches the *pattern*; this is like the `#compdef -p` function tag.

The form with `-P` is like the third, but the *function* will be called only if no function for the command itself was found or if one was found and it set the `_compskip` parameter to a value *not* containing the substring `patterns`.

The form with `-k` defines a widget with the same name as the *function* which will be called for each of the *key-sequences*; this is like the `#compdef -k` tag. The function should generate the completions needed and will otherwise behave like the builtin widget whose name is given as the *style* argument. The widgets usable for this are: `complete-word`, `delete-char-or-list`, `expand-or-complete`, `expand-or-complete-prefix`, `list-choices`, `menu-complete`, `menu-expand-or-complete`, and `reverse-menu-complete`, as well as `menu-select` if the `zsh/complist` module is loaded. The option `-n` prevents the key being bound if it is already to bound to something other than `undefined-key`.

The form with `-K` is similar and defines multiple widgets based on the same *function*, each of which requires the set of three arguments *name*, *style* and *key-sequences*, where the latter two are as for `-k` and the first must be a unique widget name beginning with an underscore.

In each of the forms supporting it the `-a` option makes the *function* autoloadable (exactly equivalent to `autoload -U` *function*).

The `compdef` function is the place to turn to when one wants to define what the completion system should complete for a certain command. The function named can of course be one of the functions supplied or one written by the user. For example, if one has a command `foo` that gets process identifiers as arguments, one could do:

```
compdef _pids foo
```

using the `_pids` function from the distribution to generate the process identifiers. Not also the `_gnu_generic` function described below, which can be used to complete options for commands that understand the '`--help`' option.

19.3 Completion System Configuration

This section gives a short overview of how the completion system works, and then more detail on how users can configure how and when matches are generated.

19.3.1 Overview

When completion is attempted somewhere on a command line the completion system first tries to find out the context where completion was tried. The context depends on such things as the name of the command when completing an argument, and possibly also the name of an option when completing an argument to that option.

The 'context' of a completion is a string consisting of multiple fields. This is used to look up styles that can be used to configure the completion system. Since it is not possible to build the whole context string in advance, completion functions may modify some of the fields and hence the context used for lookup may vary during the same call to the completion system.

The context string always consists of the following fields, separated by colons and with a leading colon before the first:

- The literal string `completion`, saying that this style is used by the completion system.

- The *function*; in many cases this field will be blank, but when the completion system is called from other functions, like `predict-on` or one of the functions in the `Command` directory of the distribution, this field contains the name of that function, often in an abbreviated form.
- The *completer* currently active, which is the name of the function without the leading underscore. A 'completer' is in overall control of how completion is to be performed; '`complete`' is the basic one for ordinary completion, but completers may perform various related tasks such as correction, or modify the behaviour of a later completer (see Section 19.4 [Control Functions], page 154 for more information).
- The *context* or *command*. This is either one of the special context names such as `-condition-` as explained for the `_complete` completer below, or the name of the command we are completing arguments for. Completion functions for commands that have sub-commands usually modify this field to contain the name of the command followed by a minus sign and the sub-command (e.g. the completion function for the `cvs` command sets this field to strings such as `cvs-add` when completing for the `add` sub-command).
- The *argument*, describing which argument we are completing. Normally this is either a string of the form `argument-`*n*, where *n* is the number of the argument or it is a string of the form `option-`*opt*`-`*n* when completing the *n*'th argument of the option *opt*.
- The *tag*. Tags are used to discriminate between the types of matches a completion function can generate in a certain context.

As an example, the context name

```
:completion::complete:dvips:option-o-1:files
```

says that normal completion was attempted on an argument of the `dvips` command (more precisely: completion was attempted on the first argument after the `-o` option) and the completion function will generate filenames for this context.

In many of the possible contexts the completion system can generate matches, often multiple types of matches. These types are represented as simple names called 'tags'. The completion system will decide internally what sort of tags are allowed; a list of the standard possibilities is given below. To determine in which order the tags are to be used by the completion function, the '`tag-order`' style for the appropriate context may be set, as described in the list of standard styles below. Only those types of matches whose tags were selected by this style will be produced, and in the order given, although the default is to try all relevant tags in an order determined by the particular completion in use.

The `_complete_help` bindable command described in Section 19.5 [Bindable Commands], page 159 can be invoked to find out the context and tag names and styles used at a particular point in completion. It shows the list of contexts and tags that would be used in if completion were tried at the current cursor position. Hence one can easily find out all the information needed to change the behaviour of the `tag-order` style for a particular context.

Completion behaviour can be modified by various other styles defined with the `zstyle` builtin command (Section 21.23 [The zsh/zutil Module], page 210). When looking up styles the completion system uses full context names, including the tag.

Styles determine such things as how the matches are generated; some of them correspond to shell options (for example, the use of menu completion), but styles provide more specific control. They can have any number of strings as their value. Looking up the value of a style therefore consists of two things: the context, which may be matched as a pattern, and the name of the style itself, which must be given exactly.

For example, many completion functions can generate matches in a simple and a verbose form and use the `verbose` style to decide which form should be used. To make all such functions use the verbose form, put

```
zstyle ':completion:*' verbose yes
```

in one of the startup files like `.zshrc`; this sort of style can also be configured with the `compinstall` function. This definition simply means that the `verbose` style has `yes` as its value in every context inside the completion system. If the context pattern were '`*`', the verbose style would have this value anywhere the style mechanism is used, not just in completion.

As a more specific example, the completion function for the `kill` builtin command uses the `verbose` style to decide if jobs and processes are listed only as job numbers and process identifiers or if they are listed with the full job texts and the command lines of the processes (the latter is achieved by calling the `ps` command). To make this builtin list the matches only as numbers one could call:

```
zstyle ':completion:*:*:kill:*' verbose no
```

Furthermore, if one wanted to see the command lines for processes but not the job texts one could use the fact that the context name contains the tag name when styles are looked up. As the function for the `kill` builtin command uses the tags `jobs` and `processes`, we can use:

```
zstyle ':completion:*:*:kill:*:jobs' verbose no
```

To have more control over when certain values for styles are used one can use the special parameters available in completion widgets (see Chapter 18 [Completion Widgets], page 113)) and the `-e` option to `zstyle` that makes the value be evaluated when looked up. For example, to make the `completer` style have a different value when completing for the `cvs` command, one could use the `words` special array:

```
zstyle -e ':completion:*' completer '
    if [[ $words[1] = cvs ]]; then
      reply=(_complete)
    else
      reply=(_complete _approximate)
    fi'
```

One should be careful not to use too complicated code with this option, at least for the styles that are looked up quite often. These are basically those that define some global completion behaviour but allow that to be different for all matches or groups of matches (such as the `menu` and `list-rows-first` styles). Alternatively one can always use a less general pattern for the context than in the example above and use a second call to `zstyle` with a generic pattern and without using the `-e` option to define the default behaviour.

Note that the order in which styles are *defined* does not matter; the style mechanism uses the most specific possible match for a particular style to determine the set of values. More precisely, strings are preferred over patterns (for example, '`:completion::complete:foo`' is more specific than '`:completion::complete:*`'), and longer patterns are preferred over shorter patterns.

As with tags, completion functions can use any style they choose, so there can't be a complete list. However, the following two sections list those tags and styles that are used in many places of the completion system.

19.3.2 Standard Tags

Here are the tags currently used by the completion system. Some of them are only used when looking up styles and do not refer to a particular type of match.

`accounts` used to look up the `users-hosts` style

`all-expansions`
: used by the `_expand` completer when adding the single string containing all possible expansions

`all-files`
: for the names of all files (as distinct from a particular subset, see the `globbed-files` tag).

`arguments`
: when an argument of a command may be completed

`arrays`
: for names of array parameters

`association-keys`
: for keys of associative arrays; used when completing inside a subscript of a parameter of this type

`bookmarks`
: when completing bookmarks (e.g. for URLs and the `zftp` function suite)

`builtins`
: for names of builtin commands

`characters`
: used for commands like `stty` when completing characters; also used when completing character classes after a opening bracket

`colormapids`
: for X colormap ids

`colors`
: for color names

`commands`
: for names of external commands and names of sub-commands (used by some commands like `cvs`)

`contexts`
: for contexts used by the zstyle builtin command

`corrections`
: used by the `_approximate` and `_correct` completers for the possible corrections

`cursors`
: for cursor names used by X programs

`default`
: used to look up default values for various styles that may also be set for tags that are used when generating matches; note that this tag is used when only the *function* field of the context name is set up

`descriptions`
: used when looking up the value of the `format` style for descriptions

`devices`
: for names of device special files

`directories`
: for names of directories

`directory-stack`
: for entries in the directory stack

`displays`
: for X display names

`domains`
: for network domains

`expansions`
: used by the `_expand` completer for individual possibilities resulting from expansion of a word

`extensions`
: for X server extensions

`file-descriptors`
: for the numbers of open file descriptors

`files` the generic file-matching tag used by completion functions that can complete the names of some kind of file

`fonts` used for X font names

`functions`
names of functions, normally shell functions although certain commands may understand other kinds of function

`globbed-files`
for names of files matching the glob pattern used by completion functions that expect a certain type of file

`groups` used when completing names of user groups

`history-words`
for words from the history

`hosts` for hostnames

`indexes` used for array indexes

`jobs` used for jobs

`keymaps` for names of zsh keymaps

`keysyms` for names of X keysyms

`libraries`
for names of system libraries

`limits` for system limits

`local-directories`
for names of directories which are subdirectories of the current working directory when completing for the `cd` and related builtin commands

`manuals` for names of manual pages

`maps` for map names (e.g. NIS maps)

`messages` used to look up the `format` style for messages

`modifiers`
for names of X modifiers

`modules` for modules (e.g. `zsh` modules)

`my-accounts`
used to look up the `users-hosts` style

`named-directories`
for named directories (you wouldn't have guessed that, would you?)

`names` for all kinds of names

`nicknames`
for nicknames of NIS maps

`options` for command options

`original` used by the `_approximate`, `_correct` and `_expand` completers when adding the original string

`other-accounts`
used to look up the `users-hosts` style

`packages` for packages (e.g. `rpm` or installed `Debian` packages)

`parameters`
for names of parameters

`path-directories`
for names of directories found by searching the `cdpath` array when completing for the `cd` and related builtin commands

`paths` used to look up the values of the `expand`, `ambiguous` and `special-dirs` styles

`pods` for perl pods (documentation files)

`ports` for communication ports

`prefixes` for prefixes (like those of a URL)

`printers` for printer names

`processes`
for process identifiers

`processes-names`
used to look up the `command` style when generating the names of processes for `killall`

`sequences`
for sequences (e.g. `mh` sequences)

`sessions` for sessions in the `zftp` function suite

`signals` for signal names

`strings` for strings (e.g. the replacement strings for the `cd` builtin command)

`styles` for styles used by the zstyle builtin command

`tags` for tags (e.g. `rpm` tags)

`targets` for makefile targets

`types` for types of whatever (e.g. address types for the `xhost` command)

`urls` used to look up the `urls` and `local` styles when completing URLs

`users` for usernames

`values` when completing a value out of a set of values (or a list of such values)

`version` used by `_call_program` to look up the command to run to determine the installed version of various other commands (such as `diff` and `make`).

`warnings` used to look up the `format` style for warnings

`widgets` for zsh widget names

`windows` for IDs of X windows

`zsh-options`
for shell options

19.3.3 Standard Styles

Here are the names of the styles used by the completion system. Note that the values of several of these styles represent boolean values; here, any of the strings '`true`', '`on`', '`yes`', and '`1`' can be used for the truth value 'true' and the strings '`false`', '`off`', '`no`', and '`0`' are interpreted as 'false'. The behavior for any other value is undefined unless the description for the particular style mentions other possible values; in particular, the default value may be either on or off if the style is not set.

Some of these styles are tested for every tag used to add possible matches and for the `default` tag (most notably `menu`, `list-colors` and the styles controlling the completion listing like `list-packed` and `last-prompt`). When tested for the `default` tag, only the *function* field of the context will be set up, so the default value will normally be set like:

```
zstyle ':completion:*:default' menu ...
```

`accept-exact`
: This is tested for the default tag and the tags used when generating matches. If it is set to 'true' for at least one match which is the same as the string on the line, this match will immediately be accepted.

  When completing pathnames (where it is looked up for the `paths` tag), this style also accepts any number of patterns as the value. If this is used, pathnames matching one of these patterns will be accepted immediately even if the command line contains some more partially typed pathname components and these match no file under the directory accepted.

  Note that this is also used by the `_expand` completer to decide if words beginning with a tilde or parameter expansion should be expanded. This means that if, for example, there are parameters `foo` and `foobar`, the string '`$foo`' will only be expanded if `accept-exact` is set to 'true'.

`add-space`
: This style is used by the `_expand` completer. If it is 'true' (the default), a space will be inserted after all words resulting from the expansion (except for directory names which get a slash). The value may also be the string '`file`' to make the completer add a space only to names of existing files. Finally, the 'true' values and '`file`' may be combined with '`subst`' to keep the completer from adding a space when the resulting words were generated by expanding a substitution of the form '`$(...)`' or '`${...}`'.

  It is also used by the `_prefix` completer as a simple boolean value to decide if a space should be inserted before the suffix.

`ambiguous`
: This applies when completing non-final components of filename paths. If it is set, the cursor is left after the first ambiguous component, even if menu completion is in use. It is tested with the `paths` tag.

`assign-list`
: When completing after an equals sign, the completion system normally completes only one filename. In some cases, particularly for certain parameters such as `PATH`, a list of filenames separated by colons is required. This style can be set to a list of patterns matching the names of such parameters.

  The default is to complete lists when the word on the line already contains a colon.

`auto-description`
: If set, this style's value will be used as the description for options which are not described by the completion functions, but that have exactly one argument. The

sequence '%d' in the value will be replaced by the description for this argument. Depending on personal preferences, it may be useful to set this style to something like 'specify: %d'. Note that this may not work for some commands.

avoid-completer
: This is used by the _all_matches completer to decide if the string consisting of all matches should be added to the list currently being generated. Its value is a list of names of completers. If any of these is the name of the completer that generated the matches in this completion, the string will not be added.

  The default value for this style is '_expand _old_list _correct _approximate', i.e. it contains the completers for which a string with all matches will almost never be wanted.

cache-path
: This style defines the path where any cache files containing dumped completion data are stored. Defaults to '$ZDOTDIR/.zcompcache', or '$HOME/.zcompcache' if $ZDOTDIR is not defined. The completion layer will not be used unless the use-cache style is set.

call-command
: Currently this is only used by the function completing make targets. If it is set to 'true' and the installed version of the make command allows it, make is called in a way to generate all possible targets. The default value of this style is 'false' because calling make can potentially take a very long time and in some cases may even cause actions from the makefile be executed despite the options given to make.

command
: In many places, completion functions need to call external commands to generate the list of completions. This style can be used to override the command which is called in some such cases. The elements of the value are joined with spaces to form a command line to execute. The value can also start with a hyphen, in which case the usual command will be added to the end; this is most useful for putting 'builtin' or 'command' in front to make sure the appropriate version of a command is called, for example to avoid calling a shell function with the same name as an external command.

  As an example, the function generating process IDs as matches uses this style with the processes tag to generate the IDs to complete and the list of processes to display (if the verbose style is 'true'). The list produced by the command should look like the output of the ps command. The first line is not displayed, but is searched for the string 'PID' (or 'pid') to find the position of the process IDs in the following lines. If the line does not contain 'PID', the first numbers in each of the other lines are taken as the process IDs to complete.

  Note that the completion function generally has to call the command every time it is called. Because of that care should be taken to specify only commands that take a short time to run (and that will eventually stop at all).

commands
: This is used by the function completing sub-commands for the system initialisation scripts (residing in /etc/init.d or somewhere not too far away from that). It's values give the default commands to complete for those commands for which the completion function isn't able to find them out automatically. The default for this style are the two strings 'start' and 'stop'.

complete
: This is used by the _expand_alias function when invoked as a bindable command. If it set to 'true' and the word on the command line is not the name of an alias, matching alias names will be completed.

`completer`
: The strings given as the value of this style provide the names of the completer functions to use. The available completer functions are described in Section 19.4 [Control Functions], page 154.

  Each string may be the name of a completer function or a string of the form '*function*`:`*name*'. In the first case the *completer* field of the context will contain the name of the completer without the leading underscore and with all other underscores replaced by hyphens. In the second case the *function* is the name of the completer to call, but the context will contain the *name* in the *completer* field of the context. If the *name* starts with a hyphen, the string for the context will be build from the name of the completer function as in the first case with the *name* appended to it. For example:

```
zstyle ':completion:*' completer _complete _complete:-foo
```

  Here, completion will call the `_complete` completer twice, once using '`complete`' and once using '`complete-foo`' in the *completer* field of the context. Normally, using the same completer more than once makes only sense when used with the '*functions*`:`*name*' form, because otherwise the context name will be the same in all calls to the completer; possible exceptions to this rule are the `_ignored` and `_prefix` completers.

  The default value for this style is `_complete _ignored`, i.e. normally only completion will be done, first using the `ignored-patterns` style and the `$fignore` array and then without ignoring matches.

`condition`
: This style is used by the `_list` completer function to decide if insertion of matches should be delayed unconditionally. The default is 'true'.

`disabled`
: If this is set to 'true', the `_expand_alias` completer and bindable command will try to expand disabled aliases, too. The default is '`false`'.

`disable-stat`
: This is used with an empty tag by the function completing for the `cvs` command to decide if the `zsh/stat` module should be used to generate names of modified files in the appropriate places (this is its only use). If set, completion will use the `ls` command.

`domains`
: If set, gives the names of network domains that should be completed. If this is not set by the user domain names will be taken from the file `/etc/resolv.conf`.

`expand`
: This style is used when completing strings consisting of multiple parts, such as path names. If one of its values is the string '`prefix`', the partially typed word from the line will be expanded as far as possible even if trailing parts cannot be completed. If one of its values is the string '`suffix`', matching names for components after the first ambiguous one will also be added. This means that the resulting string is the longest unambiguous string possible, but if menu completion is started on the list of matches generated this way, this will also cycle through the names of the files in pathname components after the first ambiguous one.

`fake-files`
: This style is used when completing files and looked up without a tag. Its values are of the form '*dir*`:`*names...*'. This will add the *names* (strings separated by spaces) as possible matches when completing in the directory *dir*, even if no such files really exist.

  This can be useful on systems that support special filesystems whose top-level pathnames can not be listed or generated with glob patterns. It can also be used for directories for which one does not have read permission.

`fake-parameters`

This is used by the completion function generating parameter names as matches. Its values are names of parameters which might not yet be set, but which should be completed nonetheless. Each name may also be followed by a colon and a string specifying the type of the parameter (like '`scalar`', '`array`' or '`integer`'). If such a type is given, the name will only be completed if parameters of that type are requested in the particular context. Names for which no type is specified will always be completed.

`file-patterns`

In most places where filenames are completed, the function `_files` is used which can be configured with this style. If the style is unset, `_files` offers, one after another, up to three tags: '`globbed-files`', '`directories`' and '`all-files`', depending on the types of files expected by the caller of `_files`.

If the `file-patterns` style is set, the default tags are not used. Instead, the value of the style says which tags and which patterns are to be offered. The strings in the value contain specifications of the form '*pattern*`:`*tag*'; each string may contain any number of such specifications. The *pattern* gives a glob pattern that is to be used to generate filenames. If it contains the sequence '`%p`', that is replaced by the pattern(s) given by the calling function. Colons in the pattern must be preceded by a backslash to make them distinguishable from the colon before the *tag*. If more than one pattern is needed, the patterns can be given inside braces, separated by commas. The *tag*s of all strings in the value will be offered by `_files` (again, one after another) and used when looking up other styles. For strings containing more than one specification, the filenames for all specifications will be generated at the same try. If no '`:`*tag*' is given the '`files`' tag will be used. The *tag* may also be followed by an optional second colon and a description. If that is given, this description will be used for the '`%d`' in the value of the `format` style (if that is set) instead of the default description supplied by the completion function. If the description given here contains itself a '`%d`', that is replaced with the description supplied by the completion function.

For example, to make the `rm` command first complete only names of object files and the names of all files if no object file matches the string on the line, one would do:

```
zstyle ':completion:*:*:rm:*' file-patterns \
    '*.o:object-files' '%p:all-files'
```

Another interesting example is to change the default behaviour that makes completion first offer files matching the patterns given by the calling function, then directories and then all files. Many people prefer to get both the files matching the given patterns and the directories in the first try and all files at the second try. To achieve this, one could do:

```
zstyle ':completion:*' file-patterns \
    '%p:globbed-files *(-/):directories' '*:all-files'
```

This works even for contexts in which all files would be completed, because `_files` will not try a pattern more than once and it stops when the pattern '`*`' was tried.

Note also that during the execution of completion functions, the `EXTENDED_GLOB` option is in effect, so the characters '`#`', '`~`' and '`^`' have special meanings in the patterns.

`file-sort`

The completion function that generates filenames as possible matches uses this style without a tag to determine in which order the names should be listed and completed when using menu completion. The value may be one of '`size`' to sort them by the size of the file, '`links`' to sort them by the number of links to the file, '`modification`'

(or '`time`' or '`date`') to sort them by the last modification time, '`access`' to sort them by the last access time, or '`inode`' (or '`change`') to sort them by the last inode change time. If the style is set to any other value, or is unset, files will be sorted alphabetically by name. If the value contains the string '`reverse`', sorting is done in decreasing order.

`force-list`
: This forces a list of completions to be shown at any point where listing is done, even in cases where the list would usually be suppressed. For example, normally the list is only shown if there are at least two different matches. By setting this style to '`always`', the list will always be shown, even if there is only a single match which is immediately accepted. The style may also be set to a number. In this case the list will be shown if there are at least that many matches, even if they would all insert the same string.

  This style is tested for the default tag and all tags used when generating matches. This allows one to turn unconditional listing on for certain types of matches.

`format`
: If this is set for the `descriptions` tag, its value is used as a string to display above matches in completion lists. The sequence '`%d`' in this string will be replaced with a short description of what these matches are. This string may also contain the sequences to specify output attributes, such as '`%B`', '`%S`' and '`%{...%}`'.

  For the same purpose, this style is also tested with the tags used when matches are generated before it is tested for the `descriptions` tag. This provides the possibility of defining different format strings for different types of matches.

  Note also that some completer functions define additional '`%`'-sequences. These are described for the completer functions that make use of them.

  For the `messages` tag, this style defines a string used by some completion functions to display messages. Here, the '`%d`' is replaced with a message given by the completion function.

  Finally, when set with the `warnings` tag, the format string is printed when no matches could be generated at all. In this case the '`%d`' is replaced with the descriptions for the matches that were expected separated by spaces and the sequence '`%D`' is replaced with those descriptions separated by newlines.

  The '`%`' for the sequences that are replaced by strings provided by the completion functions like the '`%d`' may be followed by field width specifications as described for the `zformat` builtin command from the `zsh/zutil` module, see Section 21.23 [The zsh/zutil Module], page 210.

`glob`
: This is used by the `_expand` completer. If it is set to 'true' (the default), globbing will be attempted on the words resulting from substitution (see the `substitute` style) or the original string from the line.

`global`
: If this is set to 'true' (the default), the `_expand_alias` completer and bindable command will try to expand global aliases.

`group-name`
: The completion system can put different types of matches in different groups which are then displayed separately in the list of possible completions. This style can be used to give the names for these groups for particular tags. For example, in command position the completion system generates names of builtin and external commands, names of aliases, shell functions and parameters and reserved words as possible completions. To have the external commands and shell functions listed separately, one can set:

```
zstyle ':completion:*:*:-command-:*:commands' group-name commands
zstyle ':completion:*:*:-command-:*:functions' group-name functions
```

This also means that if the same name is used for different types of matches, then those matches will be displayed together in the same group.

If the name given is the empty string, then the name of the tag for the matches will be used as the name of the group. So, to have all different types of matches displayed separately, one can just set:

```
zstyle ':completion:*' group-name ''
```

All matches for which no group name is defined will be put in a group named `-default-`.

`group-order`
: This style is to be used together with the `group-name` style. Once different types of matches are put into different groups, this style can be used to define in which order these groups should appear when listing (compare `tag-order`, which determines which completions appear at all). The strings in the value are taken as group names and the named groups will be shown in the order in which their names appear in the value. All groups whose names are not given in the value of this style will appear in the order defined by the function generating the matches.

  For example, to have names of builtin commands, shell functions and external commands appear in this order when completing in command position one would set:

```
zstyle ':completion:*:*:-command-:*' group-order \
       builtins functions commands
```

`groups`
: A style holding the names of the groups that should be completed. If this is not set by the user, the group names from the YP database or the file '`/etc/group`' will be used.

`hidden`
: If this is set to one of the 'true' values, the matches for the tags for which this is set will not appear in the list; only the description for the matches as set with the `format` style will be shown. If this is set to '`all`', not even the description will be displayed.

  Note that the matches will still be completed; they are just not shown in the list. To avoid having matches considered as possible completions at all, the `tag-order` style can be modified as described below.

`hosts`
: A style holding the names of hosts that should be completed. If this is not set by the user the hostnames in '`/etc/hosts`' will be used.

`hosts-ports`
: This style is used by commands that need or accept hostnames and ports. The strings in the value should be of the form '*host*`:`*port*'. These hostnames and ports are completed depending on the information already on the line, so that if, for example, the hostname is already typed, only those ports specified for that host will be completed. Multiple ports for the same host may appear.

`ignore-line`
: This style is tested for the tags used when generating matches. If it is set to 'true', then none of the words that are already on the line will be considered possible completions. If it is set to '`current`', the word the cursor is on will not be considered a possible completion. The same happens if the value is '`current-shown`', but only if the list of completions is currently shown on the screen. Finally, if it is set to '`other`' all words except the current one will not be considered to be a possible completion.

  The values '`current`' and '`current-shown`' are a bit like the opposite of `accept-exact`. They mean that only strings with missing characters will be completed.

  Note that you almost certainly don't want to set this to 'true' or '`other`' for a general context such as '`:completion:*`'. This is because it would disallow completion of,

for example, options multiple times even if the command in question accepts the option more than once.

`ignore-parents`
: The style is tested by the function completing pathnames without a tag to determine whether to ignore the names of directories already mentioned in the current word, or the name of the current working directory. The value must include one or both of the following strings:

  `parent` The name of any directory whose path is already contained in the word on the line is ignored. For example, when completing after `foo/../`, the directory `foo` will not be considered a valid completion.

  `pwd` The name of the current working directory will not be completed, so that, for example, completion after `../` will not use the name of the current directory.

  In addition, the value may include one or both of:

  `..` Ignore the specified directories only when the word on the line contains the substring '`../`'.

  `directory` Ignore only when names of directories are completed, not when completing names of files.

  Note that names of directories ignored because of one of the tests will be ignored in the same way as the matches ignored because of the `ignored-patterns` style. I.e., by using the `_ignored` completer it is possible to complete these directories nonetheless.

`ignored-patterns`
: This style can be used to specify a list of patterns which are tested against against the trial completions in a given context; any matching completions will be removed from the list of possibilities. The `_ignored` completer can appear in the list of completers to produce a list which includes these matches once more. This is a more configurable version of the shell parameter `$fignore`.

  Note that during the execution of completion functions, the `EXTENDED_GLOB` option is in effect, so the characters '`#`', '`~`' and '`^`' have special meanings in the patterns.

`insert-ids`
: When completing process IDs, for example as arguments to the `kill` and `wait` builtins, completion allows the user to type the name of a command, which will be converted to the appropriate process ID. A problem arises when the process name typed is not unique. By default (or if this style is set explicitly to '`menu`') the name will be converted immediately to a set of possible IDs, and menu completion will be started to cycle through them. If the value of the style is '`single`', however, the shell will wait until the user has typed enough to make the command unique before converting the name to an ID; the user must type any additional characters required. If the value is any other string, menu completion will be started when the string typed by the user is longer than the common prefix of the corresponding IDs.

`insert-tab`
: If this has one of the 'true' values, the completion system will insert a TAB character (assuming it was used to start completion) instead of performing completion when there is no non-blank character to the left of the cursor. If set to 'false', completion will be done even there.

  The value may also contain the substrings '`pending`' or '`pending=`*val*' to make the character typed to start completion be inserted instead of completion being tried

when there is input pending which has not yet been processed by the shell. If a *val* is given, completion will not be done if there are at least that many characters of unprocessed input. This is often useful to have set when pasting characters into a terminal. Note however, that it relies on the `$PENDING` special parameter from the `zsh/zle` module being set properly which is not guaranteed on all platforms.

The default value of this style is 'true' unless when completing inside the `vared` builtin command, where it defaults to 'false'.

`insert-unambiguous`
: This is used by the `_match` and `_approximate` completer functions, where the possible completions may not have a common prefix so that menu completion is often the most useful may of choosing completions. If the style is set to 'true', the completer will start menu completion only if no unambiguous string could be generated that is at least as long as the original string typed by the user. Note that the `_approximate` completer uses it after setting the completer field in the context name to one of `correct-`*num* or `approximate-`*num*, where *num* is the number of errors that were accepted.

  When used for the `_match` completer, the style may also be set to the string '`pattern`'. This makes the pattern on the line be left unchanged if it didn't match unambiguously.

`keep-prefix`
: This style is used by the `_expand` completer. If it is 'true', the completer will try to keep a prefix containing a tilde or parameter expansion. I.e., the string '`~/f*`' would be expanded to '`~/foo`' instead of '`/home/user/foo`'. If the style is set to '`changed`' (the default), the prefix will only be left unchanged if there were other changes between the expanded words and the original word from the command line. Any other value makes the prefix be expanded unconditionally.

  Note that with one of the 'true' values, the `_expand` completer returns if there is only one expansion and that is, after restoring the original prefix, the same as the original word. This means that other completers will be called immediately after `_expand`.

`last-prompt`
: This is used to determine if the completion code should try to put the cursor back onto the previous command line after showing a completion listing (as for the `ALWAYS_LAST_PROMPT` option). As with several other styles, it is tested for the `default` tag as well as all the possible tags when generating matches. The cursor will be moved back to the previous line if this style is 'true' for all types of matches added. Note also that this is independent of the numeric argument, unlike the `ALWAYS_LAST_PROMPT` option.

`list`
: This style is used by the `_history_complete_word` bindable command. If it is set to 'true' it has no effect, but if it is set to 'false' the matches will not be listed, overriding the setting of the options that control listing behaviour, especially `AUTO_LIST`. Use the context prefix '`:completion:history-words`'.

`list-colors`
: If the `zsh/complist` module is used, this style can be used to set color specifications as with the `ZLS_COLORS` and `ZLS_COLOURS` parameters, which will not be honored under this completion system (see Section 21.6 [The zsh/complist Module], page 189).

  If this style is set for the `default` tag, the strings in the value are taken as specifications that are to be used everywhere. If it is set for other tags, the specifications are used only for matches of the type described by the tag. For this to work best, the `group-name` style must be set to an empty string. If the `group-name` tag specifies

other names for the groups the matches in these groups can be colored by using these names together with the '`(group)...`' syntax described for the `ZLS_COLORS` and `ZLS_COLOURS` parameters and adding the specifications to the value for this style with the `default` tag (although in most cases it should work by setting this style for the appropriate tags).

It is possible to use the same specifications set up for the GNU version of the `ls` command:

```
zstyle ':completion:*:default' list-colors ${(s.:.)LS_COLORS}
```

The default colors are the same as for the GNU `ls` command and can be obtained by setting the style to an empty string (i.e. `''`).

`list-packed`
: Like the `list-colors` style, this is tested with the `default` tag and all tags used when generating matches. If it is set to 'true' for a tag, the matches added for it will be listed as if the `LIST_PACKED` option were set. If it is set to 'false', they are listed normally.

`list-prompt`
: If this style is set for the `default` tag, completion lists that don't fit on the screen can be scrolled (see Section 21.6 [The zsh/complist Module], page 189). The value, if not the empty string, will be displayed after every screenful and the shell will prompt for a key press; if the style is set to the empty string, a default prompt will be used. The value may contain the escape sequences '`%l`' or '`%L`', which will be replaced by the number of the last line displayed and the total number of lines; '`%m`' or '`%M`', which will be replaced by the number of the last match shown and the total number of matches; and '`%p`' and '`%P`', which will be replaced by '`Top`' when at the beginning of the list, '`Bottom`' when at the end and the position shown in percent of the total length otherwise. In each of these cases the form with the uppercase letter is replaced by a string of fixed width, padded to the right with spaces. As in other prompt strings, the escape sequences '`%S`', '`%s`', '`%B`', '`%b`', '`%U`', '`%u`', and '`%{...%}`' for entering and leaving the display modes standout, bold and underline are also available.

`list-rows-first`
: This style is tested in the same way as the `list-packed` style and determines if matches are to be listed in a rows-first fashion, as for the `LIST_ROWS_FIRST` option.

`list-suffixes`
: This style is used by the function used to complete filenames. If completion is attempted on a string containing multiple partially typed pathname components and this style is set to 'true', all components starting with the first one for which more than one match could be generated will be shown.

`local`
: This style is used by completion functions which generate URLs as possible matches to add suitable matches when a URL points to a local web server, that is, one whose files are available directly on the local file system. Its value should consist of three strings: a hostname, the path to the default web pages for the server and the directory name used by a user placing web pages within their home area. For example, completion after '`http://toast/~yousir/`' will attempt to match the name '`toast`' against the first argument to the style, and if successful will look in the directory under `~yousir` given by the third argument to the style for possible completions.

`match-original`
: This is used by the `_match` completer. If it is set to `only`, `_match` will try to generate matches without inserting a '`*`' at the cursor position. If set to any other non-empty

value, it will first try to generate matches without inserting the '*' and if that yields no matches, it will try again with the '*' inserted. If it is unset or set to the empty string, matching will only be done with the '*' inserted.

`matcher` This style is tested for tags used when generating matches. Its value is used as an match specification additional to any given by the `matcher-list` style which should be in the form described in Section 18.5 [Matching Control], page 124.

`matcher-list`
This style is used by the main completion function to retrieve match specifications that are to be used everywhere. Its value should be a list of such specifications. The completion system will try them one after another for each completer selected. For example, to first try simple completion and, if that generates no matches, case-insensitive completion one would do:

```
zstyle ':completion:*' matcher-list '' 'm:{a-zA-Z}={A-Za-z}'
```

By default every specification replaces previous ones. If specification is prefixed with `+`, it is added to the existing list. This allows testing more general patterns without repeating the whole list every time, as in:

```
zstyle ':completion:*' matcher-list '' '+m{a-Z}={A-Z}' '+m{A-Z}={a-
z}'
```

The style allows even finer control by specifying a particular completer, without the leading underscore, in the third field of the completion context. For example, if one uses the completers `_complete` and `_prefix` but wants to try case-insensitive completion only when using the `_complete` completer, one would do:

```
zstyle ':completion:*' completer _complete _prefix
zstyle ':completion:*:complete:*' matcher-list \
       '' 'm:{a-zA-Z}={A-Za-z}'
```

Note that the `completer` style allows user-defined names to be used in the context instead of the name of the completer. This is useful if, for example, one wants to try normal completion without a match specification and with case-insensitive matching first, correction if that doesn't generate any matches and partial-word completion if that doesn't yield any matches either. In this case one can give the `_complete` completer more than once in the `completer` style and define different match specifications for each occurrence, as in:

```
zstyle ':completion:*' completer _complete _correct _complete:foo
zstyle ':completion:*:complete:*' matcher-list \
    '' 'm:{a-zA-Z}={A-Za-z}'
zstyle ':completion:*:foo:*' matcher-list \
    'm:{a-zA-Z}={A-Za-z} r:|[-_./]=* r:|=*'
```

If the style is unset in any context no match specification is applied; further, some completers such as `_correct` and `_approximate` do not use the match specifications at all. However, it is always safe to use the simple form for this style (as in the first example above), since any completers which do not use match specifications will only ever be called once, rather than once per specification.

Since the specification-strings in this style have to be tried one after another, it is a good idea to keep their number low. In most cases one to three strings (each of which may, without to large a performance hit, consist of more than one single match specification) will give acceptable performance.

`max-errors`
This is used by the `_approximate` and `_correct` completer functions to determine the maximum number of errors to allow. The completer will try to generate completions by first allowing one error, then two errors, and so on, until either a match

or matches were found or the maximum number of errors given by this style has been reached.

If the value for this style contains the string '`numeric`', the completer function will take any numeric argument as the maximum number of errors allowed. For example, with

```
zstyle ':completion:*:approximate:::' max-errors 2 numeric
```

two errors are allowed if no numeric argument is given, but with a numeric argument of six (as in '`ESC-6 TAB`'), up to six errors are accepted. Hence with a value of '`0 numeric`', no correcting completion will be attempted unless a numeric argument is given.

If the value contains the string '`not-numeric`', the completer will *not* try to generate corrected completions when given a numeric argument, so in this case the number given should be greater than zero. For example, '`2 not-numeric`' specifies that correcting completion with two errors will usually be performed, but if a numeric argument is given, correcting completion will not be performed.

The default value for this style is '`2 numeric`'.

`menu` If this is set to true in a given context, using any of the tags defined for a given completion, menu completion will be used. The tag '`default`' can be used to set the default value, but a specific tag will take precedence. If none of the values found in this way is true but at least one is set to '`auto`' the behaviour will be as for the `AUTO_MENU` option. Finally, if one of the values is explicitly set to false, menu completion will be turned off even if it would otherwise be active (for example, with the `MENU_COMPLETE` option).

Using the form '`yes=`*num*', where '`yes`' may be any of the true values ('`yes`', '`true`', '`on`' and '`1`') turns on menu completion if there at least *num* matches. Using this for one of the 'false' values (as in '`no=10`') makes menu completion *not* be used if there are *num* or more matches. Of course, this is only useful when menu completion is normally used, e.g. by setting the `MENU_COMPLETE` option. The 'true' values may also be used in the form '`yes=long`' to turn on menu completion if the list does not fit onto the screen. This will start menu completion only if normal completion was attempted, not when only the list of possible completions was requested. To start menu completion even then, the value '`yes=long-list`' can be used.

In addition to (or instead of) the above possibilities, the value may contain the string '`select`', optionally followed by an equals sign and a number. In this case menu selection (as defined by the `zsh/complist` module) will be started. Without the optional number, it will be started unconditionally and with a number it will be started only if at least that many matches are generated; if the values for more than one tag provide a number, the smallest number is taken. Menu selection can be turned off explicitly by defining a value containing the string '`no-select`'.

It is also possible to start menu selection only if the list of matches does not fit on the screen by using the value '`select=long`'. This will only start menu selection if the widget invoked does completion, not simply listing as done by `delete-char-or-list`; to start menu selection even here, use the value '`select=long-list`'.

To turn on menu completion or menu selection when a certain number of matches is generated *or* the list of matches does not fit onto the screen, both of '`yes=`' and '`select=`' can be given twice, once with a number and once with '`long`' or '`long-list`'.

`numbers` This is used with the `jobs` tag. If it is 'true', the shell will complete the job numbers instead of the shortest unambiguous strings of the jobs' command lines. If the value is a number, job numbers will only be used if that many words from the

job descriptions are required to resolve ambiguities. For example, if the value is '1', strings will only be used if all jobs differ in the first word on their command lines.

`old-list` This is used by the `_oldlist` completer. If it is set to '`always`', then standard widgets which perform listing will retain the current list of matches, however they were generated; this can be turned off explicitly with the value '`never`', giving the behaviour without the `_oldlist` completer. If the style is unset, or any other value, then the existing list of completions is displayed if it is not already; otherwise, the standard completion list is generated; this is the default behaviour of `_oldlist`. However, if there is an old list and this style contains the name of the completer function that generated the list, then the old list will be used even if it was generated by a widget which does not do listing.

For example, suppose you type `^Xc` to use the `_correct_word` widget, which generates a list of corrections for the word under the cursor. Usually, typing `^D` would generate a standard list of completions for the word on the command line, and show that. With `_oldlist`, it will instead show the list of corrections already generated.

As another example consider the `_match` completer: with the `insert-unambiguous` style set to 'true' it inserts only a common prefix string, if there is any. However, this may remove parts of the original pattern, so that further completion could produce more matches than on the first attempt. By using the `_oldlist` completer and setting this style to `_match`, the list of matches generated on the first attempt will be used again.

`old-matches`
This is used by the `_all_matches` completer to decide if an old list of matches should be used if one exists. It may be set to one of the 'true' values or to the string '`only`' to use such a list. If it is set to '`only`', `_all_matches` will only use an old list and won't have any effect on the list of matches currently being generated.

`old-menu` This is used by the `_oldlist` completer. It controls how menu completion behaves when a completion has already been inserted and the user types a standard completion key type such as `TAB`. The default behaviour of `_oldlist` is that menu completion always continues with the existing list of completions. If this style is set to 'false', however, a new completion is started if the old list was generated by a different completion command; this is the behaviour without the `_oldlist` completer.

For example, suppose you type `^Xc` to generate a list of corrections, and menu completion is started in one of the usual ways. Usually, or with this style set to `false`, typing `TAB` at this point would start trying to complete the line as it now appears. With `_oldlist`, it instead continues to cycle through the list of corrections.

`original` This is used by the `_approximate` and `_correct` completers to decide if the original string should be added as one possible completion. Normally, this is done only if there are at least two possible corrections, but if this style is set to 'true', it is always added. Note that these completers use this style after setting the completer field in the context name to `correct-`*num* or `approximate-`*num*, where *num* is the number of errors that were accepted.

`packageset`
This style is used when completing arguments of the Debian '`dpkg`' program. It contains an override for the default package set for a given context. For example,

```
zstyle ':completion:*:complete:dpkg:option--status-1:*' \
                    packageset avail
```

causes available packages, rather than only installed packages, to be completed for 'dpkg –status'.

`path` The function that completes color names uses this style with the `colors` tag. The value should be the pathname of a file containing color names in the format of an X11 `rgb.txt` file. If the style is not set but this file is found in one of various standard locations it will be used as the default.

`ports` A style holding the service names of ports to complete. If this is not set by the user, the service names from '`/etc/services`' will be used.

`prefix-hidden`
This is used when matches with a common prefix are added (e.g. option names). If it is 'true', this prefix will not be shown in the list of matches.

The default value for this style is 'false'.

`prefix-needed`
This, too, is used for matches with a common prefix. If it is set to 'true' this common prefix has to be typed by the user to generate the matches. E.g. for options this means that the '`-`', '`+`', or '`--`' has to be on the line to make option names be completed at all.

The default style for this style is 'true'.

`preserve-prefix`
This style is used when completing path names. Its value should be a pattern matching an initial prefix of the word to complete that should be left unchanged under all circumstances. For example, on some Unices an initial '`//`' (double slash) has a special meaning and hence should be kept. For that one could set this style to the string '`//`'. As another example, setting this style to '`?:/`' under Cygwin would allow completion after '`a:/...`' and the like.

`range` This is used by the `_history` completer and the `_history_complete_word` bindable command to decide which words should be completed. It may be set to a number, *N*, to say that only the last *N* words from the history should be completed. The value may also be of the form '*max*`:`*slice*'. This means that first the last *slice* words will be completed. If that yields no matches, the *slice* words before those will be tried and so on, until either at least one match is generated or *max* words have been tried. The default is to complete all words from the history at once.

`regular` This style is used by the `_expand_alias` completer and bindable command. If set to 'true' (the default), regular aliases will be expanded but only in command position. If it is set to 'false', regular aliases will never be expanded and if it is set to the string '`always`', regular aliases will be expanded even if not in command position.

`remove-all-dups`
The `_history_complete_word` bindable command and the `_history` completer use this to decide if all duplicate matches should be removed, rather than just consecutive duplicates.

`select-prompt`
If this is set for the `default` tag, its value will be displayed during menu selection (see the `menu` style above) when the completion list does not fit on the screen as a whole. The same escapes as for the `list-prompt` style are understood, but give the number of the match or line the mark is on. A default prompt is used when the value is the empty string.

`select-scroll`
This style is tested for the `default` tag and determines how a completion list is scrolled during a menu selection (see the `menu` style above) when the completion list does not fit on the screen as a whole. Its value should be '`0`' (zero) to scroll by half-screenfuls, a positive integer to scroll by that many lines and a negative

number to scroll by the number of lines of the screen minus that number (or plus the number, since it is negative). The default is to scroll by single lines.

`single-ignored`
: This is used by the `_ignored` completer. It specifies what should be done if it can generate only one match, which is often a special case. If its value is '`show`', the single match will be displayed but not inserted. If the value is '`menu`', then the single match and the original string are both added as matches and menu completion is started so that one can easily select either of them.

`sort`
: If set to 'true', completion functions that generate words from the history as possible matches sort these words alphabetically instead of keeping them in the order in which they appear in the history (from youngest to oldest).

  This is also used by the `_expand` completer. Here, if it is set to 'true', the expansions generated will always be sorted. If it is set to '`menu`', then the expansions are only sorted when they are offered as single strings (not in the string containing all possible expansions).

`special-dirs`
: Normally, the completion code will not produce the directory names '`.`' and '`..`' as possible completions. If this style is set to 'true', it will add both '`.`' and '`..`' as possible completions; if it is set to '`..`', only '`..`' will be added.

`squeeze-slashes`
: If set to 'true', sequences of slashes (as in '`foo//bar`') will be treated as if they were only one slash when completing pathnames. This is the usual behaviour of UNIX paths. However, by default the file completion function behaves as if there were a '`*`' between the slashes.

`stop`
: If set to 'true', the `_history_complete_word` bindable command will stop once when reaching the beginning or end of the history. Invoking `_history_complete_word` will then wrap around to the opposite end of the history. If this style is set to 'false' (the default), `_history_complete_word` will loop immediately as in a menu completion.

`subst-globs-only`
: This is used by the `_expand` completer. If it is set to 'true', the expansion will only be used if it resulted from globbing; hence, if expansions resulted from the use of the `substitute` style described below, but these were not further changed by globbing, the expansions will be rejected.

  The default for this style is 'false'.

`substitute`
: This boolean style controls whether the `_expand` completer will first try to expand all substitutions in the string (such as '`$(...)`' and '`${...}`').

  The default is 'true'.

`suffix`
: This is used by the `_expand` completer if the word starts with a tilde or contains a parameter expansion. If it is set to 'true', the word will only be expanded if it doesn't have a suffix, i.e. if it is something like '`~foo`' or '`$foo`', but not if it is '`~foo/`' or '`$foo/bar`', unless that suffix itself contains characters eligible for expansion. The default for this style is 'true'.

`tag-order`
: This provides a mechanism for sorting how the tags available in a particular context will be used.

  The values for the style are sets of space-separated lists of tags. The tags in each value will be tried at the same time; if no match is found, the next value is used. (See the `file-patterns` style for an exception to this behavior.)

For example:

```
zstyle ':completion:*:complete:-command-:*' tag-order \
    'commands functions'
```

specifies that completion in command position should offer only completions for external commands and shell functions immediately.

In addition to tag names, each string in the value may take one of the following forms:

`-` If any string in the value consists of only a hyphen, then *only* the tags specified by the other strings in the value are generated. Normally all tags not explicitly selected are tried last if the specified tags fail to generate any matches. This means that a value consisting only of a single hyphen turns off completion.

`!` *tags...* A string starting with an exclamation mark specifies names of tags that are *not* to be used. The effect is the same as if all other possible tags for the context had been listed.

tag`:`*label ...*
In strings not starting with an exclamation mark, it is also possible to specify tag labels instead of only tags, where *tag* is one of the tags offered by the completion function for the current context and *label* is a name. For this, the completion function will generate matches in the same way as for the *tag* but it will use the *label* in place of the tag in the context names used to look up styles. If the *label* starts with a hyphen, the *tag* is prepended to the *label* to form the name used for lookup. This can be used to make the completion system try a certain tag more than once, supplying different style settings for each attempt, see below for an example.

The *label* may optionally be followed by a second colon and a description. This description will then be used for the '`%d`' in the value of the `format` style instead of the default description supplied by the completion function. Spaces in the description have to be quoted by preceding them with a backslash and a '`%d`' appearing in the description is replaced with the description given by the completion function.

In each of the cases above, the tag may also be a pattern or more than one pattern inside braces and separated by commas. In this case all of the offered tags matching the pattern(s) will be used except for those that are given explicitly in the same string. There are probably two main uses of this. One is the case where one wants to try one of the tags more than once, setting other styles differently for each try, but still wants to use all the other tags without having to repeat them all. For example, to make completion of function names in command position ignore all the completion functions starting with an underscore the first time completion is tried, one could do:

```
zstyle ':completion:*:*:-command-:*' tag-order \
    'functions:-non-comp *' functions
zstyle ':completion:*:functions-non-comp' ignored-patterns '_*'
```

Here, the completion system will first try all tags offered, but will use the tag label `functions-non-comp` when looking up styles for the function names completed. For this, the `ignored-patterns` style is set to exclude functions starting with an underscore from the set of possible matches. If none of the generated matches match the string on the line, the completion system will use the second value of the `tag-order` style and complete functions names again, but this time using the name

`functions` to look up styles, so that the `ignored-patterns` style is not used and all function names are considered.

Of course, this can also be used to split the matches for one tag into different groups. For example:

```
zstyle ':completion:*' tag-order \
    'options:-long:long\ options
     options:-short:short\ options
     options:-single-letter:single\ letter\ options'


zstyle ':completion:*:options-long' ignored-patterns '[-+](|-|[^-
]*)'
zstyle ':completion:*:options-short' ignored-patterns '--*' '[-+]?'
zstyle ':completion:*:options-single-letter' ignored-patterns '???*'
```

With the `group-names` style set, this makes options beginning with '`--`', options beginning with a single '`-`' or '`+`' but containing multiple characters, and single-letter options be displayed in separate groups with different descriptions.

The second interesting use of patterns is the case where one wants to try multiple match specifications one after another. The `matcher-list` style offers something similar, but it is tested very early in the completion system and hence can't be set for single commands nor for more specific contexts. Here is how to try normal completion without any match specification and, if that generates no matches, try again with case-insensitive matching, restricting the effect to arguments of the command `foo`:

```
zstyle ':completion:*:*:foo:*' tag-order '*' '*:-case'
zstyle ':completion:*-case' matcher 'm:{a-z}={A-Z}'
```

First, all the tags offered when completing after `foo` are tried using the normal tag name. If that generates no matches, the second value of `tag-order` is used, which tries all tags again except that this time each has `-case` appended to its name for lookup of styles. Hence this time the value for the `matcher` style from the second call to `zstyle` in the example is used to make completion case-insensitive.

Using the `-e` option of the `zstyle` builtin command, it is possible to specify conditions saying when certain tags are to be used. For example:

```
zstyle -e '*:-command-:*' tag-order '
    if [[ -n $PREFIX ]]; then
      reply=( )
    else
      reply=( - )
    fi'
```

Makes completion in command position happen only if the string on the line is not empty. This is tested using the `PREFIX` parameter which is special in completion widgets; see Chapter 18 [Completion Widgets], page 113 for a description of these special parameters. Setting `reply` to an empty array ensures that only the default behaviour of trying all tags at once is used and setting it to an array containing only a hyphen disables that default behaviour – thus keeping all tags from being tried.

If no style has been defined for a context, the strings '`(|*-)argument-* (|*-)option-* values`' and '`options`' plus all tags offered by the completion function will be used to provide a sensible default behavior that causes arguments (whether normal command arguments or arguments of options) to be completed before option names for most commands.

`urls` This is used together with the the `urls` tag by completion functions that generate URLs as possible matches. If the value consists of more than one string or if the only string does not name a file or directory, the strings are used as the URLs to complete.

If the value contains only one string and that is the name of a normal file, the URLs are taken from that file (where the URLs may be separated by white space or newlines).

Finally, if the only string in the value names a directory, that should contain sub-directories named after the retrieval methods which occur as the first part of a URL, i.e. '`http`', '`ftp`', '`bookmark`', and so on. These sub-directories should contain files and other sub-directories whose pathnames are possible completions after the initial '`http://`', '`ftp://`', etc. See the description in the file `_urls` in the `User` sub-directory of the completion system for more information.

`use-cache`
If this is set, the completion caching layer is activated for any completions which use it (via the `_store_cache`, `_retrieve_cache`, and `_cache_invalid` functions). The directory containing the cache files can be changed with the `cache-path` style.

`use-compctl`
If this style is set to a string *not* equal to `false`, `0`, `no`, and `off`, the completion system may use any completion specifications defined with the `compctl` builtin command. If the style is unset, this is done only if the `zsh/compctl` module is loaded. The string may also contain the substring '`first`' to make the definition for '`compctl -T`' be used, and the substring '`default`' to make the one for '`compctl -D`' be used.

Note that this is only intended to smooth the transition from `compctl` to the new completion system and may disappear in the future.

Note also that the definitions from `compctl` will only be used if there is no specific completion function for the command in question. For example, while completing arguments to the command `foo`, if this was handled by a command function `_foo`, `compctl` would never be tried, while if it was handled by `_default`, `compctl` would be tried.

`users` This may be set to a list of names that should be completed whenever a username is needed. If it is not set or the string on the line doesn't match any of the strings in this list, all usernames will be completed.

`users-hosts`
The values of this style should be of the form '*user*`@`*host*' or '*user*`:`*host*'. It is used for commands that need pairs of user- and hostnames. For such commands, only the pairs from this style are used and if, for example, the username is already typed, then only the hostnames for which there is a pair with that username is defined.

If set for the `my-accounts` tag, this is used for commands such as `rlogin` and `ssh`; in this case the style should contain the names of the user's own accounts on remote hosts. If set for the `other-accounts` tag, it is used for commands such as `talk` and `finger` and should contain other people's accounts. Finally, it may also be used by some commands with the `accounts` tag.

`users-hosts-ports`
Like `users-hosts` but used for commands like `telnet` and containing strings of the form '*user*`@`*host*`:`*port*'.

`verbose` This is used in several contexts to decide if only a simple or a verbose list of matches should be generated. For example some commands show descriptions for option names if this style is 'true'.

The default value for this style is 'true'.

`word` This is used by the `_list` completer, which prevents the insertion of completions until a second completion attempt when the line has not changed. The normal way of finding out if the line has changed is to compare its entire contents between the two occasions. If this style is true, the comparison is instead performed only on the current word. Hence if completion is performed on another word with the same contents, completion will not be delayed.

19.4 Control Functions

The initialization script `compinit` redefines all the widgets which perform completion to call the supplied widget function `_main_complete`. This function acts as a wrapper calling the so-called 'completer' functions that generate matches. If `_main_complete` is called with arguments, these are taken as the names of completer functions to be called in the order given. If no arguments are given, the set of functions to try is taken from the `completer` style. For example, to use normal completion and correction if that doesn't generate any matches:

```
zstyle ':completion:*' completer _complete _correct
```

after calling `compinit`. The default value for this style is '`_complete _ignored`', i.e. normally only ordinary completion is tried, first with the effect of the `ignored-patterns` style and then without it. The `_main_complete` function uses the return value of the completer functions to decide if other completers should be called. If the return value is zero, no other completers are tried and the `_main_complete` function returns.

If the first argument to `_main_complete` is a single hyphen, the arguments will not be taken as names of completers. Instead, the second argument gives a name to use in the *completer* field of the context and the other arguments give a command name and arguments to call to generate the matches.

The following completer functions are contained in the distribution (users may write their own):

`_all_matches`

This completer can be used to add a string consisting of all other matches. To ensure, that this string is always added, this completer has to be used as the first completer in the list. The `avoid-completer` style is used to decide if the string should be added. This will only be done if the matches were generated by a completer not named by one of the values of the style.

This function also uses the style `old-matches`. If it is set to 'true' or to the string '`only`' and there is a list of matches from a previous completion, those matches will be inserted in the command line. If it is set to the the string '`only`', it will only insert an old list and won't add the string for all matches of the list currently being generated.

With the `old-matches` style set, this completer should probably not be called unconditionally. Instead one could use the `-e` option of the `zstyle` builtin command to add a condition to the `completer` or to the `old-matches` style. Alternatively, one could use the `_generic` function to bind `_all_matches` to a separate key binding, for example:

```
zle -C all-matches complete-word _generic
bindkey '^Xa' all-matches
zstyle ':completion:all-matches:*' old-matches only
zstyle ':completion:all-matches:*' completer _all_matches
```

`_approximate`

This completer function uses the `_complete` completer to generate a list of strings for the context the cursor is currently in, allowing you to specify a maximum number of errors: see the description of approximate matching in Section 13.8 [Filename Generation], page 38 for how errors are counted. The resulting list of corrected and completed strings is then presented to the user. The intended use of this completer function is to try after the normal `_complete` completer by setting:

```
zstyle ':completion:*' completer _complete _approximate
```

This will give correcting completion if and only if normal completion yields no possible completions. When corrected completions are found, the completer will normally start menu completion allowing you to cycle through these strings.

This completer uses the tags `corrections` and `original` when generating the possible corrections and the original string. The `format` style for the former may contain the additional sequences '`%e`' and '`%o`' which will be replaced by the number of errors accepted to generate the corrections and the original string, respectively.

As with all completers, `_approximate` uses its name without the underscore in the *completer* field of the context name. Once it has started trying to generate matches, it will append a minus sign and the number of errors accepted to its name. `_approximate` will first look for completions with one error, then two, and on so up to the limit on the number of errors set by the `max-errors` style. Hence on the first try the completer field of the context contains '`approximate-1`', on the second try '`approximate-2`', and so on.

When `_approximate` is called from another function, the number of errors to accept may be given with the `-a` option. Its argument should be the same as the value of the `max-errors` style, all in one string.

Note that this completer (and the `_correct` completer mentioned below) can be quite expensive to call, especially when a large number of errors are allowed. One way to avoid this is to set up the `completer` style using the `-e` option to zstyle so that some completers are only used when completion is attempted a second time on the same string, e.g.:

```
zstyle ':completion:*' completer '
  if [[ $_last_try != "$HISTNO$BUFFER$CURSOR" ]]; then
    _last_try="$HISTNO$BUFFER$CURSOR"
    reply=(_complete _match _prefix)
  else
    reply=(_ignored _correct _approximate)
  fi'
```

This uses the `HISTNO` parameter and the `BUFFER` and `CURSOR` special parameters that are available inside zle and completion widgets to find out if the command line hasn't changed since the last time completion was tried. Only then are the `_ignored`, `_correct` and `_approximate` completers called.

`_complete`

This completer generates all possible completions in a context-sensitive manner, i.e. using the settings defined with the `compdef` function explained above and the current settings of all special parameters. This gives the normal completion behaviour.

To complete arguments of commands, `_complete` uses the utility function `_normal`, which is in turn responsible for finding the particular function; it is described below. Various contexts of the form *-context-*, as mentioned above for the `#compdef` tag, are handled specially. These are:

`-array-value-`
: for completion on the right hand side of an array-assignment (‘`foo=(...)`’).

`-brace-parameter-`
: for completing the name of a parameter expansion within braces (‘`${...}`’).

`-command-`
: for completing in a command position.

`-condition-`
: for completion inside conditions (‘`[[...]]`’).

`-default-`
: for generating completions when no special completion function is used.

`-equal-`
: for completion of words beginning with an equals sign

`-first-`
: for adding completions before any other completion functions are tried; if this function sets the `_compskip` parameter to `all`, no other completion functions will be called, if it is set to a string containing the substring `patterns`, no pattern completion functions will be called, and if it is set to a string containing `default` the function for the ‘`-default-`’ context will not be called, but functions defined for commands will.

`-math-`
: for completion inside mathematical contexts, such as ‘`((...))`’.

`-parameter-`
: for completing the name of a parameter expansion (‘`$...`’).

`-redirect-`
: for completion after a redirection operator.

`-subscript-`
: for completion inside subscripts.

`-tilde-`
: for completion after a tilde (‘`~`’) character, but before a slash.

`-value-`
: for completion on the right hand side of an assignment.

Default implementations are supplied for each of these contexts, in most cases named after the context itself (e.g. completion for the ‘`-tilde-`’ context is done by the function named ‘`_tilde`’).

Before trying to find a function for a specific context, `_complete` checks if the parameter ‘`compcontext`’ is set. If it is set to an array, the elements are taken to be the possible matches which will be completed using the tag ‘`values`’ and the description ‘`value`’. If it is set to an associative array, the keys are used as the possible completions and the values (if non-empty) are used as descriptions for the matches. If ‘`compcontext`’ is set to a string containing colons, it should be of the form ‘*tag*`:`*descr*`:`*action*’. In this case the *tag* and *descr* give the tag and description to use and the *action* says what should be completed in one of the forms described for the `_arguments` utility function below.

Finally, if ‘`compcontext`’ is set to a string without colons, the value is taken as the name of the context to use and the function defined for that context will be called. For this purpose, there is a special context named `-command-line-` that completes whole command lines (commands and their arguments) and is not used by the completion system itself, but has a function handling completion for it.

`_correct`
: Generate corrections, but not completions, for the current word; this is similar to `_approximate` but will not allow any number of extra characters at the cursor as

that completer does, hence this is similar to spell-checking. It calls `_approximate` but uses a different *completer* field in the context name.

For example, with:

```
zstyle ':completion:::::' completer _complete _correct _approximate
zstyle ':completion:*:correct:::' max-errors 2 not-numeric
zstyle ':completion:*:approximate:::' max-errors 3 numeric
```

correction will accept up to two errors. If a numeric argument is given, correction will not be performed, but correcting completion will be, and will accept as many errors as given by the numeric argument. Without a numeric argument, first correction and then correcting completion will be tried, with the first one accepting two errors and the second one accepting three errors.

When `_correct` is called as a function, the number of errors to accept may be given following the `-a` option. The argument should be the same as the value of the `accept` style, all in one string.

This completer function is intended to be used without the `_approximate` completer or, as in the example, just before it. Using it after the `_approximate` completer is useless since `_approximate` will at least generate the corrected strings generated by the `_correct` completer – and probably more.

`_expand` This completer function does not really do completion, but instead checks if the word on the command line is eligible for expansion and, if it is, gives detailed control over how this expansion is done. When using this, one should not use the `expand-or-complete` widget, but instead use `complete-word`, as `expand-or-complete` will expand the string on the line before the completion widget is called. Also, this completer should be called before the `_complete` completer function.

The tags used when generating expansions are `all-expansions` for the string containing all possible expansions, `expansions` when adding the possible expansions as single matches and `original` when adding the original string from the line. In which order these strings are generated and which of these strings are generated at all can be controlled by using the `group-order` style and by modifying the `tag-order` style, as usual.

The format string for `all-expansions` and for `expansions` may contain the sequence '`%o`' which will be replaced by the original string from the line.

Which kind of expansion is tried is controlled by the `substitute`, `glob` and `subst-globs-only` styles.

When `_expand` is called as a function, the different modes may be selected with options. The `-s` to `substitute`, `-g` to `glob` and `-o` to `subst-globs-only`.

`_expand_alias`
If the word the cursor is on is an alias, it is expanded and no other completers are called. The types of aliases which are to be expanded can be controlled with the `regular`, `global` and `disabled` styles.

This function is also a bindable command, see Section 19.5 [Bindable Commands], page 159.

`_history` Complete words from the shell's command history. This completer uses the `remove-all-dups`, and `sort` styles also used by the `_history_complete_word` bindable command, see Section 19.5 [Bindable Commands], page 159 and Section 19.3 [Completion System Configuration], page 131.

`_ignored` The `ignored-patterns` style can be set to a list of patterns which are compared against possible completions; matching ones are removed. With this completer those matches can be reinstated, as if no `ignored-patterns` style were set. The

completer actually generates its own list of matches; which completers are used for this is determined in the same way as for the `_prefix` completer.

The `single-ignored` style is used if only one match could be generated. It can be set to `show` to prevent that match from being displayed or inserted into the line, or it can be set to `menu`, in which case the single match and the original string from the line will be offered in a menu completion.

`_list` This completer allows one to delay the insertion of matches until completion is attempted a second time without the word on the line being changed. On the first attempt, only the list of matches will be shown. It is affected by the styles `condition` and `word`, see Section 19.3 [Completion System Configuration], page 131.

`_match` This completer is intended to be used after the `_complete` completer. It allows one to give patterns on the command line and to complete all strings matching these patterns from the set of possible completions for the context the cursor is in, without having to set the `GLOB_COMPLETE` option.

Normally this will be done by taking the pattern from the line, inserting a '`*`' at the cursor position and comparing the resulting pattern with the possible completions generated. However, if the `match-original` style has a value of `only`, no '`*`' will be inserted. If `match-original` has any other non-empty string as its value, this completer will first try to generate matches without, then with a '`*`' inserted at the cursor position.

The generated matches will be offered in a menu completion unless the `insert-unambiguous` style is set to 'true'. In this case menu completion will only be started if no unambiguous string could be generated that is at least as long as the original string. The style may also be set to the string '`pattern`'. This will keep the pattern on the line intact as long as there isn't an unambiguous completion with which it could be replaced.

Note that the matcher specifications defined globally or used by the completion functions will not be used.

`_menu` This completer is a simple example function implemented to show how menu completion can be done in shell code. It should be used as the first completer and has the effect of making the code perform menu completion. Note that this is independent of the setting of the `MENU_COMPLETE` option and does not work with the other menu completion widgets such as `reverse-menu-complete`, or `accept-and-menu-complete`.

`_oldlist` This completer controls how the standard completion widgets behave when there is an existing list of completions which may have been generated by a special completion (i.e. a separately-bound completion command). It allows the ordinary completion keys to continue to use the list of completions thus generated, instead of producing a new list of ordinary contextual completions. It should appear in the list of completers before any of the widgets which generate matches. It uses two styles: `old-list` and `old-menu`, see Section 19.3 [Completion System Configuration], page 131.

`_prefix` This completer can be used to try completion with the suffix (everything after the cursor) ignored. In other words, the suffix will not be considered to be part of the word to complete and hence does not need to be matched. It uses the `completer` style to decide which other completers to call to try to generate matches. If this style is unset, the list of completers set for the current context is used – except, of course, the `_prefix` completer itself. Furthermore, if this completer appears more than once in the list of completers only those completers not already tried by the last invocation of `_prefix` will be called.

For example, consider this global `completer` style:

```
zstyle ':completion:*' completer \
    _complete _prefix _correct _prefix:foo
```

Here, the `_prefix` completer tries normal completion but ignoring the suffix. If that doesn't generate any matches, and neither does the call to the `_correct` completer after it, `_prefix` will be called a second time and, now only trying correction with the suffix ignored. If you want to use `_prefix` as the last resort and try only normal completion, you can use:

```
zstyle ':completion:*' completer _complete ... _prefix
zstyle ':completion::prefix:*' completer _complete
```

The `add-space` style is also used. If it is set to 'true' then `_prefix` will insert a space between the matches generated (if any) and the suffix.

Note that this completer is only useful if the `COMPLETE_IN_WORD` option is set; otherwise, the cursor will be moved to the end of the current word before the completion code is called and hence there will be no suffix.

19.5 Bindable Commands

In addition to the context-dependent completions provided, which are expected to work in an intuitively obvious way, there are a few widgets implementing special behaviour which can be bound separately to keys. The following is a list of these and their default bindings.

`_bash_completions`
: This function is used by two widgets, `_bash_complete-word` and `_bash_list-choices`. It exists to provide compatibility with completion bindings in bash. The last character of the binding determines what is completed: '`!`', command names; '`$`', environment variables; '`@`', host names; '`/`', file names; '`~`' user names. In bash, the binding preceded by '`\e`' gives completion, and preceded by '`^X`' lists options. As some of these bindings clash with standard zsh bindings, only '`\e~`' and '`^X~`' are bound by default. To add the rest, the following should be added to `.zshrc` after `compinit` has been run:

```
for key in '!' '$' '@' '/' '~'; do
  bindkey "\e$key" _bash_complete-word
  bindkey "^X$key" _bash_list-choices
done
```

This includes the bindings for '`~`' in case they were already bound to something else; the completion code does not override user bindings.

`_correct_filename (^XC)`
: Correct the filename path at the cursor position. Allows up to six errors in the name. Can also be called with an argument to correct a filename path, independently of zle; the correction is printed on standard output.

`_correct_word` (^Xc)
: Performs correction of the current argument using the usual contextual completions as possible choices. This stores the string '`correct-word`' in the *function* field of the context name and then calls the `_correct` completer.

`_expand_alias (^Xa)`
: This function can be used as a completer and as a bindable command. It expands the word the cursor is on if it is an alias. The types of aliases expanded can be controlled with the `regular`, `global` and `disabled` styles.

When used as a bindable command there is one additional feature that can be selected by setting the `complete` style to 'true'. In this case, if the word isn't the name of an alias, `_expand_alias` tries to complete the word to a full alias name without expanding it (but leaving the cursor directly after the completed word so that invoking `_expand_alias` once more will expand the now-complete alias name).

`_expand_word (^Xe)`
Performs expansion on the current word: equivalent to the standard `expand-word` command, but using the `_expand` completer. Before calling it, the *function* field is set to '`expand-word`'.

`_generic` This function is not defined as a widget and not bound by default. However, it can be used to define a widget and will then store the name of the widget in the *function* field of the context and call the completion system. This allows custom completion widgets with their own set of style settings to be easily defined. For example, to define a widget that does normal completion and starts menu selection, one could do:

```
zle -C foo complete-word _generic
bindkey '...' foo
zstyle ':completion:foo:*' menu yes select=1
```

`_history_complete_word` (\e/)
Complete words from the shell's command history. This uses the `list`, `remove-all-dups`, `sort`, and `stop` styles.

`_most_recent_file (^Xm)`
Complete the name of the most recently modified file matching the pattern on the command line (which may be blank). If given a numeric argument N, complete the Nth most recently modified file. Note the completion, if any, is always unique.

`_next_tags` (^Xn)
This command alters the set of matches used to that for the next tag, or set of tags, either as given by the `tag-order` style or as set by default; these matches would otherwise not be available. Successive invocations of the command cycle through all possible sets of tags.

`_read_comp (^X^R)`
Prompt the user for a string, and use that to perform completion on the current word. There are two possibilities for the string. First, it can be a set of words beginning '`_`', for example '`_files -/`', in which case the function with any arguments will be called to generate the completions. Unambiguous parts of the function name will be completed automatically (normal completion is not available at this point) until a space is typed.

Second, any other string will be passed as a set of arguments to `compadd` and should hence be an expression specifying what should be completed.

A very restricted set of editing commands is available when reading the string: '`DEL`' and '`^H`' delete the last character; '`^U`' deletes the line, and '`^C`' and '`^G`' abort the function, while '`RET`' accepts the completion. Note the string is used verbatim as a command line, so arguments must be quoted in accordance with standard shell rules.

Once a string has been read, the next call to `_read_comp` will use the existing string instead of reading a new one. To force a new string to be read, call `_read_comp` with a numeric argument.

`_complete_debug (^X?)`
This widget performs ordinary completion, but captures in a temporary file a trace of the shell commands executed by the completion system. Each completion attempt

gets its own file. A command to view each of these files is pushed onto the editor buffer stack.

`_complete_help (^Xh)`
: This widget displays information about the context names, the tags, and the completion functions used when completing at the current cursor position. If given a numeric argument other than `1` (as in '`ESC-2 ^Xh`'), then the styles used and the contexts for which they are used will be shown, too.

  Note that the information about styles may be incomplete; it depends on the information available from the completion functions called, which in turn is determined by the user's own styles and other settings.

`_complete_tag (^Xt)`
: This widget completes symbol tags created by the `etags` or `ctags` programmes (note there is no connection with the completion system's tags) stored in a file `TAGS`, in the format used by `etags`, or `tags`, in the format created by `ctags`. It will look back up the path hierarchy for the first occurrence of either file; if both exist, the file `TAGS` is preferred. You can specify the full path to a `TAGS` or `tags` file by setting the parameter `$TAGSFILE` or `$tagsfile` respectively. The corresponding completion tags used are `etags` and `vtags`, after emacs and vi respectively.

19.6 Utility Functions

Descriptions follow for utility functions that may be useful when writing completion functions. Most of these reside in the `Base` subdirectory. Like the example functions for commands in the distribution, the utility functions generating matches all follow the convention of returning zero if they generated completions and non-zero if no matching completions could be added.

When writing completion functions or other ZLE widgets that call completion, it might be interesting to know about two more features offered by the `_main_complete` function. The arrays `compprefuncs` and `comppostfuncs` may be set to contain names of functions that are to be called immediately before or after completion has been tried. The functions will only be called once, unless they put themselves into the arrays again.

`_all_labels` [`-12VJ`] *tag name descr* [*command args ...*]
: This is a convenient interface to the `_next_label` function below, implementing the loop shown in the `_next_label` example. The *command* is the one that should be called to generate the matches. The options stored in the parameter *name* will automatically be inserted into the *args* given to the *command*. Normally, they are put directly after the *command*, but if one of the *args* is a single hyphen, they are inserted directly before that. If the hyphen is the last argument, that will be removed from the argument list before the *command* is called. This allows `_all_labels` to be used in almost all cases where the matches can be generated by a single call to the `compadd` builtin command or by a call to one of the utility functions.

  For example:

```
local expl
...
if _requested foo; then
  ...
  _all_labels foo expl '...' compadd ... - $matches
fi
```

  Will complete the strings from the `matches` parameter, using `compadd` with additional options which will take precedence over those generated by `_all_labels`.

`_alternative` [`-C` *name*] *specs* ...

This function is useful in simple cases where multiple tags are available. Essentially, it implements a loop like the one described for the `_tags` function above.

The tags to use and the action to perform if a tag is requested are described using the *specs* which are of the form: '*tag*`:`*descr*`:`*action*'. The *tag*s are offered using `_tags` and if the tag is requested, the *action* is executed with the given description *descr*. The *action*s supported are those used by the `_arguments` function (described below), without the '`->`*state*' and '`=`...' forms.

For example, the *action* may be a simple function call. With that one could do:

```
_alternative \
    'users:user:_users' \
    'hosts:host:_hosts'
```

to offer usernames and hostnames as possible matches (which are generated by the `_users` and `_hosts` functions respectively).

Note that, like `_arguments` this will also use `_all_labels` to execute the actions, so one doesn't need to call that explicitly unless another tag is to be used, for example in a function called from `_alternative`.

Like `_tags` this function supports the `-C` option to give a different name for the argument context field.

`_arguments` *spec* ...

This function can be used to complete words on the line by describing the options and arguments which may be passed to the command for which completion is being performed. The description is given as arguments to this function, with each *spec* describing one option or normal argument of the command. The forms of *spec* understood are:

n`:`*message*`:`*action*
n`::`*message*`:`*action*

This describes the *n*'th normal argument. The *message* will be printed above the matches generated and the *action* says what can be completed in this position (see below). If there are two colons before the *message*, this describes an optional argument. If the *message* contains only white space, nothing will be printed above the matches unless the action adds an explanation string itself.

`:`*message*`:`*action*
`::`*message*`:`*action*

Like the previous one, but describing the *next* argument. I.e. if you want to describe all arguments a command can get, you can leave out the numbers in the description and just use this form to describe them one after another in the order they have to appear on the line.

`*:`*message*`:`*action*
`*::`*message*`:`*action*
`*:::`*message*`:`*action*

This describes how arguments (usually non-option arguments, those not beginning with `-` or `+`) are to be completed when no description with one of the first two forms was given. This also means that any number of arguments can be completed.

With two colons before the *message*, the `words` special array and the `CURRENT` special parameter are modified to refer only to the normal arguments when the *action* is executed or evaluated. With three colons before the *message* they are modified to refer only to the normal arguments covered by this description.

optspec[*description* ...]
: This describes an option and (if *description* is given) the arguments that have to come after the option. If no *description* is given, this means to offer only the option name as a possible completion in the right places. (Note that the brackets, above, around *description*, indicate that zero or more *description*s may appear; but the brackets are not themselves part of this format. If brackets are used, they are part of the *optspec*; see below.)

  In the descriptions below, the option names represented by *optname* are normally taken to be multi-character names, and a word from the line is considered to contain only one option (or none). By giving the `-s` option to `_arguments` before the first *spec*, each *optname* is considered to be a single character and each word from the line may contain more than one such option letter. However, words beginning with two hyphens (like '`--prefix`') are still considered to contain only one option name. This allows the use of the '`-s`' option to describe single-letter options together with such long option names.

  The `-s` option may be combined with the option `-w` to say that more option characters are to be expected even after an option that takes an argument. For example, if a command takes the options '`a`' and '`b`', where '`a`' takes an argument in the next word, `_arguments` would normally not complete the other option directly after '`-a`', but it would allow that if given the `-w` option.

  Similarly, the option `-W` may be given together with `-s` to force completion of single-letter options even after options that get an argument in the same word. For example, if a command takes the options '`a`' and '`b`', where '`a`' needs an argument in the same word, directly after the option character, `_arguments` would normally only execute the action for that argument and not offer other single-letter options as possible completions. If given the `-W` option, it will offer other options as possible completions after executing the action for the argument. Note that, depending on the action, this may mean that the other options can't really be completed, but at least they will be listed. For more control, use an utility function like `_guard` in the argument's action.

  The forms of *optspec* are:

  `*`*optspec*
  : If the option may be given more than once, a star ('`*`') must be added in front of one of the following forms of *optspec*. Otherwise, if the option is already on the line and to the left of the cursor, it is not offered as a possible completion again.

  `-`*optname*
  `+`*optname*
  : In the simplest form the *optspec* is just the option name beginning with a minus or a plus sign, such as '`-foo`'. The first argument for the option (if any) must follow as a *separate* word directly after the option.

    If the command accepts the option with either a leading minus or a leading plus sign, use either '`-+`*optname*' or '`+-`*optname*' to define both variants at once.

    In all the following forms, the leading '`-`' may be replaced or paired with '`+`' in this way.

-optname-
: The first argument of the option must come directly after the option name *in the same word*, as in '`-foo-:...`'.

-optname+
: The first argument may appear immediately after *optname* in the same word, or may instead appear as a separate word after the option.

-optname=
: The argument may appear as the next word, or in same word as the option name provided that it is separated from it by an equals sign.

-optname=-
: The argument to the option must appear after an equals sign in the same word, and may not be given in the next argument.

optspec`[`*explanation*`]`
: An explanation string may be appended to any of the preceding forms of *optspec* by enclosing it in brackets, as in '`-q[query operation]`'.

  The `verbose` style is used to decide if these explanation strings should be displayed with the option in a completion listing.

  If no bracketed explanation string is given but the `auto-description` style is set and only one argument is described for this *optspec*, the value of the style is displayed, with any appearance of the sequence '`%d`' in it replaced by the *message* of the first *description* that follows the *optspec*; see below.

Note that the special meaning of a leading or trailing `-` or `+` in *optspec* means that when the command to be completed accepts options like '`-+`' or '`-=`', the second character has to be quoted with a backslash, as in '`-\+`'.

Each *description* following an *optspec* must take one of the following forms:

`:`*message*`:`*action*
`::`*message*`:`*action*
: Describes a mandatory argument with one colon, or an optional argument with two colons. As in other forms of *spec*, the *message* will be printed above the matches generated (unless it contains only white space, see above) and the *action* says what can be completed in this position.

`:*`*pattern*`:`*message*`:`*action*
`:*`*pattern*`::`*message*`:`*action*
`:*`*pattern*`:::`*message*`:`*action*
: This describes multiple arguments. Only the *last* description may be given in this form. If the *pattern* is empty (i.e., `:*:`), all following words on the line are to be completed as described by the *action*; otherwise, all words up to a word matching the *pattern* are to be completed using the *action*.

When the *message* is preceded by two colons, the `words` special array and the `CURRENT` special parameter are modified during the execution or evaluation of the *action* to refer only to the words after the option. When preceded by three colons, they are modified to refer only to the words covered by this description.

Note that only one such '`:*`'-specification is useful and no other argument specification may be given after it.

To include a colon in any *optname*, *message*, or *action* anywhere above, it has to be preceded by a backslash, as '`\:`'.

Each of the six forms of *spec* (yes, there are six, keep track of the nestings) may be preceded by a list of option names and argument numbers with which the option or argument described is mutually exclusive. This list is given in parentheses, as in '`(-two -three 1)-one:...`' or '`(-foo):...`'. In the first example, the options '`-two`' and '`-three`' and the first argument will not be offered as possible completions if the option '`-one`' is on the line before the cursor, and in the second example the option '`-foo`' will not be offered if the argument described by the specification is on the line.

The list may also contain a single star (`*`) as one of its elements to specify that the description for the rest arguments (i.e. a specification of the form '`*:...`') should not be used, a colon (`:`) to specify that the descriptions for all normal (non-option-) arguments should not be used and a hyphen (`-`) to specify that the descriptions for all options should not be used. This paragraph desperately needs rewriting.

To simplify writing writing functions that call `_arguments` more than once, the *specs* may also start with the character '`!`' (exclamation mark) to make the spec *not* be completed. However, if this is used with one of the forms describing options, the option (and its arguments, if it takes any) will be understood and skipped if they appear on the command line. It's just that the option itself will not be completed. This is intended to be used with an array containing the options used in the first call to `arguments`. The second call can then use '`\!${^global_options}`' to ignore those options and complete only the ones understood in the current context.

In every case above, the *action* determines how the possible completions should be generated. In places where no sensible matches can be generated, the action should consist of only a space. This will make the *message* be displayed but no possible completions listed. Note that even in this case the colon at the end of the *message* is needed. The only case where it can be left is when neither a *message*, nor a *action* is given.

Except for the '`->`*string*' form below, the *action* will be executed by calling the `_all_labels` function to process all tag labels, so one doesn't need to call that explicitly unless another tag is to be used, for example in a function called in the *action*.

When only one of a fixed set of strings can be completed, the *action* can consist of these strings as a list in parentheses, as in:

```
:foo:(foo bar baz)
```

Such a list in doubled parentheses should contain strings consisting of the string to complete followed by '`\:`' and a description, as in:

```
:foo:((a\:bar b\:baz))
```

The matches will be listed together with their descriptions if the `description` style for the `values` tag is set.

An *action* of the form '`->`*string*' is used by functions that implement a state machine. In this case, the '*string*'s (with all leading and trailing spaces and tabs removed)

of all actions that have to be used will be stored in the global array `state`. The function returns with a non-zero return value if the cursor is not in a position where options can be completed or if the current word could not be completed to an option. But if the `-R` option is given to `_arguments`, the function will instead return with a return value of 300 (to make it distinguishable from other return values) after setting the global '`context`', '`line`' and '`opt_args`' parameters as described below, and without resetting any changes made to the special parameters such as `PREFIX` and `words`. This enables wrapper functions around `_arguments` to be able to find out if they have to make sure that the special completion parameters are not reset when they return.

Note that this means that a function calling `_arguments` with at least one action containing such a '`->`*string*' has to declare appropriate local parameters as in:

```
local context state line
typeset -A opt_args
```

This will ensure that `_arguments` does not create unused global parameters.

A string in braces is evaluated to generate the matches and if the *action* does not begin with an opening parentheses or brace, it is also split into separate words and executed. If the *action* starts with a space, this list of words will be invoked unchanged, otherwise it will be invoked with some extra strings placed after the first word which can be given as arguments to the `compadd` builtin command and which make sure that the *message* given in the description will be shown above the matches. These arguments are taken from the array parameter '`expl`' which will be set up before executing the *action* and hence may be used in it (normally in an expansion like '`$expl[@]`').

If the *action* starts with '`= `' (an equals sign followed by a space), `_arguments` will insert the contents of the *argument* field of the current context as the new first element in the `words` special array and increments the value of the `CURRENT` special parameter. In other words, it inserts a dummy element in the `words` array and makes `CURRENT` still point to the word in that array where the cursor is. This is only really useful when used with one of the forms that make `_arguments` modify the `words` array to contain only some of the words from the line, i.e. one of the argument description forms where the *message* is preceded by two or three colons. For example, when the function called in the action for such an argument itself uses `_arguments`, the dummy element is needed to make that second call to `_arguments` use all words from the restricted range for argument parsing. Without the inserted dummy element, the first word in the range would be taken (by the second `_arguments`) to be the command name and hence ignored.

During the evaluation or execution of the action the array '`line`' will be set to the command name and normal arguments from the command line, i.e. to the words from the command line excluding all options and their arguments. These are stored in the associative array '`opt_args`', using the option names as keys and their arguments as the values. For options that have more than one argument these are given as one string, separated by colons. All colons in the original arguments are preceded with backslashes.

The parameter '`context`' (set only in the calling function when using an action of the form '`->`*string*', not during the evaluation of other *action*s) is set to the automatically created context names. These are either strings of the form '`option-`*opt*`-`*n*' for the *n*'th argument of the option *-opt*, or strings of the form '`argument-`*n*' for the *n*'th argument (for rest arguments the *n* is the string '`rest`'). For example, when completing the argument of the `-o` option, the name is '`option-o-1`' and for the second normal (non-option-) argument it is '`argument-2`'.

Also, during the evaluation of the *action*, the context name in the `curcontext` parameter is changed by appending the same string that is stored in the `context` parameter.

It is also possible to specify multiple sets of options and arguments with the sets separated by single hyphens. The specifications before the first hyphen are shared by all sets given after the first hyphen. The first word in every other set gives the name of the set. This name may appear in exclusion lists in the specifications, either alone or before one of the possible values described above (with a '`-`' between the name and the rest).

For example:

```
_arguments \
    -a \
  - set1 \
    -c \
  - set2 \
    -d \
    ':arg:(x2 y2)'
```

This defines two sets. When the command line contains the option '`-c`', the '`-d`' option and the argument will not be considered possible completions. When it contains '`-d`' or an argument, the option '`-c`' will not be completed any more, but if '`-a`' is given, both sets will still be considered valid, because it appears before the first hyphen, so both sets contain this option.

If the name-string is of the form '`(`*name*`)`' then all specifications in the set have an implicit exclusion list containing the name of the set, i.e. all specifications are mutual exclusive with all other specifications in the same set. This is useful for defining multiple sets of options which are mutually exclusive and in which the options are aliases for each other. E.g.:

```
_arguments \
    -a -b \
  - '(compress)' \
    {-c,--compress}'[compress]' \
  - '(uncompress)' \
    {-d,--decompress}'[decompress]'
```

Note that using multiple sets will be slower than using only one set because the completion code has to parse the command line once for every set. So more than one set should only be used if the command syntax is too complicated. Note also that an option specification with rest-arguments (as in '`-foo:*:...`') often allows the use of multiple sets to be avoided.

To simplify the specifications for commands with standard option parsing, the options `-S` and `-A` may be given. With `-S`, no option will be completed after a '`--`' on the line and this argument will otherwise be ignored. With `-A`, no options will be completed after the first non-option argument on the line. The `-A` has to be followed by a pattern matching all strings which are not to be taken as arguments. For example, to make `_arguments` stop completing options after the first normal argument, but ignoring all strings starting with a hyphen even if they are not described by one of the *optspec*s, one would use: '`-A "-*"`'.

Another option supported is '`-O` *name*'. The *name* will be taken as the name of an array and its elements will be given to functions called to generate matches when executing the *actions*. For example, this allows one to give options for the `compadd` builtin that should be used for all *action*s.

Also, the `-M` option followed by a string may be given before the first description. The string will be used as the match specification when completing option names and values instead of the default '`r:|[_-]=* r:|=*`'.

Finally, the option `-C` can be given to make `_arguments` modify the `curcontext` parameter when an action of the form '`->`*state*' is used. This parameter is used to keep track of the current context and in this case it (and not the parameter `context` as explained above) has to be made local to make sure that calling functions don't use the modified value. Also, the local version of `curcontext` has to be initialised with the old value as in:

```
local curcontext="$curcontext"
```

The function can also be made to automatically complete long options for commands that support the '`--help`' option as, for example, most of the GNU commands do. For this, the string '`--`' must be given as one argument and if it is, the command from the line is invoked with the '`--help`' option and its output is parsed to find possible option names. Note that this means that you should be careful to make sure that this feature is not used for a command that does not support this option.

For such automatically found options that get an argument after an '`=`', the function also tries to automatically find out what should be completed as the argument. The possible completions for option-arguments can be described with the arguments after the '`--`' (which are not used as described above). Each argument contains one description of the form '*pattern*`:`*message*`:`*action*'. The *message* and the *action* have the same format as for the normal option descriptions described above. The *action* will be executed to complete arguments of options whose description in the output of the command from the line with the '`--help`' option matches the *pattern*. For example:

```
_arguments -- '*\*:toggle:(yes no)' \
              '*=FILE*:file:_files' \
              '*=DIR*:directory:_files -/'
```

Here, '`yes`' and '`no`' will be completed as the argument of options whose description ends in a star, file names for options that contain the substring '`=FILE`' in the description, and paths for options whose description contains '`=DIR`'. In fact, the last two patterns are not needed since this function always completes files for option descriptions containing '`=FILE`' and paths for option descriptions that contain '`=DIR`' or '`=PATH`'. These builtin patterns can be overridden by patterns given as arguments, however.

Note also that `_arguments` tries to find out automatically if the argument for an option is optional. If it fails to automatically detect this, the colon before the *message* can be doubled to tell it about this as described for the normal option descriptions above.

If the *pattern* ends in '`(-)`', this will removed from the pattern and the *action* will be used only directly after the '`=`', not in the next word. I.e., this is like a normal specification as described above using '`=-`'.

The option '`-i` *patterns*' (which must be given after the '`--`') can be used to give patterns for options which should not be completed. The patterns can be given as the name of an array parameter or as a literal list in parentheses. E.g. '`-i "(--(en|dis)able-FEATURE*)"`' will make the options '`--enable-FEATURE`' and '`--disable-FEATURE`' be ignored. The option '`-s` *pairs*' (again, after the '`--`') can be used to describe option aliases. Each *pair* consists of a pattern and a replacement. E.g. some `configure`-scripts describe options only as '`--enable-foo`', but also accept '`--disable-foo`'. To allow completion of the second form, one would use '`-s "(#--enable- --disable-)"`'.

Example:

```
_arguments '-l+:left border:' \
           '-format:paper size:(letter A4)' \
           '*-copy:output file:_files::resolution:(300 600)' \
           ':postscript file:_files -g \*.\(ps\|eps\)' \
           '*:page number:'
```

This describes three options: '`-l`', '`-format`', and '`-copy`'. The first one gets one argument described as '*left border*' for which no completion will be offered because of the empty action. The argument may come directly after the '`-l`' or it may be given as the next word on the line. The '`-format`' option gets one argument (in the next word) described as '*paper size*' for which only the strings '`letter`' and '`A4`' will be completed. The '`-copy`' option differs from the first two in that it may appear more than once on the command line and in that it accepts two arguments. The first one is mandatory and will be completed as a filename. The second one is optional (because of the second colon before the description '*resolution*') and will be completed from the strings '`300`' and '`600`'.

The last two descriptions say what should be completed as arguments. The first one describes the first argument as a '*postscript file*' and makes files ending in '`ps`' or '`eps`' be completed. The last description says that all other arguments are '*page numbers*' but does not give possible completions.

`_cache_invalid` *cache_identifier*

This function returns 0 if the completions cache corresponding to the given cache identifier needs rebuilding. It determines this by looking up the `cache-policy` style for the current context, and if it exists, runs the function of the same name, supplying the full path to the relevant cache file as the only argument.

Example:

```
_example_caching_policy () {
    # rebuild if cache is more than a week old
    oldp=( "$1"(Nmw+1) )
    (( $#oldp ))
}
```

`_call_function` *return name* [*args* ...]

If a function *name* exists, it is called with the arguments *args*. Unless it is the empty string or a single hyphen, *return* is taken as the name of a parameter and the return status from the called function is stored in it. The return value of `_call_function` itself is zero if the function *name* exists and was called and non-zero otherwise.

`_call_program` *tag string* ...

This function is used in places where a command is called, making it possible for the user to override the default command call. It looks up the `command` style with the supplied *tag*. If the style is set, its value is used as the command to execute.

In any case, the *string*s from the call to `_call_program` or from the style are concatenated with spaces between them and the resulting string is evaluated. The return value is the return value of the command called.

`_combination` [`-s` *pattern*] *tag style specs* ... *field opts* ...

This function is used to complete combinations of values such as pairs of hostnames and usernames. The possible values will be taken from the *style* whose name is given as the second argument. The first argument is the *tag* to use to do the lookup.

The style name should consist of multiple parts separated by hyphens which are then used as field names. Known values for such fields can be given after the second argument in arguments of the form '*field=pattern*'. The first argument without

an equals sign is taken as the name of the field for which completions should be generated.

The matches generated will be taken from the value of the style. These values should contain the possible values for the combinations where the values for the different fields are separated by colons or characters matching the pattern given after the `-s` option to `_combination`; normally this is used to define character classes like the '`-s "[:@]"`' used for the `users-hosts` style.

Only the values for the requested fields for which the patterns given in the '*field=pattern*' match the respective fields in the strings from the style value are generated as possible matches.

If no style with the given name is defined for the given tag but a function named with the name of the requested field preceded by an underscore is defined, that function will be called to generate the matches. This is also done if none of the strings in the value of the style match all the patterns given as arguments.

If the same name is used for more than one field, in both the '*field=pattern*' and the argument that gives the field name to complete for, the number of the field (starting with one) may be given after the fieldname, separated from it by a colon.

All arguments after the requested field name are passed to `compadd` when generating matches from the style value, or to the functions for the fields if they are called.

`_contexts` *names ...*

This function looks up the definitions for the context and command names given as arguments and calls the handler functions for them if there is a definition (given with the `compdef` function). For example, the function completing inside subscripts might use '`_contexts -math-`' to include the completions generated for mathematical environments.

`_describe` [`-o`] *descr name1* [*name2*] *opts ...* `--` *...*

This function is useful for preparing a list of command options or arguments, together with their descriptions *descr*, as matches. Multiple groups separated by `--` can be supplied, potentially with different completion options *opts*.

The *descr* is taken as a string to display above the matches if the `format` style for the `descriptions` tag is set. After this come one or two names of arrays followed by options to pass to `compadd`. The first array contains the possible completions with their descriptions in the form '*completion*`:`*description*'. If a second array is given, it should have the same number of elements as the first one and the corresponding elements are added as possible completions instead of the *completion* strings from the first array. The completion list will retain the descriptions from the first array. Finally, a set of completion options can appear.

If the option '`-o`' appears before the first argument, the matches added will be treated as option names (typically following a '`-`', '`--`' or '`+`' on the command line). This makes `_describe` use the `prefix-hidden`, `prefix-needed` and `verbose` styles to find out if the strings should be added at all and if the descriptions should be shown. Without the '`-o`' option, only the `verbose` style is used.

`_describe` uses the `_all_labels` function to generate the matches, so it does not need to appear inside a loop over tag labels.

`_description` [`-12VJ`] *tag name descr* [*specs ...*]

This function is called before completions are added (typically by a call to `compadd`); it tests various styles and arranges for any necessary options to be passed on to `compadd`. The styles are tested in the current context using the given *tag*; options are put into the array called *name* for passing on to `compadd`; the description for the current set of matches is passed in *descr*. The styles tested are: `format` (which is first tested for the given *tag* and then for the `descriptions` tag if that isn't

defined), `hidden`, `matcher`, `ignored-patterns` and `group-name` (the last are tested only for the tag given as the first argument). This function also calls the `_setup` function which tests some more styles.

The string returned by the `format` style (if any) will be modified so that the sequence '`%d`' is replaced by the *descr* given as the third argument without any leading or trailing white space. If, after removing the white space, the *descr* is the empty string, the format style will not be used and the options put into the *name* array will not contain an explanation string to be displayed above the matches. If `_description` is called with more than three arguments, the additional *specs* should be of the form '*char*`:`*str*' and every appearance of '`%`*char*' in the format string will be replaced by *string*.

The options placed in the array will also make sure that the matches are placed in a separate group, depending on the value of the `group-name` style. Normally a sorted group will be used for this (with the '`-J`' option), but if an option starting with '`-V`', '`-J`', '`-1`', or '`-2`' is given, that option will be included in the array, so that it is possible to make the group unsorted by giving the option '`-V`', '`-1V`', or '`-2V`'.

In most cases, the function will be used like this:

```
local expl
_description files expl file
compadd "$expl[@]" - "$files[@]"
```

Note the use of the parameter `expl`, the hyphen, and the list of matches. Almost all calls to `compadd` within the completion system use a similar format; this ensures that user-specified styles are correctly passed down to the builtins which implement the internals of completion.

`_files` The function `_files` uses the `file-patterns` style and calls `_path_files` with all the arguments it was passed except for `-g` and `-/`. These two options are used depending on the setting of the `file-patterns` style.

See `_path_files` below for a description of the full set of options accepted by `_files`.

`_gnu_generic`
This function is a simple wrapper around the `_arguments` function described above. It can be used to automatically complete long options for commands that understand the '`--help`' option. It is not intended to be used from completion functions but as a top-level completion function in its own right. For example, to enable option completion for the commands `foo` and `bar`, one would call:

```
compdef _gnu_generic foo bar
```

in one of the initialization files after the call to `compinit`.

The default installation uses this function only to generate completions for some GNU-commands because to complete the options, the command has to be called and hence it shouldn't be used if one can't be sure that the command understands the '`--help`' option.

`_guard` [*options*] *pattern* [*descr*]
This function is intended to be used in an action of functions like `_arguments`. It returns immediately with a non-zero return value if the string to be completed does not match the *pattern*. If the pattern matches, the *descr* is displayed and the function returns zero if the word to complete is not empty and non-zero otherwise.

The *pattern* may be preceded by those options understood by `compadd` that are passed down from `_description`, namely `-M`, `-J`, `-V`, `-1`, `-2`, `-n`, `-F` and `-X`. All of these options, except `-X`, will be ignored. If the `-X` option appears, the description

following it will be used as the string to display if the *pattern* matches, unless the option *descr* is given to `_guard` itself, which will then take precedence.

As an example, consider a command taking the options `-n` and `-none`, where `-n` has to be followed by a numeric value in the same word. By using either of:

```
_argument '-n-:numeric value:_guard "[0-9]#"' '-none'
```

or

```
_argument '-n-: :_guard "[0-9]#" "numeric value"' '-none'
```

`_arguments` can be made to both display the message '`numeric value`' and complete options after '`-n<TAB>`'. If the '`-n`' is already followed by one or more digits (matching the pattern given to `_guard`), only the message will be displayed and if the '`-n`' is followed by another character, only options are completed.

`_message` [-r] *descr*
The *descr* is used like the third argument to the `_description` function. However, the resulting string will always be shown whether or not matches were generated. This is useful to display help texts in places where no completions can be generated automatically.

This function also uses the `format` style for the `messages` tag in preference to the `format` style for the `descriptions` tag. The latter is used only if the former is unset.

If the `-r` option is given, no style is used and the *descr* is used literally as the string to display. This is only used in cases where that string is taken from some pre-processed argument list containing an expanded description.

`_multi_parts` *sep array*
This function receives two arguments: a separator character and an array. As usual, the *array* may be either the name of an array parameter or a literal array in the form '`(foo bar)`' (i.e. a list of words separated by white space in parentheses). With these arguments, this function will complete to strings from the array where the parts separated by the separator character are completed independently. For example, the `_tar` function from the distribution caches the pathnames from the tar file in an array, and then calls this function to complete these names in the way normal filenames are completed by the `_path_files` function, by using '`_multi_parts /` *patharray*'.

If the `-i` option is present, then any time there is a unique match it will immediately be inserted even if that requires additional separators to be inserted as well. When completing from a fixed set of possible completions which are really words, this is often the expected behaviour; however, if `_multi_parts` should behave like completing pathnames, the `-i` option should not be used.

Like other utility functions, this function accepts the '`-V`', '`-J`', '`-1`', '`-2`', '`-n`', '`-f`', '`-X`', '`-M`', '`-P`', '`-S`', '`-r`', '`-R`', and '`-q`' options and passes them to the `compadd` builtin.

`_next_label` [`-12VJ`] *tag name descr* [*options* ...]
This function should be called repeatedly to generate the tag labels. On each call it will check if another tag label is to be used and, if there is at least one, zero is returned. If no more tag labels are to be used, a non-zero status is returned.

The `-12JV` options and the first three arguments are given to the `_description` function using the tag label instead of the first argument as appropriate. The *options* given after the *descr* should be other options to be used for `compadd` or whatever function is to be called to add the matches. `_next_label` will store these *options* in the parameter whose *name* is given as the second argument. This is

done in such a way that the description given by the user to the `tag-order` style is preferred over the one given to `_next_label`.

Note that this function must not be called without a previous call to `_tags` or `_requested` because it uses the tag label for the current tag found by these functions.

A normal use of this function for the tag labels of the tag `foo` looks like this:

```
local expl ret=1
...
if _requested foo; then
  ...
  while _next_label foo expl '...'; do
    compadd "$expl[@]" ... && ret=0
  done
  ...
fi
return ret
```

`_normal` This function is used for normal command completion. It has two tasks: completing the first word on the command line as the name of a command, and completing the arguments to this command. In the second case, the name of the command is looked up to see if special completions exists, including completions defined for patterns which match the name. If none is found, completion is performed for the context `-default-`.

The function can also be called by other completion functions which need to treat a range of words as a command line. For example, the function to complete after the pre-command specifiers such as `nohup` removes the first word from the `words` array, decrements the `CURRENT` parameter, then calls `_normal` again, with the effect that '`nohup` *cmd ...*' is treated the same way was '*cmd ...*'.

If the command name matches a pattern, the parameter `_compskip` is checked after the call to the corresponding completion function. This has the same effect here as in the `-first-` context: if it is set, no more completion functions are called even if there are no matches so far.

`_options` This can be used to complete option names. It uses a matching specification that ignores a leading '`no`', ignores underscores and allows the user to type upper-case letters which will match their lower-case counterparts. All arguments passed to this function are propagated unchanged to the `compadd` builtin.

`_options_set` and `_options_unset`
These functions complete only set or unset options, with the same matching specification used in the `_options` function.

Note that you need to uncomment a few lines in the `_main_complete` function for these functions to work properly. The lines in question are used to store the option settings in effect before the completion widget locally sets the options it needs. Hence these options are not generally used by the completion system.

`_parameters`
This should be used to complete parameter names. `_parameters` can take a `-g` *pattern* option which specifies that only parameters whose type matches the *pattern* should be completed. Strings of the same form as those returned by the `t` parameter expansion flag are used here when matching the type. All other arguments are passed unchanged to the `compadd` builtin.

`_path_files`

The function `_path_files` is used throughout the completion system to complete filenames. It allows completion of partial paths. For example, the string '`/u/i/s/sig`' may be completed to '`/usr/include/sys/signal.h`'.

The options accepted by both `_path_files` and `_files` are:

`-f` Complete all filenames. This is the default.

`-/` Specifies that only directories should be completed.

`-g` *pattern* Specifies that only files matching the *pattern* should be completed.

`-W` *paths* Specifies path prefixes that are to be prepended to the string from the line to generate the filenames but that should not be inserted in the line or shown in a completion listing. Here, *paths* may be the name of an array parameter, a literal list of paths enclosed in parentheses or an absolute pathname.

`-F` This option from the `compadd` builtin gives direct control over which filenames should be ignored. If the option is not present, the `ignored-patterns` style is used.

These functions also accept the '`-J`', '`-V`', '`-1`', '`-2`', '`-n`', '`-X`', '`-M`', '`-P`', '`-S`', '`-q`', '`-r`', and '`-R`' options from the `compadd` builtin.

Finally, the `_path_files` function uses the styles `expand`, `ambiguous`, `special-dirs`, `list-suffixes` and `file-sort`.

`_regex_arguments` *name specs ...*

This function is a compiler to generate a completion function. The first argument specifies the name of the generated function while the remaining arguments specify a completion as a set of regular expressions with actions. The generated function has the structure of a finite-state machine whose states correspond to the state (i.e. the context) of the completion. This state machine uses a command line, which comes from the concatenation of the `words` array up to the current cursor position using null characters as separators with no extra quotation. This is analysed and at the end the appropriate action is executed.

Specification arguments take one of following forms, in which metacharacters such as '`(`', '`)`', '`#`' and '`|`' should be quoted.

/pattern/ [*%lookahead%*] [*-guard*] [*: tag : descr : action*]

This is a primitive element, corresponding to one state of the compiled state machine. The state is entered if '`(#b)((#B)`*pattern*`)(#B)`*lookahead*`*`' matches the command line string. If it matches, '*guard*' is evaluated and its return status is examined; if this is successful, the state is entered, otherwise the test fails and other candidates are tried. The *pattern* string '`[]`' is guaranteed never to match.

If the test succeeds and the state is entered, the left part of the command line string matched as *pattern* is removed and the next state is tried, proceeding from inside to outside and from left to right.

If no test succeeds and the remaining command line string contains no null character, the completion target is restricted to the remainder of the command line string and *action*s for the target are executed. In this case, nothing is actually removed from the command line string so that any previous or neighbouring state may also have *action*s. *action*s evaluation are ordered by the `tag-order` style and specified *tag* by `_alternative`. So, the various formats supported by `_alternative`

can be used in *action*. *descr* is used for setting up the array parameter `expl`.

/pattern/+ [*%lookahead%*] [*-guard*] [*:tag:descr:action*]
This is similar to '*/pattern/* ...' but the left part of the command line string is also considered as part of the completion target.

/pattern/- [*%lookahead%*] [*-guard*] [*:tag:descr:action*]
This is similar to '*/pattern/* ...' but the *action*s of the current and previous states are ignored even if the following state's '*pattern*' matches the empty string.

`(` *spec* `)` This groups *spec*s.

spec `#` This allows any number of repetitions of *spec*.

spec spec This represents the concatenation of two *spec*s.

spec `|` *spec*
Either of the two *spec*s can be matched.

`_requested` [`-12VJ`] *tag* [*name descr* [*command args* ...]]
This function is called to decide whether a tag already registered by a call to `_tags` (see below) is requested and hence completion should be performed for it; it returns status zero if the tag is requested and non-zero otherwise. This will usually be done in a loop such as the following:

```
_tags foo bar baz
while _tags; do
  if _requested foo; then
    ... # perform completion for foo
  fi
  ... # test the tags bar and baz in the same way
  ... # exit loop if matches were generated
done
```

Note that the test for whether matches were generated is not performed until the end of the `_tags` loop. This is so that the user can specify a set of tags to be tested at the same time in the `tag-order` parameter.

If the *name* and the *descr* are given, `_requested` calls the `_description` function with these arguments, including the options.

If the *command* is given, the `_all_labels` function will be called immediately with the same arguments. This is often useful to do both the testing of the tag, getting the description for the matches and adding the matches at once. For example:

```
local expl ret=1
_tags foo bar baz
while _tags; do
  _requested foo expl 'description' \
      compadd foobar foobaz && ret=0
  ...
  (( ret )) || break
done
```

Note that this means that the *command* has to accept the options that have to be passed down to `compadd`.

`_retrieve_cache` *cache_identifier*
This function retrieves completion information from the file given by *cache_identifier*, stored in a directory specified by the `cache-path` style (defaults to `~/.zsh/cache`).

The return value is zero if retrieval was successful. It will only attempt retrieval if the `use-cache` style is set, so you can call this function without worrying about whether the user wanted to use the caching layer.

See `_store_cache` below for more details.

`_sep_parts`
: This function is passed alternating arrays and separators as arguments. The arrays specify completions for parts of strings to be separated by the separators. The arrays may be the names of array parameters or a quoted list of words in parentheses. For example, with the array '`hosts=(ftp news)`' the call '`_sep_parts '(foo bar)' @ hosts`' will complete the string '`f`' to '`foo`' and the string '`b@n`' to '`bar@news`'.

  This function passes the '`-V`', '`-J`', '`-1`', '`-2`', '`-n`', '`-X`', '`-M`', '`-P`', '`-S`', '`-r`', '`-R`', and '`-q`' options and their arguments to the `compadd` builtin used to add the matches.

`_setup` *tag* [*group*]
: This function expects a tag as its argument and sets up the special parameters used by the completion system appropriately for the tag, using styles such as `list-colors` and `last-prompt`.

  The optional *group* gives the name of the group in which the matches will be placed. If it is not given, the *tag* is used as the group name.

  Note that this function is called automatically from `_description` so that one normally doesn't have to call it explicitly.

`_store_cache` *cache_identifier vars ...*
: This function, when combined with `_retrieve_cache` and `_cache_invalid`, makes it easy to implement a caching layer for your completion functions. If a completion function needs to perform a costly operation in order to generate data which is used to calculate completions, you can store that data in variables, and use this function to dump the values of those variables to a file. Then, if they are needed in subsequent shell invocations, they can be retrieved quickly from that file via `_retrieve_cache`, avoiding the need for repeating the costly operation.

  The *cache_identifier* specifies the file which the data should be dumped to, and is stored in a directory specified by the `cache-path` style (defaults to `~/.zsh/cache`). The remaining *vars* arguments are the variables to dump to the file.

  The return value is zero if storage was successful. The function will only attempt storage if the `use-cache` style is set, so you can call this function without worrying about whether the user wanted to use the caching layer.

  If your completion function avoids calling _retrieve_cache when it already has the completion data in the environment, it should probably at least call `_cache_invalid` to check whether this data and the data cached on disk is still valid.

  See the _perl_modules completion function for a simple example of usage of this caching layer.

`_tags` [`-C` *name* [*tags ...*]]
: If called with arguments, these are taken as the names of the tags for the types of matches the calling completion function can generate in the current context. These tags are stored internally and sorted by using the `tag-order` style. Following calls to this function without arguments from the same function will then select the first, second, etc. set of tags requested by the user. To test if a certain tag should be tried, the `_requested` function has to be called (see above).

  The return value is zero if at least one of the tags is requested and non-zero otherwise.

  This function also accepts the `-C` option followed by a *name*. This name is temporarily (i.e. not visible outside `_tags`) stored in the argument field of the context name in the `curcontext` parameter. This allows `_tags` to be made to use a more specific

context name without having to change and reset the `curcontext` parameter (which would otherwise have the same effect).

`_values` *specs* ...

This is used to complete values (strings) and their arguments or lists of such values. It can be used in two ways.

If the first argument is the option '`-O` *name*', this will be used in the same way as by the `_arguments` function, in other words the elements of the *name* array will be given to calls to `compadd` and when executing an action.

Otherwise, if the first argument (or the first argument after the '`-O` *name*' option if that is used) is the option '`-s`', the next argument is used as the character that separates multiple values. Thus the values completed appear in the same word on the command line, unlike completion using `_arguments`.

The first argument (after the options and separator character if they are given) is used as a string to print as a description before listing the values.

All other arguments describe the possible values and their arguments in the same format used for the description of options by the `_arguments` function (see above). The only differences are that no minus or plus sign is required at the beginning, that values can have only one argument and that those forms of actions beginning with an equal sign are not supported.

The character separating a value from its argument can be set using the option `-S` (like `-s`, followed by the character to use as the separator in the next argument). If this option is not used, the equal sign will be used as the separator.

Example:

```
_values -s , 'description' \
        '*foo[bar]' \
        '(two)*one[number]:first count:' \
        'two[another number]::second count:(1 2 3)'
```

This describes three possible values: '`foo`', '`one`', and '`two`'. The first is described as '`bar`', takes no argument and may appear more than once. The second is described as '`number`', may appear more than once, and takes one mandatory argument described as '`first count`' for which no action is specified so that it will not be completed automatically. The '`(two)`' at the beginning says that if the value '`one`' is on the line, the value '`two`' will not be considered to be a possible completion anymore. Finally, the last value ('`two`') is described as '`another number`' and takes an optional argument described as '`second count`' which will be completed from the strings '`1`', '`2`', and '`3`'. The `_values` function will complete lists of these values separated by commas.

Like `_arguments` this function temporarily adds another context name component to the current context name while executing the *action*. Here this name is just the name of the value for which the argument is completed.

To decide if the descriptions for the values (not those for the arguments) should be printed, the style `verbose` is used.

One last difference from `_arguments` is that this function uses the associative array `val_args` to report values and their arguments, although otherwise this is the same as the `opt_args` association used by `_arguments`. This also means that the function calling `_values` should declare the `state`, `line`, `context` and `val_args` parameters as in:

```
local context state line
typeset -A val_args
```

when using an action of the form '`->`*string*'. With this function the `context` parameter will be set to the name of the value whose argument is to be completed.

Note also that `_values` normally adds the character used as the separator between values as a auto-removable suffix so that users don't have to type it themselves. But when using a '`->`*string*' action `_values` can't do that because the matches for the argument will be generated by the calling function. To get the usual behaviour, the implementor of the calling function has to add the suffix directly by passing the options '`-qS` *x*' (where *x* is the separator character specified with the `-s` option of `_values`) to the function generating the matches or to the `compadd` builtin.

Like `_arguments`, `_values` supports the `-C` option in which case you have to make the parameter `curcontext` local instead of `context` (as described above).

`_wanted` [`-C` *name*] [`-12VJ`] *tag name descr command args ...*
In many contexts, completion will generate one particular set of matches (usually corresponding to a single tag); however, it is still necessary to decide whether the user requires matches of this type. This function is useful in such a case.

Like `_requested`, it should be passed arguments as for `_description`. It calls `_tags` with the given *tag* and if that returns zero (so that the *tag* is requested by the user) it calls `_description`. Hence to offer only one tag and immediately use the description generated:

```
_wanted tag expl 'description' \
    compadd matches...
```

Unlike `_requested`, however, `_wanted` cannot be called without the *command*. This is because `_wanted` also implements the loop over the tags, not just the one for the labels; conversely, it should not be called in the middle of a `_tags` loop.

Note that, as for `_requested`, the *command* has to accept the options that have to be passed down to `compadd`.

Like `_tags` this function supports the `-C` option to give a different name for the argument context field.

19.7 Completion Directories

In the source distribution, the files are contained in various subdirectories of the `Completion` directory. They may have been installed in the same structure, or into one single function directory. The following is a description of the files found in the original directory structure. If you wish to alter an installed file, you will need to copy it to some directory which appears earlier in your `fpath` than the standard directory where it appears.

`Base` The core functions and special completion widgets automatically bound to keys. You will certainly need most of these, though will probably not need to alter them. Many of these are documented above.

`Zsh` Functions for completing arguments of shell builtin commands and utility functions for this. Some of these are also used by functions from the `Unix` directory.

`Unix` Functions for completing arguments of external commands and suites of commands. They may need modifying for your system, although in many cases some attempt is made to decide which version of a command is present. For example, completion for the `mount` command tries to determine the system it is running on, while completion for many other utilities try to decide whether the GNU version of the command is in use, and hence whether the `--help` option is supported..

`X`, `AIX`, `BSD`, ...
Completion and utility function for commands available only on some systems.

20 Completion Using compctl

20.1 Types of completion

This version of zsh has two ways of performing completion of words on the command line. New users of the shell may prefer to use the newer and more powerful system based on shell functions; this is described in Chapter 19 [Completion System], page 127, and the basic shell mechanisms which support it are described in Chapter 18 [Completion Widgets], page 113. This chapter describes the older `compctl` command.

20.2 Description

`compctl` [`-CDT`] *options* [*command ...*]

`compctl` [`-CDT`] *options* [`-x` *pattern options* `-` ... `--`] [`+` *options* [`-x` ... `--`] ... [`+`]] [*command ...*]

`compctl -M` *match-specs ...*

`compctl -L` [`-CDTM`] [*command ...*]

`compctl +` *command ...*

Control the editor's completion behavior 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.

20.3 Command Flags

Completion of the arguments of a command may be different for each command or may use the default. The behavior 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 20.4 [Option Flags], page 180:

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. If the command starts with a `=`, completion is tried with the pathname of the command.
>
> Any of the *command* strings may be patterns of the form normally used for filename generation. These should be be quoted to protect them from immediate expansion; for example the command string `'foo*'` arranges for completion of the words of any command beginning with `foo`. When completion is attempted, all pattern completions are tried in the reverse order of their definition until one matches. By default, completion then proceeds as normal, i.e. the shell will try to generate more matches for the specific command on the command line; this can be overridden by including `-tn` in the flags for the pattern completion.

Note that aliases are expanded before the command name is determined unless the `COMPLETE_ALIASES` option is set. Commands may not be combined with the `-C`, `-D` or `-T` flags.

`-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.

`-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.

`-T` supplies completion flags to be used before any other processing is done, even before processing for `compctl`s defined for specific commands. This is especially useful when combined with extended completion (the `-x` flag, see Section 20.6 [Extended Completion], page 185 below). Using this flag you can define default behavior which will apply to all commands without exception, or you can alter the standard behavior 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 -T -x 's[~] C[0,[^/]#]' -k friends -S/ -tn
```

to complete the strings in the array `friends` after a '`~`'. The `C[...]` argument is necessary so that this form of `~`-completion is not tried after the directory name is finished.

`-L` lists the existing completion behavior in a manner suitable for putting into a start-up script; the existing behavior is not changed. Any combination of the above forms, or the `-M` flag (which must follow the `-L` flag), 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 behavior 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.

The form with `-M` as the first and only option defines global matching specifications (see Section 18.5 [Matching Control], page 124). The match specifications given will be used for every completion attempt (only when using `compctl`, not with the new completion system) and are tried in the order in which they are defined until one generates at least one match. E.g.:

```
compctl -M '' 'm:{a-zA-Z}={A-Za-z}'
```

This will first try completion without any global match specifications (the empty string) and, if that generates no matches, will try case insensitive completion.

20.4 Option Flags

[`-fcFBdeaRGovNAIOPZEnbjrzu/12`]
[`-k` *array*] [`-g` *globstring*] [`-s` *substring*]
[`-K` *function*]
[`-Q`] [`-P` *prefix*] [`-S` *suffix*]
[`-W` *file-prefix*] [`-H` *num pattern*]
[`-q`] [`-X` *explanation*] [`-Y` *explanation*]

[`-y` *func-or-var*] [`-l` *cmd*] [`-h` *cmd*] [`-U`]
[`-t` *continue*] [`-J` *name*] [`-V` *name*]
[`-M` *match-spec*]

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 as follows.

20.4.1 Simple Flags

These produce completion lists made up by the shell itself:

`-f` Filenames and filesystem paths.

`-/` Just filesystem 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 15 [Options], page 59).

`-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.

20.4.2 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 the 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.)

`-s` *substring*
The *substring* is split into words and these words are than expanded using all shell expansion mechanisms (see Chapter 13 [Expansion], page 25). The resulting words are taken as possible completions. The `fignore` special parameter is not applied to the resulting files. Note that `-g` is faster for filenames.

`-K` *function*
Call the given function to get the completions. Unless the name starts with an underscore, 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 whole command line can be accessed with the `-c` and `-l` flags of the `read` builtin. 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 '`reply=`users``' would be 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 ''
```

which forces completion to look back in the history list for a word if no filename matches.

20.4.3 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.

`-W` *file-prefix*
With directory *file-prefix*: for command, file, directory and globbing completion (options `-c`, `-f`, `-/`, `-g`), the file prefix is implicitly added in front of the completion. For example,

```
compctl -/ -W ~/Mail maildirs
```

completes any subdirectories to any depth beneath the directory `~/Mail`, although that prefix does not appear on the command line. The *file-prefix* may also be of the form accepted by the `-k` flag, i.e. the name of an array or a literal list in parenthesis. In this case all the directories in the list will be searched for possible completions.

`-q` If used with a suffix as specified by the `-S` option, this causes the suffix to be removed if the next character typed is a blank or does not insert anything or if the suffix consists of only one character and the next character typed is the same character; this the same rule used for the `AUTO_REMOVE_SLASH` option. The option is most useful for list separators (comma, colon, etc.).

`-l` *cmd* This option 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 20.6 [Extended Completion], page 185 below) the range is restricted to the range of 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.

`-h` *cmd* Normally zsh completes quoted strings as a whole. With this option, completion can be done separately on different parts of such strings. It works like the `-l` option but makes the completion code work on the parts of the current word that are separated by spaces. These parts are completed as if they were arguments to the given *cmd*. If *cmd* is the empty string, the first part is completed as a command name, as with `-l`.

`-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. Since the produced possible completions seldom have interesting common prefixes and suffixes, menu completion is started immediately if `AUTO_MENU` is set and this flag is used.

`-y` *func-or-var*
: The list provided by *func-or-var* is displayed instead of the list of completions whenever a listing is required; the actual completions to be inserted are not affected. It can be provided in two ways. Firstly, if *func-or-var* begins with a `$` it defines a variable, or if it begins with a left parenthesis a literal array, which contains the list. A variable may have been set by a call to a function using the `-K` option. Otherwise it contains the name of a function which will be executed to create the list. The function will be passed as an argument list all matching completions, including prefixes and suffixes expanded in full, and should set the array `reply` to the result. In both cases, the display list will only be retrieved after a complete list of matches has been created.

  Note that the returned list does not have to correspond, even in length, to the original set of matches, and may be passed as a scalar instead of an array. No special formatting of characters is performed on the output in this case; in particular, newlines are printed literally and if they appear output in columns is suppressed.

`-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 that were added for this explanation string. The explanation only appears if completion was tried and there was no unique match, or when listing completions. Explanation strings will be listed together with the matches of the group specified together with the `-X` option (using the `-J` or `-V` option). If the same explanation string is given to multiple `-X` options, the string appears only once (for each group) and the number of matches shown for the '`%n`' is the total number of all matches for each of these uses. In any case, the explanation string will only be shown if there was at least one match added for the explanation string.

  The sequences `%B`, `%b`, `%S`, `%s`, `%U`, and `%u` specify output attributes (bold, standout, and underline) and `%{...%}` can be used to include literal escape sequences as in prompts.

`-Y` *explanation*
: Identical to `-X`, except that the *explanation* first undergoes expansion following the usual rules for strings in double quotes. The expansion will be carried out after any functions are called for the `-K` or `-y` options, allowing them to set variables.

`-t` *continue*
: The *continue*-string contains a character that specifies which set of completion flags should be used next. It is useful:

  (i) With `-T`, or when trying a list of pattern completions, when `compctl` would usually continue with ordinary processing after finding matches; this can be suppressed with '`-tn`'.

  (ii) With a list of alternatives separated by `+`, when `compctl` would normally stop when one of the alternatives generates matches. It can be forced to consider the next set of completions by adding '`-t+`' to the flags of the alternative before the '`+`'.

  (iii) In an extended completion list (see below), when `compctl` would normally continue until a set of conditions succeeded, then use only the immediately following flags. With '`-t-`', `compctl` will continue trying extended completions after the next '`-`'; with '`-tx`' it will attempt completion with the default flags, in other words those before the '`-x`'.

`-J` *name* This gives the name of the group the matches should be placed in. Groups are listed and sorted separately; likewise, menu completion will offer the matches in the groups in the order in which the groups were defined. If no group name is explicitly given, the matches are stored in a group named *default*. The first time a group name is encountered, a group with that name is created. After that all matches with the same group name are stored in that group.

This can be useful with non-exclusive alternative completions. For example, in

```
compctl -f -J files -t+ + -v -J variables foo
```

both files and variables are possible completions, as the `-t+` forces both sets of alternatives before and after the `+` to be considered at once. Because of the `-J` options, however, all files are listed before all variables.

`-V` *name* Like `-J`, but matches within the group will not be sorted in listings nor in menu completion. These unsorted groups are in a different name space from the sorted ones, so groups defined as `-J files` and `-V files` are distinct.

`-1` If given together with the `-V` option, makes only consecutive duplicates in the group be removed. Note that groups with and without this flag are in different name spaces.

`-2` If given together with the `-J` or `-V` option, makes all duplicates be kept. Again, groups with and without this flag are in different name spaces.

`-M` *match-spec*
This defines additional matching control specifications that should be used only when testing words for the list of flags this flag appears in. The format of the *match-spec* string is described in Section 18.5 [Matching Control], page 124.

20.5 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. If the list of flags contains a `-t` with a `+` character, the next list of flags is used even if the current list produced matches.

20.6 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 20.4 [Option Flags], page 180 above, 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 metacharacters 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. For example,

```
compctl -s ''`users`'' -x 'n[1,@]' -k hosts -- talk
```

  will usually complete usernames, but if you insert an `@` after the name, names from the array *hosts* (assumed to contain hostnames, though you must make the array yourself) will be completed. Other commands such as `rcp` can be handled similarly.

`N[`*index*`,`*string*`]`...
: Like `n` 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 after the one matched by *str1* it matches only if the cursor is before this word. If the comma and *str2* are omitted, it matches if the cursor is after a word with prefix *str1*.

`R[`*str1*`,`*str2*`]`...
: Like `r` but using pattern matching instead.

`q[`*str*`]`...
: Matches the word currently being completed is in single quotes and the *str* begins with the letter 's', or if completion is done in double quotes and *str* starts with the letter 'd', or if completion is done in backticks and *str* starts with a 'b'.

20.7 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.

21 Zsh Modules

21.1 Description

Some optional parts of zsh are in modules, separate from the core of the shell. Each of these modules may be linked in to the shell at build time, or can be dynamically linked while the shell is running if the installation supports this feature. The modules that are bundled with the zsh distribution are:

`zsh/cap`
: Builtins for manipulating POSIX.1e (POSIX.6) capability (privilege) sets.

`zsh/clone`
: A builtin that can clone a running shell onto another terminal.

`zsh/compctl`
: The `compctl` builtin for controlling completion.

`zsh/complete`
: The basic completion code.

`zsh/complist`
: Completion listing extensions.

`zsh/computil`
: A module with utility builtins needed for the shell function based completion system.

`zsh/deltochar`
: A ZLE function duplicating EMACS' `zap-to-char`.

`zsh/example`
: An example of how to write a module.

`zsh/files`
: Some basic file manipulation commands as builtins.

`zsh/mapfile`
: Access to external files via a special associative array.

`zsh/mathfunc`
: Standard scientific functions for use in mathematical evaluations.

`zsh/parameter`
: Access to internal hash tables via special associative arrays.

`zsh/sched`
: A builtin that provides a timed execution facility within the shell.

`zsh/stat` A builtin command interface to the `stat` system call.

`zsh/termcap`
: Interface to the termcap database.

`zsh/terminfo`
: Interface to the terminfo database.

`zsh/zftp` A builtin FTP client.

`zsh/zle` The Zsh Line Editor, including the `bindkey` and `vared` builtins.

`zsh/zleparameter`
: Access to internals of the Zsh Line Editor via parameters.

`zsh/zprof`
: A module allowing profiling for shell functions.

`zsh/zpty` A builtin for starting a command in a pseudo-terminal.

`zsh/zutil`
: Some utility builtins, e.g. the one for supporting configuration via styles.

21.2 The zsh/cap Module

The `zsh/cap` module is used for manipulating POSIX.1e (POSIX.6) capability sets. If the operating system does not support this interface, the builtins defined by this module will do nothing. The builtins in this module are:

`cap` [*capabilities*]
: Change the shell's process capability sets to the specified *capabilities*, otherwise display the shell's current capabilities.

`getcap` *filename* ...
: This is a built-in implementation of the POSIX standard utility. It displays the capability sets on each specified *filename*.

`setcap` *capabilities filename* ...
: This is a built-in implementation of the POSIX standard utility. It sets the capability sets on each specified *filename* to the specified *capabilities*.

21.3 The zsh/clone Module

The `zsh/clone` module makes available one builtin command:

`clone` *tty* Creates a forked instance of the current shell, attached to the specified *tty*. In the new shell, the `PID`, `PPID` and `TTY` special parameters are changed appropriately. `$!` is set to zero in the new shell, and to the new shell's PID in the original shell.

The return value of the builtin is zero in both shells if successful, and non-zero on error.

21.4 The zsh/compctl Module

The `zsh/compctl` module makes available two builtin commands. `compctl`, is the old, deprecated way to control completions for ZLE. See Chapter 20 [Completion Using compctl], page 179. The other builtin command, `compcall` can be used in user-defined completion widgets, see Chapter 18 [Completion Widgets], page 113.

21.5 The zsh/complete Module

The `zsh/complete` module makes available several builtin commands which can be used in user-defined completion widgets, see Chapter 18 [Completion Widgets], page 113.

21.6 The zsh/complist Module

The `zsh/complist` module offers three extensions to completion listings: the ability to highlight matches in such a list, the ability to scroll through long lists and a different style of menu completion.

21.6.1 Colored completion listings

Whenever one of the parameters `ZLS_COLORS` or `ZLS_COLOURS` is set and the `zsh/complist` module is loaded or linked into the shell, completion lists will be colored. Note, however, that `complist` will not automatically be loaded if it is not linked in: on systems with dynamic loading, '`zmodload zsh/complist`' is required.

The parameters `ZLS_COLORS` and `ZLS_COLOURS` describe how matches are highlighted. To turn on highlighting an empty value suffices, in which case all the default values given below will be used. The format of the value of these parameters is the same as used by the GNU version of the `ls` command: a colon-separated list of specifications of the form '*name*=*value*'. The *name* may be one of the following strings, most of which specify file types for which the *value* will be used. The strings and their default values are:

`no 0` for normal text (i.e. when displaying something other than a matched file)

`fi 0` for regular files

`di 32` for directories

`ln 36` for symbolic links

`pi 31` for named pipes (FIFOs)

`so 33` for sockets

`bd 44;37` for block devices

`cd 44;37` for character devices

`ex 35` for executable files

`mi` *none* for a non-existent file (default is the value defined for `fi`)

`lc \e[` for the left code (see below)

`rc m` for the right code

`tc 0` for the character indicating the file type printed after filenames if the `LIST_TYPES` option is set

`sp 0` for the spaces printed after matches to align the next column

`ec` *none* for the end code

Apart from these strings, the *name* may also be an asterisk ('`*`') followed by any string. The *value* given for such a string will be used for all files whose name ends with the string. The *name* may also be an equals sign ('`=`') followed by a pattern. The *value* given for this pattern will be used for all matches (not just filenames) whose display string are matched by the pattern. Definitions for both of these take precedence over the values defined for file types and the form with the leading asterisk takes precedence over the form with the leading equal sign.

The last form also allows different parts of the displayed strings to be colored differently. For this, the pattern has to use the '`(#b)`' globbing flag and pairs of parentheses surrounding the parts of the strings that are to be colored differently. In this case the *value* may consist of more than one color code separated by equal signs. The first code will be used for all parts for which no explicit code is specified and the following codes will be used for the parts matched by the sub-patterns in parentheses. For example, the specification '`=(#b)(?)*(?)=0=3=7`' will be used for all matches which are at least two characters long and will use the code '`3`' for the first character, '`7`' for the last character and '`0`' for the rest.

All three forms of *name* may be preceded by a pattern in parentheses. If this is given, the *value* will be used only for matches in groups whose names are matched by the pattern given in the parentheses. For example, '`(g*)m*=43`' highlights all matches beginning with '`m`' in groups whose names begin with '`g`' using the color code '`43`'. In case of the '`lc`', '`rc`', and '`ec`' codes, the group pattern is ignored.

Note also that all patterns are tried in the order in which they appear in the parameter value until the first one matches which is then used.

When printing a match, the code prints the value of `lc`, the value for the file-type or the last matching specification with a '`*`', the value of `rc`, the string to display for the match itself, and then the value of `ec` if that is defined or the values of `lc`, `no`, and `rc` if `ec` is not defined.

The default values are ISO 6429 (ANSI) compliant and can be used on vt100 compatible terminals such as `xterm`s. On monochrome terminals the default values will have no visible effect. The `colors` function from the contribution can be used to get associative arrays containing the codes for ANSI terminals (see Section 23.5 [Other Functions], page 232). For example, after loading `colors`, one could use '`$colors[red]`' to get the code for foreground color red and '`$colors[bg-green]`' for the code for background color green.

If the completion system invoked by compinit is used, these parameters should not be set directly because the system controls them itself. Instead, the `list-colors` style should be used (see Section 19.3 [Completion System Configuration], page 131).

21.6.2 Scrolling in completion listings

To enable scrolling through a completion list, the `LISTPROMPT` parameter must be set. Its value will be used as the prompt; if it is the empty string, a default prompt will be used. The value may contain escapes of the form '`%x`'. It supports the escapes '`%B`', '`%b`', '`%S`', '`%s`', '`%U`', '`%u`' and

'%{...%}' used also in shell prompts as well as three pairs of additional sequences: a '%l' or '%L' is replaced by the number of the last line shown and the total number of lines in the form '*number*/*total*'; a '%m' or '%M' is replaced with the number of the last match shown and the total number of matches; and '%p' or '%P' is replaced with 'Top', 'Bottom' or the position of the first line shown in percent of the total number of lines, respectively. In each of these cases the form with the uppercase letter will be replaced with a string of fixed width, padded to the right with spaces, while the lowercase form will not be padded.

If the parameter LISTPROMPT is set, the completion code will not ask if the list should be shown. Instead it immediately starts displaying the list, stopping after the first screenful, showing the prompt at the bottom, waiting for a keypress after temporarily switching to the listscroll keymap. Some of the zle functions have a special meaning while scrolling lists:

send-break
: stops listing discarding the key pressed

accept-line, down-history, down-line-or-history
down-line-or-search, vi-down-line-or-history
: scrolls forward one line

complete-word, menu-complete, expand-or-complete
expand-or-complete-prefix, menu-complete-or-expand
: scrolls forward one screenful

Every other character stops listing and immediately processes the key as usual. Any key that is not bound in the listscroll keymap or that is bound to undefined-key is looked up in the keymap currently selected.

As for the ZLS_COLORS and ZLS_COLOURS parameters, LISTPROMPT should not be set directly when using the shell function based completion system. Instead, the list-prompt style should be used.

21.6.3 Menu selection

The zsh/complist module also offers an alternative style of selecting matches from a list, called menu selection, which can be used if the shell is set up to return to the last prompt after showing a completion list (see the ALWAYS_LAST_PROMPT option in Chapter 15 [Options], page 59). It can be invoked directly by the widget menu-select defined by the module. Alternatively, the parameter MENUSELECT can be set to an integer, which gives the minimum number of matches that must be present before menu selection is automatically turned on. This second method requires that menu completion be started, either directly from a widget such as menu-complete, or due to one of the options MENU_COMPLETE or AUTO_MENU being set. If MENUSELECT is set, but is 0, 1 or empty, menu selection will always be started during an ambiguous menu completion.

When using the completion system based on shell functions, the MENUSELECT parameter should not be used (like the ZLS_COLORS and ZLS_COLOURS parameters described above). Instead, the menu style should be used with the select=... keyword.

After menu selection is started, the matches will be listed. If there are more matches than fit on the screen, only the first screenful is shown. The matches to insert into the command line can be selected from this list. In the list one match is highlighted using the value for ma from the ZLS_COLORS or ZLS_COLOURS parameter. The default value for this is '7' which forces the selected match to be highlighted using standout mode on a vt100-compatible terminal. If neither ZLS_COLORS nor ZLS_COLOURS is set, the same terminal control sequence as for the '%S' escape in prompts is used.

If there are more matches than fit on the screen and the parameter MENUPROMPT is set, its value will be shown below the matches. It supports the same escape sequences as LISTPROMPT, but the number of the match or line shown will be that of the one where the mark is placed. If its value is the empty string, a default prompt will be used.

The MENUSCROLL parameter can be used to specify how the list is scrolled. If the parameter is unset, this is done line by line, if it is set to '0' (zero), the list will scroll half the number of lines of the screen. If the value is positive, it gives the number of lines to scroll and if it is negative, the list will be scrolled the number of lines of the screen minus the (absolute) value.

As for the ZLS_COLORS, ZLS_COLOURS and LISTPROMPT parameters, neither MENUPROMPT nor MENUSCROLL should be set directly when using the shell function based completion system. Instead, the select-prompt and select-scroll styles should be used.

The completion code sometimes decides not to show all of the matches in the list. These hidden matches are either matches for which the completion function which added them explicitly requested that they not appear in the list (using the -n option of the compadd builtin command) or they are matches which duplicate a string already in the list (because they differ only in things like prefixes or suffixes that are not displayed). In the list used for menu selection, however, even these matches are shown so that it is possible to select them. To highlight such matches the hi and du capabilities in the ZLS_COLORS and ZLS_COLOURS parameters are supported for hidden matches of the first and second kind, respectively.

Selecting matches is done by moving the mark around using the zle movement functions. When not all matches can be shown on the screen at the same time, the list will scroll up and down when crossing the top or bottom line. The following zle functions have special meaning during menu selection:

accept-line
: accepts the current match and leaves menu selection

send-break
: leaves menu selection and restores the previous contents of the command line

redisplay, clear-screen
: execute their normal function without leaving menu selection

accept-and-hold, accept-and-menu-complete
: accept the currently inserted match and continue selection allowing to select the next match to insert into the line

accept-and-infer-next-history
: accepts the current match and then tries completion with menu selection again; in the case of files this allows one to select a directory and immediately attempt to complete files in it; if there are no matches, a message is shown and one can use undo to go back to completion on the previous level, every other key leaves menu selection (including the other zle functions which are otherwise special during menu selection)

undo
: removes matches inserted during the menu selection by one of the three functions before

down-history, down-line-or-history
vi-down-line-or-history, down-line-or-search
: moves the mark one line down

up-history, up-line-or-history
vi-up-line-or-history, up-line-or-search
: moves the mark one line up

forward-char, vi-forward-char
: moves the mark one column right

backward-char, vi-backward-char
: moves the mark one column left

`forward-word`, `vi-forward-word`
`vi-forward-word-end`, `emacs-forward-word`
: moves the mark one screenful down

`backward-word`, `vi-backward-word`, `emacs-backward-word`
: moves the mark one screenful up

`vi-forward-blank-word`, `vi-forward-blank-word-end`
: moves the mark to the first line of the next group of matches

`vi-backward-blank-word`
: moves the mark to the last line of the previous group of matches

`beginning-of-history`
: moves the mark to the first line

`end-of-history`
: moves the mark to the last line

`beginning-of-buffer-or-history`, `beginning-of-line`
`beginning-of-line-hist`, `vi-beginning-of-line`
: moves the mark to the leftmost column

`end-of-buffer-or-history`, `end-of-line`
`end-of-line-hist`, `vi-end-of-line`
: moves the mark to the rightmost column

`complete-word`, `menu-complete`, `expand-or-complete`
`expand-or-complete-prefix`, `menu-expand-or-complete`
: moves the mark to the next match

`reverse-menu-complete`
: moves the mark to the previous match

All movement functions wrap around at the edges; any other zle function not listed leaves menu selection and executes that function. It is possible to make widgets in the above list do the same by using the form of the widget with a '`.`' in front. For example, the widget '`.accept-line`' has the effect of leaving menu selection and accepting the entire command line.

During this selection the widget uses the keymap `menuselect`. Any key that is not defined in this keymap or that is bound to `undefined-key` is looked up in the keymap currently selected. This is used to ensure that the most important keys used during selection (namely the cursor keys, return, and TAB) have sensible defaults. However, keys in the `menuselect` keymap can be modified directly using the `bindkey` builtin command (see Section 21.19 [The zsh/zle Module], page 208). For example, to make the return key leave menu selection without accepting the match currently selected one could call

```
bindkey -M menuselect '^M' send-break
```

after loading the `zsh/complist` module.

21.7 The zsh/computil Module

The `zsh/computil` module adds several builtin commands that are used by some of the completion functions in the completion system based on shell functions (see Chapter 19 [Completion System], page 127). Except for `compquote` these builtin commands are very specialised and thus not very interesting when writing your own completion functions. In summary, these builtin commands are:

`comparguments`
: This is used by the `_arguments` function to do the argument and command line parsing. Like `compdescribe` it has an option `-i` to do the parsing and initialize some internal state and various options to access the state information to decide what should be completed.

`compdescribe`
: This is used by the `_describe` function to build the displays for the matches and to get the strings to add as matches with their options. On the first call one of the options `-i` or `-I` should be supplied as the first argument. In the first case, display strings without the descriptions will be generated, in the second case, the string used to separate the matches from their descriptions must be given as the second argument and the descriptions (if any) will be shown. All other arguments are like the definition arguments to `_describe` itself.

  Once `compdescribe` has been called with either the `-i` or the `-I` option, it can be repeatedly called with the `-g` option and the names of five arrays as its arguments. This will step through the different sets of matches and store the options in the first array, the strings with descriptions in the second, the matches for these in the third, the strings without descriptions in the fourth, and the matches for them in the fifth array. These are then directly given to `compadd` to register the matches with the completion code.

`compfiles`
: Used by the `_path_files` function to optimize complex recursive filename generation (globbing). It does three things. With the `-p` and `-P` options it builds the glob patterns to use, including the paths already handled and trying to optimize the patterns with respect to the prefix and suffix from the line and the match specification currently used. The `-i` option does the directory tests for the `ignore-parents` style and the `-r` option tests if a component for some of the matches are equal to the string on the line and removes all other matches if that is true.

`compgroups`
: Used by the `_tags` function to implement the internals of the `group-order` style. This only takes its arguments as names of completion groups and creates the groups for it (all six types: sorted and unsorted, both without removing duplicates, with removing all duplicates and with removing consecutive duplicates).

`compquote` [`-p`] *names* ...
: There may be reasons to write completion functions that have to add the matches using the `-Q` option to `compadd` and perform quoting themselves. Instead of interpreting the first character of the `all_quotes` key of the `compstate` special association and using the `q` flag for parameter expansions, one can use this builtin command. The arguments are the names of scalar or array parameters and the values of these parameters are quoted as needed for the innermost quoting level. If the `-p` option is given, quoting is done as if there is some prefix before the values of the parameters, so that a leading equal sign will not be quoted.

  The return value is non-zero in case of an error and zero otherwise.

`comptags`
`comptry`
: These implement the internals of the tags mechanism.

`compvalues`
: Like `comparguments`, but for the `_values` function.

21.8 The zsh/deltochar Module

The `zsh/deltochar` module makes available two ZLE functions:

`delete-to-char`
: Read a character from the keyboard, and delete from the cursor position up to and including the next (or, with repeat count *n*, the *n*th) instance of that character. Negative repeat counts mean delete backwards.

`zap-to-char`
: This behaves like `delete-to-char`, except that the final occurrence of the character itself is not deleted.

21.9 The zsh/example Module

The `zsh/example` module makes available one builtin command:

`example` [`-flags`] [*args ...*]
: Displays the flags and arguments it is invoked with.

The purpose of the module is to serve as an example of how to write a module.

21.10 The zsh/files Module

The `zsh/files` module makes some standard commands available as builtins:

`chgrp` [`-Rs`] *group filename ...*
: Changes group of files specified. This is equivalent to `chown` with a *user-spec* argument of '`:`*group*'.

`chown` [`-Rs`] *user-spec filename ...*
: Changes ownership and group of files specified.

  The *user-spec* can be in four forms:

user	change owner to *user*; do not change group
user`::`	change owner to *user*; do not change group
user`:`	change owner to *user*; change group to *user*'s primary group
user`:`*group*	change owner to *user*; change group to *group*
`:`*group*	do not change owner; change group to *group*

  In each case, the '`:`' may instead be a '`.`'. The rule is that if there is a '`:`' then the separator is '`:`', otherwise if there is a '`.`' then the separator is '`.`', otherwise there is no separator.

  Each of *user* and *group* may be either a username (or group name, as appropriate) or a decimal user ID (group ID). Interpretation as a name takes precedence, if there is an all-numeric username (or group name).

  The `-R` option causes `chown` to recursively descend into directories, changing the ownership of all files in the directory after changing the ownership of the directory itself.

  The `-s` option is a zsh extension to `chown` functionality. It enables paranoid behaviour, intended to avoid security problems involving a `chown` being tricked into affecting files other than the ones intended. It will refuse to follow symbolic links, so that (for example) "`chown luser /tmp/foo/passwd`" can't accidentally chown `/etc/passwd` if `/tmp/foo` happens to be a link to `/etc`. It will also check where it is after leaving directories, so that a recursive chown of a deep directory tree can't end up recursively chowning `/usr` as a result of directories being moved up the tree.

`ln` [`-dfis`] *filename dest*
`ln` [`-dfis`] *filename ... dir*

Creates hard (or, with `-s`, symbolic) links. In the first form, the specified *dest*ination is created, as a link to the specified *filename*. In the second form, each of the *filename*s is taken in turn, and linked to a pathname in the specified *dir*ectory that has the same last pathname component.

Normally, `ln` will not attempt to create hard links to directories. This check can be overridden using the `-d` option. Typically only the super-user can actually succeed in creating hard links to directories. This does not apply to symbolic links in any case.

By default, existing files cannot be replaced by links. The `-i` option causes the user to be queried about replacing existing files. The `-f` option causes existing files to be silently deleted, without querying. `-f` takes precedence.

`mkdir` [`-p`] [`-m` *mode*] *dir ...*

Creates directories. With the `-p` option, non-existing parent directories are first created if necessary, and there will be no complaint if the directory already exists. The `-m` option can be used to specify (in octal) a set of file permissions for the created directories, otherwise mode 777 modified by the current `umask` (see man page umask(2)) is used.

`mv` [`-fi`] *filename dest*
`mv` [`-fi`] *filename ... dir*

Moves files. In the first form, the specified *filename* is moved to the specified *dest*ination. In the second form, each of the *filename*s is taken in turn, and moved to a pathname in the specified *dir*ectory that has the same last pathname component.

By default, the user will be queried before replacing any file that the user cannot write to, but writable files will be silently removed. The `-i` option causes the user to be queried about replacing any existing files. The `-f` option causes any existing files to be silently deleted, without querying. `-f` takes precedence.

Note that this `mv` will not move files across devices. Historical versions of `mv`, when actual renaming is impossible, fall back on copying and removing files; if this behaviour is desired, use `cp` and `rm` manually. This may change in a future version.

`rm` [`-dfirs`] *filename ...*

Removes files and directories specified.

Normally, `rm` will not remove directories (except with the `-r` option). The `-d` option causes `rm` to try removing directories with `unlink` (see man page unlink(2)), the same method used for files. Typically only the super-user can actually succeed in unlinking directories in this way. `-d` takes precedence over `-r`.

By default, the user will be queried before removing any file that the user cannot write to, but writable files will be silently removed. The `-i` option causes the user to be queried about removing any files. The `-f` option causes files to be silently deleted, without querying, and suppresses all error indications. `-f` takes precedence.

The `-r` option causes `rm` to recursively descend into directories, deleting all files in the directory before removing the directory with the `rmdir` system call (see man page rmdir(2)).

The `-s` option is a zsh extension to `rm` functionality. It enables paranoid behaviour, intended to avoid common security problems involving a root-run `rm` being tricked into removing files other than the ones intended. It will refuse to follow symbolic links, so that (for example) "`rm /tmp/foo/passwd`" can't accidentally remove `/etc/passwd` if `/tmp/foo` happens to be a link to `/etc`. It will also check where it is after leaving directories, so that a recursive removal of a deep directory tree can't end up recursively removing `/usr` as a result of directories being moved up the tree.

`rmdir` *dir ...*
: Removes empty directories specified.

`sync`
: Calls the system call of the same name (see man page sync(2)), which flushes dirty buffers to disk. It might return before the I/O has actually been completed.

21.11 The zsh/mapfile Module

The `zsh/mapfile` module provides one special associative array parameter of the same name.

`mapfile`
: This associative array takes as keys the names of files; the resulting value is the content of the file. The value is treated identically to any other text coming from a parameter. The value may also be assigned to, in which case the file in question is written (whether or not it originally existed); or an element may be unset, which will delete the file in question. For example, '`vared mapfile[myfile]`' works as expected, editing the file '`myfile`'.

  When the array is accessed as a whole, the keys are the names of files in the current directory, and the values are empty (to save a huge overhead in memory). Thus `${(k)mapfile}` has the same affect as the glob operator `*(D)`, since files beginning with a dot are not special. Care must be taken with expressions such as `rm ${(k)mapfile}`, which will delete every file in the current directory without the usual '`rm *`' test.

  The parameter `mapfile` may be made read-only; in that case, files referenced may not be written or deleted.

21.11.1 Limitations

Although reading and writing of the file in question is efficiently handled, zsh's internal memory management may be arbitrarily baroque. Thus it should not automatically be assumed that use of `mapfile` represents a gain in efficiency over use of other mechanisms. Note in particular that the whole contents of the file will always reside physically in memory when accessed (possibly multiple times, due to standard parameter substitution operations). In particular, this means handling of sufficiently long files (greater than the machine's swap space, or than the range of the pointer type) will be incorrect.

No errors are printed or flagged for non-existent, unreadable, or unwritable files, as the parameter mechanism is too low in the shell execution hierarchy to make this convenient.

It is unfortunate that the mechanism for loading modules does not yet allow the user to specify the name of the shell parameter to be given the special behaviour.

21.12 The zsh/mathfunc Module

The `zsh/mathfunc` module provides standard mathematical functions for use when evaluating mathematical formulae. The syntax agrees with normal C and FORTRAN conventions, for example,

```
(( f = sin(0.3) ))
```

assigns the sine of 0.3 to the parameter f.

Most functions take floating point arguments and return a floating point value. However, any necessary conversions from or to integer type will be performed automatically by the shell. Apart from `atan` with a second argument and the `abs`, `int` and `float` functions, all functions behave as noted in the manual page for the corresponding C function, except that any arguments out of range for the function in question will be detected by the shell and an error reported.

The following functions take a single floating point argument: `acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `cbrt`, `ceil`, `cos`, `cosh`, `erf`, `erfc`, `exp`, `expm1`, `fabs`, `floor`, `gamma`, `j0`, `j1`, `lgamma`, `log`, `log10`, `log1p`, `logb`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `y0`, `y1`. The `atan` function can optionally take a second argument, in which case it behaves like the C function `atan2`. The `ilogb` function takes a single floating point argument, but returns an integer.

The function `signgam` takes no arguments, and returns an integer, which is the C variable of the same name, as described in man page gamma(3). Note that it is therefore only useful immediately after a call to `gamma` or `lgamma`. Note also that '`signgam()`' and '`signgam`' are distinct expressions.

The following functions take two floating point arguments: `copysign`, `fmod`, `hypot`, `nextafter`.

The following take an integer first argument and a floating point second argument: `jn`, `yn`.

The following take a floating point first argument and an integer second argument: `ldexp`, `scalb`.

The function `abs` does not convert the type of its single argument; it returns the absolute value of either a floating point number or an integer. The functions `float` and `int` convert their arguments into a floating point or integer value (by truncation) respectively.

Note that the C `pow` function is available in ordinary math evaluation as the '`**`' operator and is not provided here.

21.13 The zsh/parameter Module

The `zsh/parameter` module gives access to some of the internal hash tables used by the shell by defining some special parameters.

`options` The keys for this associative array are the names of the options that can be set and unset using the `setopt` and `unsetopt` builtins. The value of each key is either the string `on` if the option is currently set, or the string `off` if the option is unset. Setting a key to one of these strings is like setting or unsetting the option, respectively. Unsetting a key in this array is like setting it to the value `off`.

`commands` This array gives access to the command hash table. The keys are the names of external commands, the values are the pathnames of the files that would be executed when the command would be invoked. Setting a key in this array defines a new entry in this table in the same way as with the `hash` builtin. Unsetting a key as in '`unset "commands[foo]"`' removes the entry for the given key from the command hash table.

`functions`
This associative array maps names of enabled functions to their definitions. Setting a key in it is like defining a function with the name given by the key and the body given by the value. Unsetting a key removes the definition for the function named by the key.

`dis_functions`
Like `functions` but for disabled functions.

`builtins` This associative array gives information about the builtin commands currently enabled. The keys are the names of the builtin commands and the values are either '`undefined`' for builtin commands that will automatically be loaded from a module if invoked or '`defined`' for builtin commands that are already loaded.

`dis_builtins`
Like `builtins` but for disabled builtin commands.

`reswords` This array contains the enabled reserved words.

`dis_reswords`
: Like `reswords` but for disabled reserved words.

`aliases`
: This maps the names of the regular aliases currently enabled to their expansions.

`dis_aliases`
: Like `raliases` but for disabled regular aliases.

`galiases`
: Like `raliases`, but for global aliases.

`dis_galiases`
: Like `galiases` but for disabled global aliases.

`parameters`
: The keys in this associative array are the names of the parameters currently defined. The values are strings describing the type of the parameter, in the same format used by the `t` parameter flag, see Section 13.3 [Parameter Expansion], page 29 . Setting or unsetting keys in this array is not possible.

`modules`
: An associative array giving information about modules. The keys are the names of the modules loaded, registered to be autoloaded, or aliased. The value says which state the named module is in and is one of the strings '`loaded`', '`autoloaded`', or '`alias:`*name*', where *name* is the name the module is aliased to.

  Setting or unsetting keys in this array is not possible.

`dirstack`
: A normal array holding the elements of the directory stack. Note that the output of the `dirs` builtin command includes one more directory, the current working directory.

`history`
: This associative array maps history event numbers to the full history lines.

`historywords`
: A special array containing the words stored in the history.

`jobdirs`
: This associative array maps job numbers to the directories from which the job was started (which may not be the current directory of the job).

`jobtexts`
: This associative array maps job numbers to the texts of the command lines that were used to start the jobs.

`jobstates`
: This associative array gives information about the states of the jobs currently known. The keys are the job numbers and the values are strings of the form '*job-state*:*mark*:*pid*=*state*...'. The *job-state* gives the state the whole job is currently in, one of '`running`', '`suspended`', or '`done`'. The *mark* is '`+`' for the current job, '`-`' for the previous job and empty otherwise. This is followed by one '*pid*=*state*' for every process in the job. The *pid*s are, of course, the process IDs and the *state* describes the state of that process.

`nameddirs`
: This associative array maps the names of named directories to the pathnames they stand for.

`userdirs`
: This associative array maps user names to the pathnames of their home directories.

`funcstack`
: This array contains the names of the functions currently being executed. The first element is the name of the function using the parameter.

21.14 The zsh/sched Module

The `zsh/sched` module makes available one builtin command:

`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.

21.15 The zsh/stat Module

The `zsh/stat` module makes available one builtin command:

`stat` [`-gnNolLtTrs`] [`-f` *fd*] [`-H` *hash*] [`-A` *array*] [`-F` *fmt*] [`+`*element*] [*file ...*]
: The command acts as a front end to the `stat` system call (see man page stat(2)). If the `stat` call fails, the appropriate system error message printed and status 1 is returned. The fields of `struct stat` give information about the files provided as arguments to the command. In addition to those available from the `stat` call, an extra element '`link`' is provided. These elements are:

  `device` The number of the device on which the file resides.

  `inode` The unique number of the file on this device ('*inode*' number).

  `mode` The mode of the file; that is, the file's type and access permissions. With the `-s` option, this will be returned as a string corresponding to the first column in the display of the `ls -l` command.

  `nlink` The number of hard links to the file.

  `uid` The user ID of the owner of the file. With the `-s` option, this is displayed as a user name.

  `gid` The group ID of the file. With the `-s` option, this is displayed as a group name.

  `rdev` The raw device number. This is only useful for special devices.

  `size` The size of the file in bytes.

  `atime`
  `mtime`
  `ctime` The last access, modification and inode change times of the file, respectively, as the number of seconds since midnight GMT on 1st January, 1970. With the `-s` option, these are printed as strings for the local time zone; the format can be altered with the `-F` option, and with the `-g` option the times are in GMT.

  `blksize` The number of bytes in one allocation block on the device on which the file resides.

  `block` The number of disk blocks used by the file.

  `link` If the file is a link and the `-L` option is in effect, this contains the name of the file linked to, otherwise it is empty. Note that if this element is selected ("`stat +link`") then the `-L` option is automatically used.

A particular element may be selected by including its name preceded by a '+' in the option list; only one element is allowed. The element may be shortened to any unique set of leading characters. Otherwise, all elements will be shown for all files.

Options:

`-A` *array* Instead of displaying the results on standard output, assign them to an *array*, one `struct stat` element per array element for each file in order. In this case neither the name of the element nor the name of the files appears in *array* unless the `-t` or `-n` options were given, respectively. If `-t` is given, the element name appears as a prefix to the appropriate array element; if `-n` is given, the file name appears as a separate array element preceding all the others. Other formatting options are respected.

`-H` *hash* Similar to `-A`, but instead assign the values to *hash*. The keys are the elements listed above. If the `-n` option is provided then the name of the file is included in the hash with key `name`.

`-f` *fd* Use the file on file descriptor *fd* instead of named files; no list of file names is allowed in this case.

`-F` *fmt* Supplies a `strftime` (see man page strftime(3)) string for the formatting of the time elements. The `-s` option is implied.

`-g` Show the time elements in the GMT time zone. The `-s` option is implied.

`-l` List the names of the type elements (to standard output or an array as appropriate) and return immediately; options other than `-A` and arguments are ignored.

`-L` Perform an `lstat` (see man page lstat(2)) rather than a `stat` system call. In this case, if the file is a link, information about the link itself rather than the target file is returned. This option is required to make the `link` element useful.

`-n` Always show the names of files. Usually these are only shown when output is to standard output and there is more than one file in the list.

`-N` Never show the names of files.

`-o` If a raw file mode is printed, show it in octal, which is more useful for human consumption than the default of decimal. A leading zero will be printed in this case. Note that this does not affect whether a raw or formatted file mode is shown, which is controlled by the `-r` and `-s` options, nor whether a mode is shown at all.

`-r` Print raw data (the default format) alongside string data (the `-s` format); the string data appears in parentheses after the raw data.

`-s` Print `mode`, `uid`, `gid` and the three time elements as strings instead of numbers. In each case the format is like that of `ls -l`.

`-t` Always show the type names for the elements of `struct stat`. Usually these are only shown when output is to standard output and no individual element has been selected.

`-T` Never show the type names of the `struct stat` elements.

21.16 The zsh/termcap Module

The `zsh/termcap` module makes available one builtin command:

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

The `zsh/termcap` module makes available one parameter:

`termcap` An associative array that maps termcap capability codes to their values.

21.17 The zsh/terminfo Module

The `zsh/terminfo` module makes available one builtin command:

`echoti` *cap*
: Output the terminfo value corresponding to the capability *cap*.

The `zsh/terminfo` module makes available one parameter:

`terminfo` An associative array that maps terminfo capability names to their values.

21.18 The zsh/zftp Module

The `zsh/zftp` module makes available one builtin command:

`zftp` *subcommand* [*args*]
: The `zsh/zftp` module is a client for FTP (file transfer protocol). It is implemented as a builtin to allow full use of shell command line editing, file I/O, and job control mechanisms. Often, users will access it via shell functions providing a more powerful interface; a set is provided with the `zsh` distribution and is described in Chapter 22 [Zftp Function System], page 214. However, the `zftp` command is entirely usable in its own right.

  All commands consist of the command name `zftp` followed by the name of a subcommand. These are listed below. The return status of each subcommand is supposed to reflect the success or failure of the remote operation. See a description of the variable `ZFTP_VERBOSE` for more information on how responses from the server may be printed.

21.18.1 Subcommands

`open` *host* [*user* [*password* [*account*]]]
: Open a new FTP session to *host*, which may be the name of a TCP/IP connected host or an IP number in the standard dot notation. Remaining arguments are passed to the `login` subcommand. Note that if no arguments beyond *host* are supplied, `open` will *not* automatically call `login`. If no arguments at all are supplied, `open` will use the parameters set by the `params` subcommand.

  After a successful open, the shell variables `ZFTP_HOST`, `ZFTP_IP` and `ZFTP_SYSTEM` are available; see 'Variables' below.

`login` [*name* [*password* [*account*]]]
`user` [*name* [*password* [*account*]]]
: Login the user *name* with parameters *password* and *account*. Any of the parameters can be omitted, and will be read from standard input if needed (*name* is always needed). If standard input is a terminal, a prompt for each one will be printed on standard error and *password* will not be echoed. If any of the parameters are not used, a warning message is printed.

  After a successful login, the shell variables `ZFTP_USER`, `ZFTP_ACCOUNT` and `ZFTP_PWD` are available; see 'Variables' below.

  This command may be re-issued when a user is already logged in, and the server will first be reinitialized for a new user.

`params` [*host* [*user* [*password* [*account*]]]]
`params -`
: Store the given parameters for a later `open` command with no arguments. Only those given on the command line will be remembered. If no arguments are given, the parameters currently set are printed, although the password will appear as a line of stars; the return value is one if no parameters were set, zero otherwise.

  Any of the parameters may be specified as a '`?`', which may need to be quoted to protect it from shell expansion. In this case, the appropriate parameter will be read from stdin as with the `login` subcommand, including special handling of *password*. If the '`?`' is followed by a string, that is used as the prompt for reading the parameter instead of the default message (any necessary punctuation and whitespace should be included at the end of the prompt). The first letter of the parameter (only) may be quoted with a '\'; hence an argument `"\\$word"` guarantees that the string from the shell parameter `$word` will be treated literally, whether or not it begins with a '`?`'.

  If instead a single '`-`' is given, the existing parameters, if any, are deleted. In that case, calling `open` with no arguments will cause an error.

  The list of parameters is not deleted after a `close`, however it will be deleted if the `zsh/zftp` module is unloaded.

  For example,

  ```
  zftp params ftp.elsewhere.xx juser '?Password for juser: '
  ```

  will store the host `ftp.elsewhere.xx` and the user `juser` and then prompt the user for the corresponding password with the given prompt.

`test`
: Test the connection; if the server has reported that it has closed the connection (maybe due to a timeout), return status 2; if no connection was open anyway, return status 1; else return status 0. The `test` subcommand is silent, apart from messages printed by the `$ZFTP_VERBOSE` mechanism, or error messages if the connection closes. There is no network overhead for this test.

  The test is only supported on systems with either the `select(2)` or `poll(2)` system calls; otherwise the message '`not supported on this system`' is printed instead.

  The `test` subcommand will automatically be called at the start of any other subcommand for the current session when a connection is open.

`cd` *directory*
: Change the remote directory to *directory*. Also alters the shell variable `ZFTP_PWD`.

`cdup`
: Change the remote directory to the one higher in the directory tree. Note that `cd ..` will also work correctly on non-UNIX systems.

`dir` [*args...*]
: Give a (verbose) listing of the remote directory. The *args* are passed directly to the server. The command's behaviour is implementation dependent, but a UNIX

server will typically interpret *args* as arguments to the `ls` command and with no arguments return the result of '`ls -l`'. The directory is listed to standard output.

`ls` [*args*] Give a (short) listing of the remote directory. With no *args*, produces a raw list of the files in the directory, one per line. Otherwise, up to vagaries of the server implementation, behaves similar to `dir`.

`type` [*type*]
Change the type for the transfer to *type*, or print the current type if *type* is absent. The allowed values are '`A`' (ASCII), '`I`' (Image, i.e. binary), or '`B`' (a synonym for '`I`').

The FTP default for a transfer is ASCII. However, if `zftp` finds that the remote host is a UNIX machine with 8-bit byes, it will automatically switch to using binary for file transfers upon `open`. This can subsequently be overridden.

The transfer type is only passed to the remote host when a data connection is established; this command involves no network overhead.

`ascii` The same as `type A`.

`binary` The same as `type I`.

`mode` [`S` | `B`]
Set the mode type to stream (`S`) or block (`B`). Stream mode is the default; block mode is not widely supported.

`remote` *files...*
`local` [*files...*]
Print the size and last modification time of the remote or local files. If there is more than one item on the list, the name of the file is printed first. The first number is the file size, the second is the last modification time of the file in the format `CCYYMMDDhhmmSS` consisting of year, month, date, hour, minutes and seconds in GMT. Note that this format, including the length, is guaranteed, so that time strings can be directly compared via the `[[` builtin's `<` and `>` operators, even if they are too long to be represented as integers.

Not all servers support the commands for retrieving this information. In that case, the `remote` command will print nothing and return status 2, compared with status 1 for a file not found.

The `local` command (but not `remote`) may be used with no arguments, in which case the information comes from examining file descriptor zero. This is the same file as seen by a `put` command with no further redirection.

`get` *file* [...]
Retrieve all *file*s from the server, concatenating them and sending them to standard output.

`put` *file* [...]
For each *file*, read a file from standard input and send that to the remote host with the given name.

`append` *file* [...]
As `put`, but if the remote *file* already exists, data is appended to it instead of overwriting it.

`getat` *file point*
`putat` *file point*
`appendat` *file point*
Versions of `get`, `put` and `append` which will start the transfer at the given *point* in the remote *file*. This is useful for appending to an incomplete local file. However,

note that this ability is not universally supported by servers (and is not quite the behaviour specified by the standard).

`delete` *file* [...]
Delete the list of files on the server.

`mkdir` *directory*
Create a new directory *directory* on the server.

`rmdir` *directory*
Delete the directory *directory* on the server.

`rename` *old-name new-name*
Rename file *old-name* to *new-name* on the server.

`site` *args...*
Send a host-specific command to the server. You will probably only need this if instructed by the server to use it.

`quote` *args...*
Send the raw FTP command sequence to the server. You should be familiar with the FTP command set as defined in RFC959 before doing this. Useful commands may include `STAT` and `HELP`. Note also the mechanism for returning messages as described for the variable `ZFTP_VERBOSE` below, in particular that all messages from the control connection are sent to standard error.

`close`
`quit` Close the current data connection. This unsets the shell parameters `ZFTP_HOST`, `ZFTP_IP`, `ZFTP_SYSTEM`, `ZFTP_USER`, `ZFTP_ACCOUNT`, `ZFTP_PWD`, `ZFTP_TYPE` and `ZFTP_MODE`.

`session` [*sessname*]
Allows multiple FTP sessions to be used at once. The name of the session is an arbitrary string of characters; the default session is called '`default`'. If this command is called without an argument, it will list all the current sessions; with an argument, it will either switch to the existing session called *sessname*, or create a new session of that name.

Each session remembers the status of the connection, the set of connection-specific shell parameters (the same set as are unset when a connection closes, as given in the description of `close`), and any user parameters specified with the `params` subcommand. Changing to a previous session restores those values; changing to a new session initialises them in the same way as if `zftp` had just been loaded. The name of the current session is given by the parameter `ZFTP_SESSION`.

`rmsession` [*sessname*]
Delete a session; if a name is not given, the current session is deleted. If the current session is deleted, the earliest existing session becomes the new current session, otherwise the current session is not changed. If the session being deleted is the only one, a new session called '`default`' is created and becomes the current session; note that this is a new session even if the session being deleted is also called '`default`'. It is recommended that sessions not be deleted while background commands which use `zftp` are still active.

21.18.2 Parameters

The following shell parameters are used by `zftp`. Currently none of them are special.

`ZFTP_TMOUT`
: Integer. The time in seconds to wait for a network operation to complete before returning an error. If this is not set when the module is loaded, it will be given the default value 60. A value of zero turns off timeouts. If a timeout occurs on the control connection it will be closed. Use a larger value if this occurs too frequently.

`ZFTP_IP`
: Readonly. The IP address of the current connection in dot notation.

`ZFTP_HOST`
: Readonly. The hostname of the current remote server. If the host was opened as an IP number, `ZFTP_HOST` contains that instead; this saves the overhead for a name lookup, as IP numbers are most commonly used when a nameserver is unavailable.

`ZFTP_SYSTEM`
: Readonly. The system type string returned by the server in response to an FTP `SYST` request. The most interesting case is a string beginning `"UNIX Type: L8"`, which ensures maximum compatibility with a local UNIX host.

`ZFTP_TYPE`
: Readonly. The type to be used for data transfers , either '`A`' or '`I`'. Use the **`type`** subcommand to change this.

`ZFTP_USER`
: Readonly. The username currently logged in, if any.

`ZFTP_ACCOUNT`
: Readonly. The account name of the current user, if any. Most servers do not require an account name.

`ZFTP_PWD`
: Readonly. The current directory on the server.

`ZFTP_CODE`
: Readonly. The three digit code of the last FTP reply from the server as a string. This can still be read after the connection is closed, and is not changed when the current session changes.

`ZFTP_REPLY`
: Readonly. The last line of the last reply sent by the server. This can still be read after the connection is closed, and is not changed when the current session changes.

`ZFTP_SESSION`
: Readonly. The name of the current FTP session; see the description of the **`session`** subcommand.

`ZFTP_PREFS`
: A string of preferences for altering aspects of **`zftp`**'s behaviour. Each preference is a single character. The following are defined:

  `P`
  : Passive: attempt to make the remote server initiate data transfers. This is slightly more efficient than sendport mode. If the letter `S` occurs later in the string, **`zftp`** will use sendport mode if passive mode is not available.

  `S`
  : Sendport: initiate transfers by the FTP `PORT` command. If this occurs before any `P` in the string, passive mode will never be attempted.

  `D`
  : Dumb: use only the bare minimum of FTP commands. This prevents the variables `ZFTP_SYSTEM` and `ZFTP_PWD` from being set, and will mean all connections default to ASCII type. It may prevent `ZFTP_SIZE` from being set during a transfer if the server does not send it anyway (many servers do).

If `ZFTP_PREFS` is not set when `zftp` is loaded, it will be set to a default of '`PS`', i.e. use passive mode if available, otherwise fall back to sendport mode.

`ZFTP_VERBOSE`
A string of digits between 0 and 5 inclusive, specifying which responses from the server should be printed. All responses go to standard error. If any of the numbers 1 to 5 appear in the string, raw responses from the server with reply codes beginning with that digit will be printed to standard error. The first digit of the three digit reply code is defined by RFC959 to correspond to:

1. A positive preliminary reply.
2. A positive completion reply.
3. A positive intermediate reply.
4. A transient negative completion reply.
5. A permanent negative completion reply.

It should be noted that, for unknown reasons, the reply 'Service not available', which forces termination of a connection, is classified as 421, i.e. 'transient negative', an interesting interpretation of the word 'transient'.

The code 0 is special: it indicates that all but the last line of multiline replies read from the server will be printed to standard error in a processed format. By convention, servers use this mechanism for sending information for the user to read. The appropriate reply code, if it matches the same response, takes priority.

If `ZFTP_VERBOSE` is not set when `zftp` is loaded, it will be set to the default value `450`, i.e., messages destined for the user and all errors will be printed. A null string is valid and specifies that no messages should be printed.

21.18.3 Functions

`zftp_chpwd`
If this function is set by the user, it is called every time the directory changes on the server, including when a user is logged in, or when a connection is closed. In the last case, `$ZFTP_PWD` will be unset; otherwise it will reflect the new directory.

`zftp_progress`
If this function is set by the user, it will be called during a `get`, `put` or `append` operation each time sufficient data has been received from the host. During a `get`, the data is sent to standard output, so it is vital that this function should write to standard error or directly to the terminal, *not* to standard output.

When it is called with a transfer in progress, the following additional shell parameters are set:

`ZFTP_FILE`
The name of the remote file being transferred from or to.

`ZFTP_TRANSFER`
A `G` for a `get` operation and a `P` for a `put` operation.

`ZFTP_SIZE`
The total size of the complete file being transferred: the same as the first value provided by the `remote` and `local` subcommands for a particular file. If the server cannot supply this value for a remote file being retrieved, it will not be set. If input is from a pipe the value may be incorrect and correspond simply to a full pipe buffer.

`ZFTP_COUNT`
: The amount of data so far transferred; a number between zero and `$ZFTP_SIZE`, if that is set. This number is always available.

The function is initially called with `ZFTP_TRANSFER` set appropriately and `ZFTP_COUNT` set to zero. After the transfer is finished, the function will be called one more time with `ZFTP_TRANSFER` set to `GF` or `PF`, in case it wishes to tidy up. It is otherwise never called twice with the same value of `ZFTP_COUNT`.

Sometimes the progress meter may cause disruption. It is up to the user to decide whether the function should be defined and to use `unfunction` when necessary.

21.18.4 Problems

A connection may not be opened in the left hand side of a pipe as this occurs in a subshell and the file information is not updated in the main shell. In the case of type or mode changes or closing the connection in a subshell, the information is returned but variables are not updated until the next call to `zftp`. Other status changes in subshells will not be reflected by changes to the variables (but should be otherwise harmless).

Deleting sessions while a `zftp` command is active in the background can have unexpected effects, even if it does not use the session being deleted. This is because all shell subprocesses share information on the state of all connections, and deleting a session changes the ordering of that information.

On some operating systems, the control connection is not valid after a fork(), so that operations in subshells, on the left hand side of a pipeline, or in the background are not possible, as they should be. This is presumably a bug in the operating system.

21.19 The zsh/zle Module

The `zsh/zle` module contains the Zsh Line Editor. See Chapter 17 [Zsh Line Editor], page 94.

21.20 The zsh/zleparameter Module

The `zsh/zleparameter` module defines two special parameters that can be used to access internal information of the Zsh Line Editor (see Chapter 17 [Zsh Line Editor], page 94).

`keymaps`
: This array contains the names of the keymaps currently defined.

`widgets`
: This associative array contains one entry per widget defined. The name of the widget is the key and the value gives information about the widget. It is either the string '`builtin`' for builtin widgets, a string of the form '`user:`*name*' for user-defined widgets, where *name* is the name of the shell function implementing the widget, or it is a string of the form '`completion:`*type*`:`*name*', for completion widgets. In the last case *type* is the name of the builtin widgets the completion widget imitates in its behavior and *name* is the name of the shell function implementing the completion widget.

21.21 The zsh/zprof Module

When loaded, the `zsh/zprof` causes shell functions to be profiled. The profiling results can be obtained with the `zprof` builtin command made available by this module. There is no way to turn profiling off other than unloading the module.

`zprof` [`-c`]
: Without the `-c` option, `zprof` lists profiling results to standard output. The format is comparable to that of commands like `gprof`.

  At the top there is a summary listing all functions that were called at least once. This summary is sorted in decreasing order of the amount of time spent in each. The lines contain the number of the function in order, which is used in other parts of the list in suffixes of the form '`[`*num*`]`', then the number of calls made to the function. The next three columns list the time in milliseconds spent in the function and its descendents, the average time in milliseconds spent in the function and its descendents per call and the percentage of time spent in all shell functions used in this function and its descendents. The following three columns give the same information, but counting only the time spent in the function itself. The final column shows the name of the function.

  After the summary, detailed information about every function that was invoked is listed, sorted in decreasing order of the amount of time spent in each function and its descendents. Each of these entries consists of descriptions for the functions that called the function described, the function itself, and the functions that were called from it. The description for the function itself has the same format as in the summary (and shows the same information). The other lines don't show the number of the function at the beginning and have their function named indented to make it easier to distinguish the line showing the function described in the section from the surrounding lines.

  The information shown in this case is almost the same as in the summary, but only refers to the call hierarchy being displayed. For example, for a calling function the column showing the total running time lists the time spent in the described function and its descendents only for the times when it was called from that particular calling function. Likewise, for a called function, this columns lists the total time spent in the called function and its descendents only for the times when it was called from the function described.

  Also in this case, the column showing the number of calls to a function also shows a slash and then the total number of invocations made to the called function.

  As long as the `zsh/zprof` module is loaded, profiling will be done and multiple invocations of the `zprof` builtin command will show the times and numbers of calls since the module was loaded. With the `-c` option, the `zprof` builtin command will reset its internal counters and will not show the listing.)

21.22 The zsh/zpty Module

The `zsh/zpty` module offers one builtin:

`zpty` [`-e`] [`-b`] *name* [*arg* ...]
: The arguments following *name* are concatenated with spaces between, then executed as a command, as if passed to the `eval` builtin. The command runs under a newly assigned pseudo-terminal; this is useful for running commands non-interactively which expect an interactive environment. The *name* is not part of the command, but is used to refer to this command in later calls to `zpty`.

With the `-e` option, the pseudo-terminal is set up so that input characters are echoed.

With the `-b` option, input to and output from the pseudo-terminal are made non-blocking.

`zpty -d` [*names* ...]
: The second form, with the `-d` option, is used to delete commands previously started, by supplying a list of their *name*s. If no *names* are given, all commands are deleted. Deleting a command causes the HUP signal to be sent to the corresponding process.

`zpty -w` [`-n`] *name* [*strings* ...]
: The `-w` option can be used to send the to command *name* the given *strings* as input (separated by spaces). If the `-n` option is *not* given, a newline is added at the end.

If no *strings* are provided, the standard input is copied to the pseudo-terminal; this may stop before copying the full input if the pseudo-terminal is non-blocking.

Note that the command under the pseudo-terminal sees this input as if it were typed, so beware when sending special tty driver characters such as word-erase, line-kill, and end-of-file.

`zpty -r` [`-t`] *name* [*param* [*pattern*]]
: The `-r` option can be used to read the output of the command *name*. With only a *name* argument, the output read is copied to the standard output. Unless the pseudo-terminal is non-blocking, copying continues until the command under the pseudo-terminal exits; when non-blocking, only as much output as is immediately available is copied. The return value is zero if any output is copied.

When also given a *param* argument, at most one line is read and stored in the parameter named *param*. Less than a full line may be read if the pseudo-terminal is non-blocking. The return value is zero if at least one character is stored in *param*.

If a *pattern* is given as well, output is read until the whole string read matches the *pattern*, even in the non-blocking case. The return value is zero if the string read matches the pattern, or if the command has exited but at least one character could still be read. As of this writing, a maximum of one megabyte of output can be consumed this way; if a full megabyte is read without matching the pattern, the return value is non-zero.

In all cases, the return value is non-zero if nothing could be read, and is `2` if this is because the command has finished.

If the `-r` option is combined with the `-t` option, `zpty` tests whether output is available before trying to read. If no output is available, `zpty` immediately returns the value `1`.

`zpty -t` *name*
: The `-t` option without the `-r` option can be used to test whether the command *name* is still running. It returns a zero value if the command is running and a non-zero value otherwise.

`zpty` [`-L`]
: The last form, without any arguments, is used to list the commands currently defined. If the `-L` option is given, this is done in the form of calls to the `zpty` builtin.

21.23 The zsh/zutil Module

The `zsh/zutil` module only adds some builtins:

`zstyle` [`-L`]
`zstyle` [`-e` | `-` | `--`] *pattern style strings ...*
`zstyle -d` [*pattern* [*styles ...*]]
`zstyle -g` *name* [*pattern* [*style*]]
`zstyle -abs` *context style name* [*sep*]
`zstyle -Tt` *context style* [*strings ...*]
`zstyle -m` *context style pattern*

This builtin command is used to define and lookup styles. Styles are pairs of names and values, where the values consist of any number of strings. They are stored together with patterns and lookup is done by giving a string, called the 'context', which is compared to the patterns. The definition stored for the first matching pattern will be returned.

For ordering of comparisons, patterns are searched from most specific to least specific, and patterns that are equally specific keep the order in which they were defined. A pattern is considered to be more specific than another if it contains more components (substrings separated by colons) or if the patterns for the components are more specific, where simple strings are considered to be more specific than patterns and complex patterns are considered to be more specific than the pattern '`*`'.

The first form (without arguments) lists the definitions in the order `zstyle` will test them. If the `-L` option is given, listing is done in the form of calls to `zstyle`. Forms with arguments:

`zstyle` [`-` | `--` | `-e`] *pattern style strings ...*
: Defines the given *style* for the *pattern* with the *strings* as the value. If the `-e` option is given, the *strings* will be concatenated (separated by spaces) and the resulting string will be evaluated (in the same way as it is done by the `eval` builtin command) when the style is looked up. In this case the parameter '`reply`' must be assigned to set the strings returned after the evaluation. Before evaluating the value, `reply` is unset, and if it is still unset after the evaluation, the style is treated as if it were not set.

`zstyle -d` [*pattern* [*styles ...*]]
: Delete style definitions. Without arguments all definitions are deleted, with a *pattern* all definitions for that pattern are deleted and if any *styles* are given, then only those styles are deleted for the *pattern*.

`zstyle -g` *name* [*pattern* [*style*]]
: Retrieve a style definition. The *name* is used as the name of an array in which the results are stored. Without any further arguments, all *patterns* defined are returned. With a *pattern* the styles defined for that pattern are returned and with both a *pattern* and a *style*, the value strings of that combination is returned.

The other forms can be used to look up or test patterns.

`zstyle -s` *context style name* [*sep*]
: The parameter *name* is set to the value of the style interpreted as a string. If the value contains several strings they are concatenated with spaces (or with the *sep* string if that is given) between them.

`zstyle -b` *context style name*
: The value is stored in *name* as a boolean, i.e. as the string '`yes`' if the value has only one string and that string is equal to one of '`yes`', '`true`', '`on`', or '`1`'. If the value is any other string or has more than one string, the parameter is set to '`no`'.

`zstyle -a` *context style name*
: The value is stored in *name* as an array. If *name* is declared as an associative array, the first, third, etc. strings are used as the keys and the other strings are used as the values.

`zstyle -t` *context style* [*strings ...*]
`zstyle -T` *context style* [*strings ...*]
: Test the value of a style, i.e. the `-t` option only returns a status (sets `$?`). Without any *strings* the return status is zero if the style is defined for at least one matching pattern, has only one string in its value, and that is equal to one of '`true`', '`yes`', '`on`' or '`1`'. If any *strings* are given the status is zero if and only if at least one of the *strings* is equal to at least one of the strings in the value. If the style is not defined, the status is `2`.

  The `-T` option tests the values of the style like `-t`, but it returns zero (rather than `2`) if the style is not defined for any matching pattern.

`zstyle -m` *context style pattern*
: Match a value. Returns status zero if the *pattern* matches at least one of the strings in the value.

`zformat -f` *param format specs ...*
`zformat -a` *array sep specs ...*
: This builtin provides two different forms of formatting. The first form is selected with the `-f` option. In this case the *format* string will be modified by replacing sequences starting with a percent sign in it with strings from the *specs*. Each *spec* should be of the form '*char*`:`*string*' which will cause every appearance of the sequence '`%`*char*' in *format* to be replaced by the *string*. The '`%`' sequence may also contain optional minimum and maximum field width specifications between the '`%`' and the '*char*' in the form '`%`*min*`.`*max*`c`', i.e. the minimum field width is given first and if the maximum field width is used, it has to be preceded by a dot. Specifying a minimum field width makes the result be padded with spaces to the right if the *string* is shorter than the requested width. Padding to the left can be achieved by giving a negative minimum field width. If a maximum field width is specified, the *string* will be truncated after that many characters. After all '`%`' sequences for the given *specs* have been processed, the resulting string is stored in the parameter *param*.

  The second form, using the `-a` option, can be used for aligning strings. Here, the *specs* are of the form '*left*`:`*right*' where '*left*' and '*right*' are arbitrary strings. These strings are modified by replacing the colons by the *sep* string and padding the *left* strings with spaces to the right so that the *sep* strings in the result (and hence the *right* strings after them) are all aligned if the strings are printed below each other. All strings without a colon are left unchanged and all strings with an empty *right* string have the trailing colon removed. In both cases the lengths of the strings are not used to determine how the other strings are to be aligned. The resulting strings are stored in the *array*.

`zregexparse`
: This implements some internals of the `_regex_arguments` function.

`zparseopts` [`-D`] [`-K`] [`-E`] [`-a` *array*] [`-A` *assoc*] *specs*
: This builtin simplifies the parsing of options in positional parameters, i.e. the set of arguments given by `$*`. Each *spec* describes one option and must be of the form '*opt*[`=`*array*]'. If an option described by *opt* is found in the positional parameters it is copied into the *array* specified with the `-a` option; if the optional '`=`*array*' is given, it is instead copied into that array.

Note that it is an error to give any *spec* without an '=*array*' unless one of the `-a` or `-A` options is used.

Unless the `-E` option is given, parsing stops at the first string that isn't described by one of the *specs*. Even with `-E`, parsing always stops at a positional parameter equal to '`-`' or '`--`'.

The *opt* description must be one of the following. Any of the special characters can appear in the option name provided it is preceded by a backslash.

name
name`+` The *name* is the name of the option without the leading '`-`'. To specify a GNU-style long option, one of the usual two leading '`-`' must be included in *name*; for example, a '`--file`' option is represented by a *name* of '`-file`'.

If a '`+`' appears after *name*, the option is appended to *array* each time it is found in the positional parameters; without the '`+`' only the *last* occurrence of the option is preserved.

If one of these forms is used, the option takes no argument, so parsing stops if the next positional parameter does not also begin with '`-`' (unless the `-E` option is used).

name`:`
name`:-`
name`::` If one or two colons are given, the option takes an argument; with one colon, the argument is mandatory and with two colons it is optional. The argument is appended to the *array* after the option itself.

An optional argument is put into the same array element as the option name (note that this makes empty strings as arguments indistinguishable). A mandatory argument is added as a separate element unless the '`:-`' form is used, in which case the argument is put into the same element.

A '`+`' as described above may appear between the *name* and the first colon.

The options of `zparseopts` itself are:

`-a` *array* As described above, this names the default array in which to store the recognised options.

`-A` *assoc* If this is given, the options and their values are also put into an associative array with the option names as keys and the arguments (if any) as the values.

`-D` If this option is given, all options found are removed from the positional parameters of the calling shell or shell function, up to but not including any not described by the *specs*. This is similar to using the `shift` builtin.

`-K` With this option, the arrays specified with the `-a` and `-A` options and with the '=*array*' forms are kept unchanged when none of the *specs* for them is used. This allows assignment of default values to them before calling `zparseopts`.

`-E` This changes the parsing rules to *not* stop at the first string that isn't described by one of the *spec*s. It can be used to test for or (if used together with `-D`) extract options and their arguments, ignoring all other options and arguments that may be in the positional parameters.

For example,

```
set -- -a -bx -c y -cz baz -cend
zparseopts a=foo b:=bar c+:=bar
```

will have the effect of

```
foo=(-a)
bar=(-b x -c y -c z)
```

The arguments from '`baz`' on will not be used.

As an example for the `-E` option, consider:

```
set -- -a x -b y -c z arg1 arg2
zparseopts -E -D b:=bar
```

will have the effect of

```
bar=(-b y)
set -- -a x -c z arg1 arg2
```

I.e., the option `-b` and its arguments are taken from the positional parameters and put into the array `bar`.

22 Zftp Function System

22.1 Description

This describes the set of shell functions supplied with the source distribution as an interface to the `zftp` builtin command, allowing you to perform FTP operations from the shell command line or within functions or scripts. The interface is similar to a traditional FTP client (e.g. the `ftp` command itself, see man page ftp(1)), but as it is entirely done within the shell all the familiar completion, editing and globbing features, and so on, are present, and macros are particularly simple to write as they are just ordinary shell functions.

The prerequisite is that the `zftp` command, as described in Section 21.18 [The zsh/zftp Module], page 202 , must be available in the version of `zsh` installed at your site. If the shell is configured to load new commands at run time, it probably is: typing '`zmodload zsh/zftp`' will make sure (if that runs silently, it has worked). If this is not the case, it is possible `zftp` was linked into the shell anyway: to test this, type '`which zftp`' and if `zftp` is available you will get the message '`zftp: shell built-in command`'.

Commands given directly with `zftp` builtin may be interspersed between the functions in this suite; in a few cases, using `zftp` directly may cause some of the status information stored in shell parameters to become invalid. Note in particular the description of the variables `$ZFTP_TMOUT`, `$ZFTP_PREFS` and `$ZFTP_VERBOSE` for `zftp`.

22.2 Installation

You should make sure all the functions from the `Functions/Zftp` directory of the source distribution are available; they all begin with the two letters '`zf`'. They may already have been installed on your system; otherwise, you will need to find them and copy them. The directory should appear as one of the elements of the `$fpath` array (this should already be the case if they were installed), and at least the function `zfinit` should be autoloaded; it will autoload the rest. Finally, to initialize the use of the system you need to call the `zfinit` function. The following code in your `.zshrc` will arrange for this; assume the functions are stored in the directory `~/myfns`:

```
fpath=(~/myfns $fpath)
autoload -U zfinit
zfinit
```

Note that `zfinit` assumes you are using the `zmodload` method to load the `zftp` command. If it is already built into the shell, change `zfinit` to `zfinit -n`. It is helpful (though not essential) if the call to `zfinit` appears after any code to initialize the new completion system, else unnecessary `compctl` commands will be given.

22.3 Functions

The sequence of operations in performing a file transfer is essentially the same as that in a standard FTP client. Note that, due to a quirk of the shell's `getopts` builtin, for those functions that handle options you must use '`--`' rather than '`-`' to ensure the remaining arguments are treated literally (a single '`-`' is treated as an argument).

22.3.1 Opening a connection

`zfparams [` *host* `[` *user* `[` *password* `... ] ] ]`
: Set or show the parameters for a future `zfopen` with no arguments. If no arguments are given, the current parameters are displayed (the password will be shown as a line of asterisks). If a host is given, and either the *user* or *password* is not, they will be prompted for; also, any parameter given as '`?`' will be prompted for, and if the '`?`' is followed by a string, that will be used as the prompt. As `zfopen` calls `zfparams` to store the parameters, this usually need not be called directly.

  A single argument '`-`' will delete the stored parameters. This will also cause the memory of the last directory (and so on) on the other host to be deleted.

`zfopen [ -1 ] [` *host* `[` *user* `[` *password* `[` *account* `] ] ] ]`
: If *host* is present, open a connection to that host under username *user* with password *password* (and, on the rare occasions when it is necessary, account *account*). If a necessary parameter is missing or given as '`?`' it will be prompted for. If *host* is not present, use a previously stored set of parameters.

  If the command was successful, and the terminal is compatible with `xterm` or is `sun-cmd`, a summary will appear in the title bar, giving the local `host:directory` and the remote `host:directory`; this is handled by the function `zftp_chpwd`, described below.

  Normally, the *host*, *user* and *password* are internally recorded for later re-opening, either by a `zfopen` with no arguments, or automatically (see below). With the option '`-1`', no information is stored. Also, if an open command with arguments failed, the parameters will not be retained (and any previous parameters will also be deleted). A `zfopen` on its own, or a `zfopen -1`, never alters the stored parameters.

  Both `zfopen` and `zfanon` (but not `zfparams`) understand URLs of the form `ftp://`*host*`/`*path*... as meaning to connect to the *host*, then change directory to *path* (which must be a directory, not a file). The '`ftp://`' can be omitted; the trailing '`/`' is enough to trigger recognition of the *path*. Note prefixes other than '`ftp:`' are not recognized, and that all characters after the first slash beyond *host* are significant in *path*.

`zfanon [ -1 ]` *host*
: Open a connection *host* for anonymous FTP. The username used is '`anonymous`'. The password (which will be reported the first time) is generated as *user*`@`*host*; this is then stored in the shell parameter `$EMAIL_ADDR` which can alternatively be set manually to a suitable string.

22.3.2 Directory management

`zfcd [` *dir* `]`
`zfcd -`
`zfcd` *old new*

Change the current directory on the remote server: this is implemented to have many of the features of the shell builtin `cd`.

In the first form with *dir* present, change to the directory *dir*. The command '`zfcd ..`' is treated specially, so is guaranteed to work on non-UNIX servers (note this is handled internally by `zftp`). If *dir* is omitted, has the effect of '`zfcd ~`'.

The second form changes to the directory previously current.

The third form attempts to change the current directory by replacing the first occurrence of the string *old* with the string *new* in the current directory.

Note that in this command, and indeed anywhere a remote filename is expected, the string which on the local host corresponds to '`~`' is converted back to a '`~`' before being passed to the remote machine. This is convenient because of the way expansion is performed on the command line before `zfcd` receives a string. For example, suppose the command is '`zfcd ~/foo`'. The shell will expand this to a full path such as '`zfcd /home/user2/pws/foo`'. At this stage, `zfcd` recognises the initial path as corresponding to '`~`' and will send the directory to the remote host as `~/foo`, so that the '`~`' will be expanded by the server to the correct remote host directory. Other named directories of the form '`~name`' are not treated in this fashion.

`zfhere` Change directory on the remote server to the one corresponding to the current local directory, with special handling of '`~`' as in `zfcd`. For example, if the current local directory is `~/foo/bar`, then `zfhere` performs the effect of '`zfcd ~/foo/bar`'.

`zfdir [ -rfd ] [ - ] [` *dir-options* `] [` *dir* `]`

Produce a long directory listing. The arguments *dir-options* and *dir* are passed directly to the server and their effect is implementation dependent, but specifying a particular remote directory *dir* is usually possible. The output is passed through a pager given by the environment variable `$PAGER` or defaulting to '`more`'.

The directory is usually cached for re-use. In fact, two caches are maintained. One is for use when there is no *dir-options* or *dir*, i.e. a full listing of the current remote directory; it is flushed when the current remote directory changes. The other is kept for repeated use of `zfdir` with the same arguments; for example, repeated use of '`zfdir /pub/gnu`' will only require the directory to be retrieved on the first call. Alternatively, this cache can be re-viewed with the `-r` option. As relative directories will confuse `zfdir`, the `-f` option can be used to force the cache to be flushed before the directory is listed. The option `-d` will delete both caches without showing a directory listing; it will also delete the cache of file names in the current remote directory, if any.

`zfls [` *ls-options* `] [` *dir* `]`

List files on the remote server. With no arguments, this will produce a simple list of file names for the current remote directory. Any arguments are passed directly to the server. No pager and no caching is used.

22.3.3 Status commands

`zftype` [*type*]
: With no arguments, show the type of data to be transferred, usually ASCII or binary. With an argument, change the type: the types '`A`' or '`ASCII`' for ASCII data and '`B`' or '`BINARY`', '`I`' or '`IMAGE`' for binary data are understood case-insensitively.

`zfstat` [-v]
: Show the status of the current or last connection, as well as the status of some of `zftp`'s status variables. With the `-v` option, a more verbose listing is produced by querying the server for its version of events, too.

22.3.4 Retrieving files

The commands for retrieving files all take at least two options. `-G` suppresses remote filename expansion which would otherwise be performed (see below for a more detailed description of that). `-t` attempts to set the modification time of the local file to that of the remote file: this requires version 5 of `perl`, see the description of the function `zfrtime` below for more information.

`zfget [ -Gtc ]` *file1* `...`
: Retrieve all the listed files *file1* ... one at a time from the remote server. If a file contains a '`/`', the full name is passed to the remote server, but the file is stored locally under the name given by the part after the final '`/`'. The option `-c` (cat) forces all files to be sent as a single stream to standard output; in this case the `-t` option has no effect.

`zfuget [ -Gvst ]` *file1* `...`
: As `zfget`, but only retrieve files where the version on the remote server is newer (has a later modification time), or where the local file does not exist. If the remote file is older but the files have different sizes, or if the sizes are the same but the remote file is newer, the user will usually be queried. With the option `-s`, the command runs silently and will always retrieve the file in either of those two cases. With the option `-v`, the command prints more information about the files while it is working out whether or not to transfer them.

`zfcget [ -Gt ]` *file1* `...`
: As `zfget`, but if any of the local files exists, and is shorter than the corresponding remote file, the command assumes that it is the result of a partially completed transfer and attempts to transfer the rest of the file. This is useful on a poor connection which keeps failing.

  Note that this requires a commonly implemented, but non-standard, version of the FTP protocol, so is not guaranteed to work on all servers.

`zfgcp [ -Gt ]` *remote-file local-file*
`zfgcp [ -Gt ]` *rfile1* `...` *ldir*
: This retrieves files from the remote server with arguments behaving similarly to the `cp` command.

  In the first form, copy *remote-file* from the server to the local file *local-file*.

  In the second form, copy all the remote files *rfile1* ... into the local directory *ldir* retaining the same basenames. This assumes UNIX directory semantics.

22.3.5 Sending files

`zfput [ -r ]` *file1 ...*
: Send all the *file1* ... given separately to the remote server. If a filename contains a '`/`', the full filename is used locally to find the file, but only the basename is used for the remote file name.

  With the option `-r`, if any of the *files* are directories they are sent recursively with all their subdirectories, including files beginning with '`.`'. This requires that the remote machine understand UNIX file semantics, since '`/`' is used as a directory separator.

`zfuput [ -vs ]` *file1 ...*
: As `zfput`, but only send files which are newer than their local equivalents, or if the remote file does not exist. The logic is the same as for `zfuget`, but reversed between local and remote files.

`zfcput` *file1 ...*
: As `zfput`, but if any remote file already exists and is shorter than the local equivalent, assume it is the result of an incomplete transfer and send the rest of the file to append to the existing part. As the FTP append command is part of the standard set, this is in principle more likely to work than `zfcget`.

`zfpcp` *local-file remote-file*
`zfpcp` *lfile1 ... rdir*
: This sends files to the remote server with arguments behaving similarly to the `cp` command.

  With two arguments, copy *local-file* to the server as *remote-file*.

  With more than two arguments, copy all the local files *lfile1* ... into the existing remote directory *rdir* retaining the same basenames. This assumes UNIX directory semantics.

  A problem arises if you attempt to use `zfpcp` *lfile1 rdir*, i.e. the second form of copying but with two arguments, as the command has no simple way of knowing if *rdir* corresponds to a directory or a filename. It attempts to resolve this in various ways. First, if the *rdir* argument is '`.`' or '`..`' or ends in a slash, it is assumed to be a directory. Secondly, if the operation of copying to a remote file in the first form failed, and the remote server sends back the expected failure code 553 and a reply including the string '`Is a directory`', then `zfpcp` will retry using the second form.

22.3.6 Closing the connection

`zfclose` Close the connection.

22.3.7 Session management

`zfsession` [`-lvod`] [*sessname*]
: Allows you to manage multiple FTP sessions at once. By default, connections take place in a session called '`default`'; by giving the command '`zfsession` *sessname*' you can change to a new or existing session with a name of your choice. The new session remembers its own connection, as well as associated shell parameters, and also the host/user parameters set by `zfparams`. Hence you can have different sessions set up to connect to different hosts, each remembering the appropriate host, user and password.

  With no arguments, `zfsession` prints the name of the current session; with the option `-l` it lists all sessions which currently exist, and with the option `-v` it gives

a verbose list showing the host and directory for each session, where the current session is marked with an asterisk. With `-o`, it will switch to the most recent previous session.

With `-d`, the given session (or else the current one) is removed; everything to do with it is completely forgotten. If it was the only session, a new session called '`default`' is created and made current. It is safest not to delete sessions while background commands using `zftp` are active.

`zftransfer` *sess1*`:`*file1* *sess2*`:`*file2*
: Transfer files between two sessions; no local copy is made. The file is read from the session *sess1* as *file1* and written to session *sess1* as file *file2*; *file1* and *file2* may be relative to the current directories of the session. Either *sess1* or *sess2* may be omitted (though the colon should be retained if there is a possibility of a colon appearing in the file name) and defaults to the current session; *file2* may be omitted or may end with a slash, in which case the basename of *file1* will be added. The sessions *sess1* and *sess2* must be distinct.

  The operation is performed using pipes, so it is required that the connections still be valid in a subshell, which is not the case under some versions operating systems, presumably due to a system bug.

22.3.8 Bookmarks

The two functions `zfmark` and `zfgoto` allow you to 'bookmark' the present location (host, user and directory) of the current FTP connection for later use. The file to be used for storing and retrieving bookmarks is given by the parameter `$ZFTP_BMFILE`; if not set when one of the two functions is called, it will be set to the file `.zfbkmarks` in the directory where your zsh startup files live (usually `~`).

`zfmark [` *bookmark* `]`
: If given an argument, mark the current host, user and directory under the name *bookmark* for later use by `zfgoto`. If there is no connection open, use the values for the last connection immediately before it was closed; it is an error if there is none. Any existing bookmark under the same name will be silently replaced.

  If not given an argument, list the existing bookmarks and the points to which they refer in the form *user*`@`*host*`:`*directory*; this is the format in which they are stored, and the file may be edited directly.

`zfgoto [ -n ]` *bookmark*
: Return to the location given by *bookmark*, as previously set by `zfmark`. If the location has user '`ftp`' or '`anonymous`', open the connection with `zfanon`, so that no password is required. If the user and host parameters match those stored for the current session, if any, those will be used, and again no password is required. Otherwise a password will be prompted for.

  With the option `-n`, the bookmark is taken to be a nickname stored by the `ncftp` program in its bookmark file, which is assumed to be `~/.ncftp/bookmarks`. The function works identically in other ways. Note that there is no mechanism for adding or modifying `ncftp` bookmarks from the zftp functions.

22.3.9 Other functions

Mostly, these functions will not be called directly (apart from `zfinit`), but are described here for completeness. You may wish to alter `zftp_chpwd` and `zftp_progress`, in particular.

`zfinit [ -n ]`
: As described above, this is used to initialize the zftp function system. The `-n` option should be used if the zftp command is already built into the shell.

`zfautocheck [ -dn ]`
: This function is called to implement automatic reopening behaviour, as described in more detail below. The options must appear in the first argument; `-n` prevents the command from changing to the old directory, while `-d` prevents it from setting the variable `do_close`, which it otherwise does as a flag for automatically closing the connection after a transfer. The host and directory for the last session are stored in the variable `$zflastsession`, but the internal host/user/password parameters must also be correctly set.

`zfcd_match` *prefix suffix*
: This performs matching for completion of remote directory names. If the remote server is UNIX, it will attempt to persuade the server to list the remote directory with subdirectories marked, which usually works but is not guaranteed. On other hosts it simply calls `zfget_match` and hence completes all files, not just directories. On some systems, directories may not even look like filenames.

`zfget_match` *prefix suffix*
: This performs matching for completion of remote filenames. It caches files for the current directory (only) in the shell parameter `$zftp_fcache`. It is in the form to be called by the `-K` option of `compctl`, but also works when called from a widget-style completion function with *prefix* and *suffix* set appropriately.

`zfrglob` *varname*
: Perform remote globbing, as describes in more detail below. *varname* is the name of a variable containing the pattern to be expanded; if there were any matches, the same variable will be set to the expanded set of filenames on return.

`zfrtime` *lfile rfile* `[` *time* `]`
: Set the local file *lfile* to have the same modification time as the remote file *rfile*, or the explicit time *time* in FTP format `CCYYMMDDhhmmSS` for the GMT timezone.

  Currently this requires `perl` version 5 to perform the conversion from GMT to local time. This is unfortunately difficult to do using shell code alone.

`zftp_chpwd`
: This function is called every time a connection is opened, or closed, or the remote directory changes. This version alters the title bar of an `xterm`-compatible or `sun-cmd` terminal emulator to reflect the local and remote hostnames and current directories. It works best when combined with the function `chpwd`. In particular, a function of the form

```
chpwd() {
  if [[ -n $ZFTP_USER ]]; then
    zftp_chpwd
  else
    # usual chpwd e.g put host:directory in title bar
  fi
}
```

  fits in well.

`zftp_progress`
: This function shows the status of the transfer. It will not write anything unless the output is going to a terminal; however, if you transfer files in the background, you should turn off progress reports by hand using '`zstyle ':zftp:*' progress`

`none`'. Note also that if you alter it, any output *must* be to standard error, as standard output may be a file being received. The form of the progress meter, or whether it is used at all, can be configured without altering the function, as described in the next section.

`zffcache` This is used to implement caching of files in the current directory for each session separately. It is used by `zfget_match` and `zfrglob`.

22.4 Miscellaneous Features

22.4.1 Configuration

Various styles are available using the standard shell style mechanism, described in Section 21.23 [The zsh/zutil Module], page 210. Briefly, the command '`zstyle ':zftp:*'` *style value* ...'. defines the *style* to have value *value* (more than one may be given, although that is not useful in the cases described here). These values will then be used throughout the zftp function system. For more precise control, the first argument, which gives a context in which the style applies, can be modified to include a particular function, as for example '`:zftp:zfget`': the style will then have the given value only in the `zfget` function. Values for the same style in different contexts may be set; the most specific function will be used, where strings are held to be more specific than patterns, and longer patterns and shorter patterns. Note that only the top level function name, as called by the user, is used; calling of lower level functions is transparent to the user. Hence modifications to the title bar in `zftp_chpwd` use the contexts `:zftp:zfopen`, `:zftp:zfcd`, etc., depending where it was called from. The following styles are understood:

`progress` Controls the way that `zftp_progress` reports on the progress of a transfer. If empty, unset, or '`none`', no progress report is made; if '`bar`' a growing bar of inverse video is shown; if '`percent`' (or any other string, though this may change in future), the percentage of the file transferred is shown. The bar meter requires that the width of the terminal be available via the `$COLUMNS` parameter (normally this is set automatically). If the size of the file being transferred is not available, `bar` and `percent` meters will simply show the number of bytes transferred so far.

When `zfinit` is run, if this style is not defined for the context `:zftp:*`, it will be set to 'bar'.

`update` Specifies the minimum time interval between updates of the progress meter in seconds. No update is made unless new data has been received, so the actual time interval is limited only by `$ZFTP_TIMEOUT`.

As described for `progress`, `zfinit` will force this to default to 1.

`remote-glob`
If set to '1', 'yes' or 'true', filename generation (globbing) is performed on the remote machine instead of by zsh itself; see below.

`titlebar` If set to '1', 'yes' or 'true', `zftp_chpwd` will put the remote host and remote directory into the titlebar of terminal emulators such as xterm or sun-cmd that allow this.

As described for `progress`, `zfinit` will force this to default to 1.

`chpwd` If set to '1' 'yes' or 'true', `zftp_chpwd` will call the function `chpwd` when a connection is closed. This is useful if the remote host details were put into the terminal title bar by `zftp_chpwd` and your usual `chpwd` also modifies the title bar.

When `zfinit` is run, it will determine whether `chpwd` exists and if so it will set the default value for the style to 1 if none exists already.

Note that there is also an associative array `zfconfig` which contains values used by the function system. This should not be modified or overwritten.

22.4.2 Remote globbing

The commands for retrieving files usually perform filename generation (globbing) on their arguments; this can be turned off by passing the option `-G` to each of the commands. Normally this operates by retrieving a complete list of files for the directory in question, then matching these locally against the pattern supplied. This has the advantage that the full range of zsh patterns (respecting the setting of the option `EXTENDED_GLOB`) can be used. However, it means that the directory part of a filename will not be expanded and must be given exactly. If the remote server does not support the UNIX directory semantics, directory handling is problematic and it is recommended that globbing only be used within the current directory. The list of files in the current directory, if retrieved, will be cached, so that subsequent globs in the same directory without an intervening `zfcd` are much faster.

If the `remote-glob` style (see above) is set, globbing is instead performed on the remote host: the server is asked for a list of matching files. This is highly dependent on how the server is implemented, though typically UNIX servers will provide support for basic glob patterns. This may in some cases be faster, as it avoids retrieving the entire list of directory contents.

22.4.3 Automatic and temporary reopening

As described for the `zfopen` command, a subsequent `zfopen` with no parameters will reopen the connection to the last host (this includes connections made with the `zfanon` command). Opened in this fashion, the connection starts in the default remote directory and will remain open until explicitly closed.

Automatic re-opening is also available. If a connection is not currently open and a command requiring a connection is given, the last connection is implicitly reopened. In this case the directory which was current when the connection was closed again becomes the current directory (unless, of course, the command given changes it). Automatic reopening will also take place if the connection was close by the remote server for whatever reason (e.g. a timeout). It is not available if the `-1` option to `zfopen` or `zfanon` was used.

Furthermore, if the command issued is a file transfer, the connection will be closed after the transfer is finished, hence providing a one-shot mode for transfers. This does not apply to directory changing or listing commands; for example a `zfdir` may reopen a connection but will leave it open. Also, automatic closure will only ever happen in the same command as automatic opening, i.e a `zfdir` directly followed by a `zfget` will never close the connection automatically.

Information about the previous connection is given by the `zfstat` function. So, for example, if that reports:

```
Session:        default
Not connected.
Last session:   ftp.bar.com:/pub/textfiles
```

then the command `zfget file.txt` will attempt to reopen a connection to `ftp.bar.com`, retrieve the file `/pub/textfiles/file.txt`, and immediately close the connection again. On the other hand, `zfcd ..` will open the connection in the directory `/pub` and leave it open.

Note that all the above is local to each session; if you return to a previous session, the connection for that session is the one which will be reopened.

22.4.4 Completion

Completion of local and remote files, directories, sessions and bookmarks is supported. The older, `compctl`-style completion is defined when `zfinit` is called; support for the new widget-based completion system is provided in the function `Completion/Zsh/Command/_zftp`, which should be installed with the other functions of the completion system and hence should automatically be available.

23 User Contributions

23.1 Description

The Zsh source distribution includes a number of items contributed by the user community. These are not inherently a part of the shell, and some may not be available in every zsh installation. The most significant of these are documented here. For documentation on other contributed items such as shell functions, look for comments in the function source files.

23.2 Utilities

23.2.1 Accessing On-Line Help

The key sequence `ESC h` is normally bound by ZLE to execute the `run-help` widget (see Chapter 17 [Zsh Line Editor], page 94). This invokes the `run-help` command with the command word from the current input line as its argument. By default, `run-help` is an alias for the `man` command, so this often fails when the command word is a shell builtin or a user-defined function. By redefining the `run-help` alias, one can improve the on-line help provided by the shell.

The `helpfiles` utility, found in the `Util` directory of the distribution, is a Perl program that can be used to process the zsh manual to produce a separate help file for each shell builtin and for many other shell features as well. The autoloadable `run-help` function, found in `Functions/Misc`, searches for these helpfiles and performs several other tests to produce the most complete help possible for the command.

There may already be a directory of help files on your system; look in `/usr/share/zsh` or `/usr/local/share/zsh` and subdirectories below those, or ask your system administrator.

To create your own help files with `helpfiles`, choose or create a directory where the individual command help files will reside. For example, you might choose `~/zsh_help`. If you unpacked the zsh distribution in your home directory, you would use the commands:

```
mkdir ~/zsh_help
cd ~/zsh_help
man zshall | colcrt - | \
perl ~/zsh-4.0.3/Util/helpfiles
```

Next, to use the `run-help` function, you need to add lines something like the following to your `.zshrc` or equivalent startup file:

```
unalias run-help
autoload run-help
HELPDIR=~/zsh_help
```

The `HELPDIR` parameter tells `run-help` where to look for the help files. If your system already has a help file directory installed, set `HELPDIR` to the path of that directory instead.

Note that in order for '`autoload run-help`' to work, the `run-help` file must be in one of the directories named in your `fpath` array (see Section 14.6 [Parameters Used By The Shell], page 53). This should already be the case if you have a standard zsh installation; if it is not, copy `Functions/Misc/run-help` to an appropriate directory.

23.2.2 Recompiling Functions

If you frequently edit your zsh functions, or periodically update your zsh installation to track the latest developments, you may find that function digests compiled with the `zcompile` builtin are frequently out of date with respect to the function source files. This is not usually a problem, because zsh always looks for the newest file when loading a function, but it may cause slower shell startup and function loading. Also, if a digest file is explicitly used as an element of `fpath`, zsh won't check whether any of its source files has changed.

The `zrecompile` autoloadable function, found in `Functions/Misc`, can be used to keep function digests up to date.

`zrecompile` [`-qt`] [*name ...*]
`zrecompile` [`-qt`] `-p` *args* [`--` *args ...*]
: This tries to find `*.zwc` files and automatically re-compile them if at least one of the original files is newer than the compiled file. This works only if the names stored in the compiled files are full paths or are relative to the directory that contains the `.zwc` file.

  In the first form, each *name* is the name of a compiled file or a directory containing `*.zwc` files that should be checked. If no arguments are given, the directories and `*.zwc` files in `fpath` are used.

  When `-t` is given, no compilation is performed, but a return status of zero (true) is set if there are files that need to be re-compiled and non-zero (false) otherwise. The `-q` option quiets the chatty output that describes what `zrecompile` is doing.

  Without the `-t` option, the return status is zero if all files that needed re-compilation could be compiled and non-zero if compilation for at least one of the files failed.

  If the `-p` option is given, the *args* are interpreted as one or more sets of arguments for `zcompile`, separated by '`--`'. For example:

```
zrecompile -p \
           -R ~/.zshrc -- \
           -M ~/.zcompdump -- \
           ~/zsh/comp.zwc ~/zsh/Completion/*/_*
```

  This compiles `~/.zshrc` into `~/.zshrc.zwc` if that doesn't exist or if it is older than `~/.zshrc`. The compiled file will be marked for reading instead of mapping. The same is done for `~/.zcompdump` and `~/.zcompdump.zwc`, but this compiled file is marked for mapping. The last line re-creates the file `~/zsh/comp.zwc` if any of the files matching the given pattern is newer than it.

  Without the `-p` option, `zrecompile` does not create function digests that do not already exist, nor does it add new functions to the digest.

The following shell loop is an example of a method for creating function digests for all functions in your `fpath`, assuming that you have write permission to the directories:

```
for ((i=1; i <= $#fpath; ++i)); do
  dir=$fpath[i]
```

```
    zwc=${dir:t}.zwc
    if [[ $dir == (.|..) || $dir == (.|..)/* ]]; then
      continue
    fi
    files=($dir/*(N-.))
    if [[ -w $dir:h && -n $files ]]; then
      files=(${${(M)files%/*/*}#/})
      if ( cd $dir:h &&
           zrecompile -p -U -z $zwc $files ); then
        fpath[i]=$fpath[i].zwc
      fi
    fi
  done
```

The `-U` and `-z` options are appropriate for functions in the default zsh installation `fpath`; you may need to use different options for your personal function directories.

Once the digests have been created and your `fpath` modified to refer to them, you can keep them up to date by running `zrecompile` with no arguments.

23.2.3 Keyboard Definition

The large number of possible combinations of keyboards, workstations, terminals, emulators, and window systems makes it impossible for zsh to have built-in key bindings for every situation. The `zkbd` utility, found in Functions/Misc, can help you quickly create key bindings for your configuration.

Run `zkbd` either as an autoloaded function, or as a shell script:

```
zsh -f ~/zsh-4.0.3/Functions/Misc/zkbd
```

When you run `zkbd`, it first asks you to enter your terminal type; if the default it offers is correct, just press return. It then asks you to press a number of different keys to determine characteristics of your keyboard and terminal; `zkbd` warns you if it finds anything out of the ordinary, such as a Delete key that sends neither `^H` nor `^?`.

The keystrokes read by `zkbd` are recorded as a definition for an associative array named `key`, written to a file in the subdirectory `.zkbd` within either your `HOME` or `ZDOTDIR` directory. The name of the file is composed from the `TERM`, `VENDOR` and `OSTYPE` parameters, joined by hyphens.

You may read this file into your `.zshrc` or another startup file with the "source" or "." commands, then reference the `key` parameter in bindkey commands, like this:

```
source ${ZDOTDIR:-$HOME}/.zkbd/$TERM-$VENDOR-$OSTYPE
[[ -n ${key[Left]} ]] && bindkey "${key[Left]}" backward-char
[[ -n ${key[Right]} ]] && bindkey "${key[Right]}" forward-char
# etc.
```

Note that in order for '`autoload zkbd`' to work, the `zkdb` file must be in one of the directories named in your `fpath` array (see Section 14.6 [Parameters Used By The Shell], page 53). This should already be the case if you have a standard zsh installation; if it is not, copy `Functions/Misc/zkbd` to an appropriate directory.

23.2.4 Dumping Shell State

Occasionally you may encounter what appears to be a bug in the shell, particularly if you are using a beta version of zsh or a development release. Usually it is sufficient to send a description

of the problem to one of the zsh mailing lists (see Section 2.3 [Mailing Lists], page 2), but sometimes one of the zsh developers will need to recreate your environment in order to track the problem down.

The script named `reporter`, found in the `Util` directory of the distribution, is provided for this purpose. (It is also possible to `autoload reporter`, but `reporter` is not installed in `fpath` by default.) This script outputs a detailed dump of the shell state, in the form of another script that can be read with '`zsh -f`' to recreate that state.

To use `reporter`, read the script into your shell with the '`.`' command and redirect the output into a file:

```
. ~/zsh-4.0.3/Util/reporter > zsh.report
```

You should check the `zsh.report` file for any sensitive information such as passwords and delete them by hand before sending the script to the developers. Also, as the output can be voluminous, it's best to wait for the developers to ask for this information before sending it.

You can also use `reporter` to dump only a subset of the shell state. This is sometimes useful for creating startup files for the first time. Most of the output from reporter is far more detailed than usually is necessary for a startup file, but the `aliases`, `options`, and `zstyles` states may be useful because they include only changes from the defaults. The `bindings` state may be useful if you have created any of your own keymaps, because `reporter` arranges to dump the keymap creation commands as well as the bindings for every keymap.

As is usual with automated tools, if you create a startup file with `reporter`, you should edit the results to remove unnecessary commands. Note that if you're using the new completion system, you should *not* dump the `functions` state to your startup files with `reporter`; use the `compdump` function instead (see Chapter 19 [Completion System], page 127).

`reporter` [*state* ...]
: Print to standard output the indicated subset of the current shell state. The *state* arguments may be one or more of:

  `all` Output everything listed below.

  `aliases` Output alias definitions.

  `bindings` Output ZLE key maps and bindings.

  `completion`
  : Output old-style `compctl` commands. New completion is covered by `functions` and `zstyles`.

  `functions`
  : Output autoloads and function definitions.

  `limits` Output `limit` commands.

  `options` Output `setopt` commands.

  `styles` Same as `zstyles`.

  `variables`
  : Output shell parameter assignments, plus `export` commands for any environment variables.

  `zstyles` Output `zstyle` commands.

  If the *state* is omitted, `all` is assumed.

  With the exception of '`all`', every *state* can be abbreviated by any prefix, even a single letter; thus `a` is the same as `aliases`, `z` is the same as `zstyles`, etc.

23.3 Prompt Themes

23.3.1 Installation

You should make sure all the functions from the `Functions/Prompts` directory of the source distribution are available; they all begin with the string '`prompt_`' except for the special function'`promptinit`'. You also need the '`colors`' function from `Functions/Misc`. All of these functions may already have been installed on your system; if not, you will need to find them and copy them. The directory should appear as one of the elements of the `fpath` array (this should already be the case if they were installed), and at least the function `promptinit` should be autoloaded; it will autoload the rest. Finally, to initialize the use of the system you need to call the `promptinit` function. The following code in your `.zshrc` will arrange for this; assume the functions are stored in the directory `~/myfns`:

```
fpath=(~/myfns $fpath)
autoload -U promptinit
promptinit
```

23.3.2 Theme Selection

Use the `prompt` command to select your preferred theme. This command may be added to your `.zshrc` following the call to `promptinit` in order to start zsh with a theme already selected.

`prompt` [`-c` | `-l`]
`prompt` [`-p` | `-h`] [*theme ...*]
`prompt` [`-s`] *theme* [*arg ...*]

Set or examine the prompt theme. With no options and a *theme* argument, the theme with that name is set as the current theme. The available themes are determined at run time; use the `-l` option to see a list. The special *theme* '`random`' selects at random one of the available themes and sets your prompt to that.

In some cases the *theme* may be modified by one or more arguments, which should be given after the theme name. See the help for each theme for descriptions of these arguments.

Options are:

`-c` Show the currently selected theme and its parameters, if any.

`-l` List all available prompt themes.

`-p` Preview the theme named by *theme*, or all themes if no *theme* is given.

`-h` Show help for the theme named by *theme*, or for the `prompt` function if no *theme* is given.

`-s` Set *theme* as the current theme and save state.

`prompt_`*theme*`_setup`

Each available *theme* has a setup function which is called by the `prompt` function to install that theme. This function may define other functions as necessary to maintain the prompt, including functions used to preview the prompt or provide help for its use. You should not normally call a theme's setup function directly.

23.4 ZLE Functions

23.4.1 Widgets

These functions all implement user-defined ZLE widgets (see Chapter 17 [Zsh Line Editor], page 94) which can be bound to keystrokes in interactive shells. To use them, your `.zshrc` should contain lines of the form

```
autoload function
zle -N function
```

followed by an appropriate `bindkey` command to associate the function with a key sequence. Suggested bindings are described below.

`cycle-completion-positions`
: After inserting an unambiguous string into the command line, the new function based completion system may know about multiple places in this string where characters are missing or differ from at least one of the possible matches. It will then place the cursor on the position it considers to be the most interesting one, i.e. the one where one can disambiguate between as many matches as possible with as little typing as possible.

  This widget allows the cursor to be easily moved to the other interesting spots. It can be invoked repeatedly to cycle between all positions reported by the completion system.

`edit-command-line`
: Edit the command line using your visual editor, as in `ksh`.

```
bindkey -M vicmd v edit-command-line
```

`history-search-end`
: This function implements the widgets `history-beginning-search-backward-end` and `history-beginning-search-forward-end`. These commands work by first calling the corresponding builtin widget (see Section 17.6.2 [History Control], page 104) and then moving the cursor to the end of the line. The original cursor position is remembered and restored before calling the builtin widget a second time, so that the same search is repeated to look farther through the history.

  Although you `autoload` only one function, the commands to use it are slightly different because it implements two widgets.

```
zle -N history-beginning-search-backward-end \
       history-search-end
zle -N history-beginning-search-forward-end \
       history-search-end
bindkey '\e^P' history-beginning-search-backward-end
bindkey '\e^N' history-beginning-search-forward-end
```

`incarg`
: Typing the keystrokes for this widget with the cursor placed on or to the left of an integer causes that integer to be incremented by one. With a numeric prefix argument, the number is incremented by the amount of the argument (decremented if the prefix argument is negative). The shell parameter `incarg` may be set to change the default increment something other than one.

```
bindkey '^X+' incarg
```

`incremental-complete-word`
: This allows incremental completion of a word. After starting this command, a list of completion choices can be shown after every character you type, which you can

delete with ^H or DEL. Pressing return accepts the completion so far and returns you to normal editing (that is, the command line is *not* immediately executed). You can hit TAB to do normal completion, ^G to abort back to the state when you started, and ^D to list the matches.

This works only with the new function based completion system.

```
bindkey '^Xi' incremental-complete-word
```

`insert-files`

This function allows you type a file pattern, and see the results of the expansion at each step. When you hit return, all expansions are inserted into the command line.

```
bindkey '^Xf' insert-files
```

`predict-on`

This set of functions implements predictive typing using history search. After `predict-on`, typing characters causes the editor to look backward in the history for the first line beginning with what you have typed so far. After `predict-off`, editing returns to normal for the line found. In fact, you often don't even need to use `predict-off`, because if the line doesn't match something in the history, adding a key performs standard completion, and then inserts itself if no completions were found. However, editing in the middle of a line is liable to confuse prediction; see the `toggle` style below.

With the function based completion system (which is needed for this), you should be able to type TAB at almost any point to advance the cursor to the next "interesting" character position (usually the end of the current word, but sometimes somewhere in the middle of the word). And of course as soon as the entire line is what you want, you can accept with return, without needing to move the cursor to the end first.

The first time `predict-on` is used, it creates several additional widget functions:

`delete-backward-and-predict`
: Replaces the `backward-delete-char` widget. You do not need to bind this yourself.

`insert-and-predict`
: Implements predictive typing by replacing the `self-insert` widget. You do not need to bind this yourself.

`predict-off`
: Turns off predictive typing.

Although you `autoload` only the `predict-on` function, it is necessary to create a keybinding for `predict-off` as well.

```
zle -N predict-on
zle -N predict-off
bindkey '^X^Z' predict-on
bindkey '^Z' predict-off
```

`smart-insert-last-word`

This function may replace the `insert-last-word` widget, like so:

```
zle -N insert-last-word smart-insert-last-word
```

With a numeric prefix, it behaves like `insert-last-word`, except that words in comments are ignored when `INTERACTIVE_COMMENTS` is set.

Otherwise, the rightmost "interesting" word from the previous command is found and inserted. The default definition of "interesting" is that the word contains at least one alphabetic character, slash, or backslash. This definition may be overridden

by use of the `match` style. The context used to look up the style is the widget name, so usually the context is `:insert-last-word`. However, you can bind this function to different widgets to use different patterns:

```
zle -N insert-last-assignment smart-insert-last-word
zstyle :insert-last-assignment match '[[:alpha:]][][[:alnum:]]#=*'
bindkey '\e=' insert-last-assignment
```

23.4.2 Styles

The behavior of several of the above widgets can be controlled by the use of the `zstyle` mechanism. In particular, widgets that interact with the completion system pass along their context to any completions that they invoke.

`break-keys`
: This style is used by the `incremental-complete-word` widget. Its value should be a pattern, and all keys matching this pattern will cause the widget to stop incremental completion without the key having any further effect. Like all styles used directly by `incremental-complete-word`, this style is looked up using the context '`:incremental`'.

`completer`
: The `incremental-complete-word` and `insert-and-predict` widgets set up their top-level context name before calling completion. This allows one to define different sets of completer functions for normal completion and for these widgets. For example, to use completion, approximation and correction for normal completion, completion and correction for incremental completion and only completion for prediction one could use:

```
zstyle ':completion:*' completer \
        _complete _correct _approximate
zstyle ':completion:incremental:*' completer \
        _complete _correct
zstyle ':completion:predict:*' completer \
        _complete
```

It is a good idea to restrict the completers used in prediction, because they may be automatically invoked as you type. The `_list` and `_menu` completers should never be used with prediction. The `_approximate`, `_correct`, `_expand`, and `_match` completers may be used, but be aware that they may change characters anywhere in the word behind the cursor, so you need to watch carefully that the result is what you intended.

`cursor`
: The `insert-and-predict` widget uses this style, in the context '`:predict`', to decide where to place the cursor after completion has been tried. Values are:

  `complete`
  : The cursor is left where it was when completion finished, but only if it is after a character equal to the one just inserted by the user. If it is after another character, this value is the same as '`key`'.

  `key`
  : The cursor is left after the *n*th occurrence of the character just inserted, where *n* is the number of times that character appeared in the word before completion was attempted. In short, this has the effect of leaving the cursor after the character just typed even if the completion code found out that no other characters need to be inserted at that position.

Any other value for this style unconditionally leaves the cursor at the position where the completion code left it.

`list` When using the `incremental-complete-word` widget, this style says if the matches should be listed on every key press (if they fit on the screen). Use the context prefix '`:completion:incremental`'.

The `insert-and-predict` widget uses this style to decide if the completion should be shown even if there is only one possible completion. This is done if the value of this style is the string `always`. In this case the context is '`:predict`' (*not* '`:completion:predict`').

`match` This style is used by `smart-insert-last-word` to provide a pattern (using full `EXTENDED_GLOB` syntax) that matches an interesting word. The context is the name of the widget to which `smart-insert-last-word` is bound (see above). The default behavior of `smart-insert-last-word` is equivalent to:

```
zstyle :insert-last-word match '*[[:alpha:]/\\]*'
```

However, you might want to include words that contain spaces:

```
zstyle :insert-last-word match '*[[:alpha:][:space:]/\\]*'
```

Or include numbers as long as the word is at least two characters long:

```
zstyle :insert-last-word match '*([[:digit:]]?|[[:alpha:]/\\])*'
```

The above example causes redirections like "2>" to be included.

`prompt` The `incremental-complete-word` widget shows the value of this style in the status line during incremental completion. The string value may contain any of the following substrings in the manner of the `PS1` and other prompt parameters:

- `%c` Replaced by the name of the completer function that generated the matches (without the leading underscore).
- `%l` When the `list` style is set, replaced by '`...`' if the list of matches is too long to fit on the screen and with an empty string otherwise. If the `list` style is 'false' or not set, '`%l`' is always removed.
- `%n` Replaced by the number of matches generated.
- `%s` Replaced by '`-no match-`', '`-no prefix-`', or an empty string if there is no completion matching the word on the line, if the matches have no common prefix different from the word on the line, or if there is such a common prefix, respectively.
- `%u` Replaced by the unambiguous part of all matches, if there is any, and if it is different from the word on the line.

Like '`break-keys`', this uses the '`:incremental`' context.

`stop-keys`

This style is used by the `incremental-complete-word` widget. Its value is treated similarly to the one for the `break-keys` style (and uses the same context: '`:incremental`'). However, in this case all keys matching the pattern given as its value will stop incremental completion and will then execute their usual function.

`toggle` This boolean style is used by `predict-on` and its related widgets in the context '`:predict`'. If set to one of the standard 'true' values, predictive typing is automatically toggled off in situations where it is unlikely to be useful, such as when editing a multi-line buffer or after moving into the middle of a line and then deleting a character. The default is to leave prediction turned on until an explicit call to `predict-off`.

`verbose` This boolean style is used by `predict-on` and its related widgets in the context '`:predict`'. If set to one of the standard 'true' values, these widgets display a message below the prompt when the predictive state is toggled. This is most useful in combination with the `toggle` style. The default does not display these messages.

23.5 Other Functions

There are a large number of helpful functions in the `Functions/Misc` directory of the zsh distribution. Most are very simple and do not require documentation here, but a few are worthy of special mention.

23.5.1 Descriptions

`colors` This function initializes several associative arrays to map color names to (and from) the ANSI standard eight-color terminal codes. These are used by the prompt theme system (Section 23.3 [Prompt Themes], page 227). You seldom should need to run `colors` more than once.

The eight base colors are: black, red, green, yellow, blue, magenta, cyan, and white. Each of these has codes for foreground and background. In addition there are eight intensity attributes: bold, faint, standout, underline, blink, reverse, and conceal. Finally, there are six codes used to negate attributes: none (reset all attributes to the defaults), normal (neither bold nor faint), no-standout, no-underline, no-blink, and no-reverse.

Some terminals do not support all combinations of colors and intensities.

The associative arrays are:

color
colour
: Map all the color names to their integer codes, and integer codes to the color names. The eight base names map to the foreground color codes, as do names prefixed with '`fg-`', such as '`fg-red`'. Names prefixed with '`bg-`', such as '`bg-blue`', refer to the background codes. The reverse mapping from code to color yields base name for foreground codes and the `bg-` form for backgrounds.

  Although it is a misnomer to call them 'colors', these arrays also map the other fourteen attributes from names to codes and codes to names.

fg
fg_bold
fg_no_bold
: Map the eight basic color names to ANSI terminal escape sequences that set the corresponding foreground text properties. The `fg` sequences change the color without changing the eight intensity attributes.

bg
bg_bold
bg_no_bold
: Map the eight basic color names to ANSI terminal escape sequences that set the corresponding background properties. The `bg` sequences change the color without changing the eight intensity attributes.

In addition, the scalar parameters `reset_color` and `bold_color` are set to the ANSI terminal escapes that turn off all attributes and turn on bold intensity, respectively.

`fned` *name*
: Same as `zed -f`. This function does not appear in the zsh distribution, but can be created by linking `zed` to the name `fned` in some directory in your `fpath`.

`is-at-least` *needed* [*present*]
: Perform a greater-than-or-equal-to comparison of two strings having the format of a zsh version number; that is, a string of numbers and text with segments separated by dots or dashes. If the *present* string is not provided, `$ZSH_VERSION` is used.

Segments are paired left-to-right in the two strings with leading non-number parts ignored. If one string has fewer segments than the other, the missing segments are considered zero.

This is useful in startup files to set options and other state that are not available in all versions of zsh.

```
is-at-least 3.1.6-15 && setopt NO_GLOBAL_RCS
is-at-least 3.1.0 && setopt HIST_REDUCE_BLANKS
is-at-least 2.6-17 || print "You can't use is-at-least here."
```

`nslookup` [*arg* ...]
This wrapper function for the `nslookup` command requires the `zsh/zpty` module (see Section 21.22 [The zsh/zpty Module], page 209). It behaves exactly like the standard `nslookup` except that it provides customizable prompts (including a right-side prompt) and completion of nslookup commands, host names, etc. (if you use the function-based completion system). Completion styles may be set with the context prefix '`:completion:nslookup`'.

See also the `pager`, `prompt` and `rprompt` styles below.

`run-help` See 'Accessing On-Line Help' (Section 23.2 [Utilities], page 223).

`zed` [`-f`] *name*
This function uses the ZLE editor to edit a file or function. It rebinds the return key to insert a line break, and adds bindings for '`^X^W`' in the `emacs` keymap and '`ZZ`' in the `vicmd` keymap to accept (and therefore write, in the case of a file) the edited file or function. Keybindings are otherwise the standard ones; completion is available, and styles may be set with the context prefix '`:completion:zed`'.

Only one *name* argument is recognized (additional arguments are ignored). If the `-f` option is given, the name is taken to be that of a function; if the function is marked for autoloading, `zed` searches for it in the `fpath` and loads it. Note that functions edited this way are installed into the current shell, but *not* written back to the autoload file.

Without `-f`, *name* is the path name of the file to edit, which need not exist; it is created on write, if necessary.

`zcp` [`-finqQvw`] *srcpat dest*
`zln` [`-finqQsvw`] *srcpat dest*
Same as `zmv -C` and `zmv -L`, respectively. These functions do not appear in the zsh distribution, but can be created by linking `zmv` to the names `zcp` and `zln` in some directory in your `fpath`.

`zkbd` See 'Keyboard Definition' (Section 23.2 [Utilities], page 223).

`zmv` [`-finqQsvw`] [-C | -L | -M | -p *program*] [-o *optstring*] *srcpat dest*
Move (usually, rename) files matching the pattern *srcpat* to corresponding files having names of the form given by *dest*, where *srcpat* contains parentheses surrounding patterns which will be replaced in turn by $1, $2, ... in *dest*. For example,

```
zmv '(*).lis' '$1.txt'
```

renames '`foo.lis`' to '`foo.txt`', '`my.old.stuff.lis`' to '`my.old.stuff.txt`', and so on.

The pattern is always treated as an `EXTENDED_GLOB` pattern. Any file whose name is not changed by the substitution is simply ignored. Any error (a substitution resulted in an empty string, two substitutions gave the same result, the destination was an existing regular file and `-f` was not given) causes the entire function to abort without doing anything.

Options:

`-f` Force overwriting of destination files. Not currently passed down to the `mv/cp/ln` command due to vagaries of implementations (but you can use `-o-f` to do that).

`-i` Interactive: show each line to be executed and ask the user whether to execute it. 'Y' or 'y' will execute it, anything else will skip it. Note that you just need to type one character.

`-n` No execution: print what would happen, but don't do it.

`-q` Turn bare glob qualifiers off: now assumed by default, so this has no effect.

`-Q` Force bare glob qualifiers on. Don't turn this on unless you are actually using glob qualifiers in a pattern.

`-s` Symbolic, passed down to `ln`; only works with `-L`.

`-v` Verbose: print each command as it's being executed.

`-w` Pick out wildcard parts of the pattern, as described above, and implicitly add parentheses for referring to them.

`-C`
`-L`
`-M` Force `cp`, `ln` or `mv`, respectively, regardless of the name of the function.

`-p` *program*
Call *program* instead of `cp`, `ln` or `mv`. Whatever it does, it should at least understand the form

program `--` *oldname newname*

where *oldname* and *newname* are filenames generated by `zmv`.

`-o` *optstring*
The *optstring* is split into words and passed down verbatim to the `cp`, `ln` or `mv` command called to perform the work. It should probably begin with a '`-`'.

For more complete examples and other implementation details, see the `zmv` source file, usually located in one of the directories named in your `fpath`, or in `Functions/Misc/zmv` in the zsh distribution.

`zrecompile`
See 'Recompiling Functions' (Section 23.2 [Utilities], page 223).

`zstyle+` *context style value* [`+` *subcontext style value* ...]
This makes defining styles a bit simpler by using a single '`+`' as a special token that allows you to append a context name to the previously used context name. Like this:

```
zstyle+ ':foo:bar' style1 value1 \
      + ':baz'     style2 value2 \
      + ':frob'    style3 value3
```

This defines 'style1' with 'value1' for the context `:foo:bar` as usual, but it also defines 'style2' with 'value2' for the context `:foo:bar:baz` and 'style3' with 'value3' for `:foo:bar:frob`. Any *subcontext* may be the empty string to re-use the first context unchanged.

23.5.2 Styles

`insert-tab`
: The `zed` function *sets* this style in context '`:completion:zed:*`' to turn off completion when `TAB` is typed at the beginning of a line. You may override this by setting your own value for this context and style.

`pager`
: The `nslookup` function looks up this style in the context '`:nslookup`' to determine the program used to display output that does not fit on a single screen.

`prompt`
`rprompt`
: The `nslookup` function looks up this style in the context '`:nslookup`' to set the prompt and the right-side prompt, respectively. The usual expansions for the `PS1` and `RPS1` parameters may be used (see Chapter 12 [Prompt Expansion], page 22).

Concept Index

$

.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Z

Variables Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Options Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X

Z

Functions Index

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W

Z

Editor Functions Index

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Style and Tag Index

-

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Table of Contents