

NAME

zshmodules – zsh loadable modules

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. Modules are linked at runtime with the **zmodload** command, see *zshbuiltins(1)*.

The modules that are bundled with the zsh distribution are:

zsh/attr

Builtins for manipulating extended attributes (xattr).

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/complst

Completion listing extensions.

zsh/computil

A module with utility builtins needed for the shell function based completion system.

zsh/curses

curses windowing commands

zsh/datetime

Some date/time commands and parameters.

zsh/db/gdbm

Builtins for managing associative array parameters tied to GDBM databases.

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/langinfo

Interface to locale information.

zsh/mapfile

Access to external files via a special associative array.

zsh/mathfunc

Standard scientific functions for use in mathematical evaluations.

zsh/nearcolor

Map colours to the nearest colour in the available palette.

zsh/newuser

Arrange for files for new users to be installed.

zsh/parameter

Access to internal hash tables via special associative arrays.

- zsh/pcre**
Interface to the PCRE library.
- zsh/param/private**
Builtins for managing private–scoped parameters in function context.
- zsh/regex**
Interface to the POSIX regex library.
- zsh/sched**
A builtin that provides a timed execution facility within the shell.
- zsh/net/socket**
Manipulation of Unix domain sockets
- zsh/stat**
A builtin command interface to the **stat** system call.
- zsh/system**
A builtin interface to various low–level system features.
- zsh/net/tcp**
Manipulation of TCP sockets
- 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/zselect**
Block and return when file descriptors are ready.
- zsh/zutil**
Some utility builtins, e.g. the one for supporting configuration via styles.

THE ZSH/ATTR MODULE

The **zsh/attr** module is used for manipulating extended attributes. The **-h** option causes all commands to operate on symbolic links instead of their targets. The builtins in this module are:

- zgetattr** [**-h**] *filename attribute* [*parameter*]
Get the extended attribute *attribute* from the specified *filename*. If the optional argument *parameter* is given, the attribute is set on that parameter instead of being printed to stdout.
- zsetattr** [**-h**] *filename attribute value*
Set the extended attribute *attribute* on the specified *filename* to *value*.
- zdelattr** [**-h**] *filename attribute*
Remove the extended attribute *attribute* from the specified *filename*.
- zlistattr** [**-h**] *filename* [*parameter*]
List the extended attributes currently set on the specified *filename*. If the optional argument *parameter* is given, the list of attributes is set on that parameter instead of being printed to stdout.

zgetattr and **zlistattr** allocate memory dynamically. If the attribute or list of attributes grows between the allocation and the call to get them, they return 2. On all other errors, 1 is returned. This allows the calling function to check for this case and retry.

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*.

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 status of the builtin is zero in both shells if successful, and non-zero on error.

The target of **clone** should be an unused terminal, such as an unused virtual console or a virtual terminal created by

```
xterm -e sh -c 'trap : INT QUIT TSTP; tty;
while ;; do sleep 10000000; done'
```

Some words of explanation are warranted about this long `xterm` command line: when doing `clone` on a pseudo-terminal, some other session ("session" meant as a unix session group, or SID) is already owning the terminal. Hence the cloned `zsh` cannot acquire the pseudo-terminal as a controlling `tty`. That means two things:

- the job control signals will go to the `sh`-started-by-`xterm` process group (that's why we disable `INT QUIT` and `TSTP` with `trap`; otherwise the `while` loop could get suspended or killed)
- the cloned shell will have job control disabled, and the job control keys (`control-C`, `control-\` and `control-Z`) will not work.

This does not apply when cloning to an *unused* `vc`.

Cloning to a used (and unprepared) terminal will result in two processes reading simultaneously from the same terminal, with input bytes going randomly to either process.

clone is mostly useful as a shell built-in replacement for `openvt`.

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 `zshcompctl(1)`. The other builtin command, **compcall** can be used in user-defined completion widgets, see `zshcompwid(1)`.

THE ZSH/COMPLETE MODULE

The **zsh/complete** module makes available several builtin commands which can be used in user-defined completion widgets, see `zshcompwid(1)`.

THE ZSH/COMPLIST MODULE

The **zsh/compllist** 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.

Colored completion listings

Whenever one of the parameters **ZLS_COLORS** or **ZLS_COLOURS** is set and the **zsh/compllist** module is loaded or linked into the shell, completion lists will be colored. Note, however, that **compllist** will not automatically be loaded if it is not linked in: on systems with dynamic loading, **'zmodload zsh/compllist'** 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. If this has the special value **target**, symbolic links are dereferenced and the target file used to determine the display format.
- pi 31** for named pipes (FIFOs)
- so 33** for sockets
- bd 44;37**
for block devices
- cd 44;37**
for character devices
- or none**
for a symlink to nonexistent file (default is the value defined for **ln**)
- mi none**
for a non-existent file (default is the value defined for **fi**); this code is currently not used
- su 37;41**
for files with setuid bit set
- sg 30;43**
for files with setgid bit set
- tw 30;42**
for world writable directories with sticky bit set
- ow 34;43**
for world writable directories without sticky bit set
- sa none** for files with an associated suffix alias; this is only tested after specific suffixes, as described below
- st 37;44**
for directories with sticky bit set but not world writable
- ex 35** for executable files
- 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 **EXTENDED_GLOB** option will be turned on for evaluation of the 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 the form with the leading equal sign take precedence over the values defined for file types, which in turn take precedence over the form with the leading asterisk (file extensions).

The leading-equals 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. Patterns may be matched against completions, descriptions (possibly with spaces appended for padding), or lines consisting of a completion followed by a description. For consistent coloring it may be necessary to use more than one pattern or a pattern with backreferences.

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 **xterms**. 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 the section 'Other Functions' in *zshcontrib(1)*). For example, after loading **colors**, one could use **\$color[red]** to get the code for foreground color red and **\$color[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 the section 'Completion System Configuration' in *zshcompsys(1)*).

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**, **%F**, **%f**, **%K**, **%k** 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

accept-search

stop listing but take no other action

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.

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 *zshoptions(1)*).

Menu selection can be invoked directly by the widget **menu-select** defined by this module. This is a standard ZLE widget that can be bound to a key in the usual way as described in *zshzle(1)*.

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 stand-out 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. Note that the following always perform the same task within the menu selection map and cannot be replaced by user defined widgets, nor can the set of functions be extended:

accept-line, accept-search

accept the current match and leave menu selection (but do not cause the command line to be accepted)

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

vi-insert

this toggles between normal and interactive mode; in interactive mode the keys bound to **self-insert** and **self-insert-unmeta** insert into the command line as in normal editing mode but without leaving menu selection; after each character completion is tried again and the list changes to contain only the new matches; the completion widgets make the longest unambiguous string be inserted in the command line and **undo** and **backward-delete-char** go back to the previous set of matches

history-incremental-search-forward**history-incremental-search-backward**

this starts incremental searches in the list of completions displayed; in this mode, **accept-line** only leaves incremental search, going back to the normal menu selection mode

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 *zshmodules(1)*). 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.

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 *zshcompsys(1)*). 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 four parameters as its arguments. This will step through the different sets of matches and store the value of **compstate[list]** in the first scalar, the options for **compadd** in the second array, the matches in the third array, and the strings to be displayed in the completion listing in the fourth array. The arrays may then be 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 status 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.

THE ZSH/CURSES MODULE

The **zsh/curses** module makes available one builtin command and various parameters.

Builtin

zcurses init

zcurses end

zcurses addwin *targetwin nlines ncols begin_y begin_x [parentwin]*

zcurses delwin *targetwin*

zcurses refresh [*targetwin ...*]

zcurses touch *targetwin ...*

zcurses move *targetwin new_y new_x*

zcurses clear *targetwin [redraw | eol | bot]*

zcurses position *targetwin array*

zcurses char *targetwin character*

zcurses string *targetwin string*

```

zcurses border targetwin border
zcurses attr targetwin [ [+|-]attribute [fg_collbg_col] [...]]
zcurses bg targetwin [ [+|-]attribute [fg_collbg_col | @char] [...]]
zcurses scroll targetwin [ on | off ] [+|-]lines ]
zcurses input targetwin [ param [ kparam [ mparam ] ] ]
zcurses mouse [ delay num | [+|-]motion ]
zcurses timeout targetwin intval
zcurses querychar targetwin [ param ]
zcurses resize height width [ endwin | nosave | endwin_nosave ]

```

Manipulate curses windows. All uses of this command should be bracketed by ‘**zcurse**s **init**’ to initialise use of curses, and ‘**zcurse**s **end**’ to end it; omitting ‘**zcurse**s **end**’ can cause the terminal to be in an unwanted state.

The subcommand **addwin** creates a window with *nlines* lines and *ncols* columns. Its upper left corner will be placed at row *begin_y* and column *begin_x* of the screen. *targetwin* is a string and refers to the name of a window that is not currently assigned. Note in particular the curses convention that vertical values appear before horizontal values.

If **addwin** is given an existing window as the final argument, the new window is created as a subwindow of *parentwin*. This differs from an ordinary new window in that the memory of the window contents is shared with the parent’s memory. Subwindows must be deleted before their parent. Note that the coordinates of subwindows are relative to the screen, not the parent, as with other windows.

Use the subcommand **delwin** to delete a window created with **addwin**. Note that **end** does *not* implicitly delete windows, and that **delwin** does not erase the screen image of the window.

The window corresponding to the full visible screen is called **stdscr**; it always exists after ‘**zcurse**s **init**’ and cannot be deleted with **delwin**.

The subcommand **refresh** will refresh window *targetwin*; this is necessary to make any pending changes (such as characters you have prepared for output with **char**) visible on the screen. **refresh** without an argument causes the screen to be cleared and redrawn. If multiple windows are given, the screen is updated once at the end.

The subcommand **touch** marks the *targetwins* listed as changed. This is necessary before **refreshing** windows if a window that was in front of another window (which may be **stdscr**) is deleted.

The subcommand **move** moves the cursor position in *targetwin* to new coordinates *new_y* and *new_x*. Note that the subcommand **string** (but not the subcommand **char**) advances the cursor position over the characters added.

The subcommand **clear** erases the contents of *targetwin*. One (and no more than one) of three options may be specified. With the option **redraw**, in addition the next **refresh** of *targetwin* will cause the screen to be cleared and repainted. With the option **eol**, *targetwin* is only cleared to the end of the current cursor line. With the option **bot**, *targetwin* is cleared to the end of the window, i.e. everything to the right and below the cursor is cleared.

The subcommand **position** writes various positions associated with *targetwin* into the array named *array*. These are, in order:

- The y and x coordinates of the cursor relative to the top left of *targetwin*
- The y and x coordinates of the top left of *targetwin* on the screen
- The size of *targetwin* in y and x dimensions.

Outputting characters and strings are achieved by **char** and **string** respectively.

To draw a border around window *targetwin*, use **border**. Note that the border is not subsequently handled specially: in other words, the border is simply a set of characters output at the edge of the window. Hence it can be overwritten, can scroll off the window, etc.

The subcommand **attr** will set *targetwin*’s attributes or foreground/background color pair for any successive character output. Each *attribute* given on the line may be prepended by a + to set or a –

to unset that attribute; + is assumed if absent. The attributes supported are **blink**, **bold**, **dim**, **reverse**, **standout**, and **underline**.

Each *fg_collbg_col* attribute (to be read as '*fg_col* on *bg_col*') sets the foreground and background color for character output. The color **default** is sometimes available (in particular if the library is ncurses), specifying the foreground or background color with which the terminal started. The color pair **default/default** is always available. To use more than the 8 named colors (red, green, etc.) construct the *fg_collbg_col* pairs where *fg_col* and *bg_col* are decimal integers, e.g **128/200**. The maximum color value is 254 if the terminal supports 256 colors.

bg overrides the color and other attributes of all characters in the window. Its usual use is to set the background initially, but it will overwrite the attributes of any characters at the time when it is called. In addition to the arguments allowed with **attr**, an argument *@char* specifies a character to be shown in otherwise blank areas of the window. Owing to limitations of curses this cannot be a multibyte character (use of ASCII characters only is recommended). As the specified set of attributes override the existing background, turning attributes off in the arguments is not useful, though this does not cause an error.

The subcommand **scroll** can be used with **on** or **off** to enabled or disable scrolling of a window when the cursor would otherwise move below the window due to typing or output. It can also be used with a positive or negative integer to scroll the window up or down the given number of lines without changing the current cursor position (which therefore appears to move in the opposite direction relative to the window). In the second case, if scrolling is **off** it is temporarily turned **on** to allow the window to be scrolled.

The subcommand **input** reads a single character from the window without echoing it back. If *param* is supplied the character is assigned to the parameter *param*, else it is assigned to the parameter **REPLY**.

If both *param* and *kparam* are supplied, the key is read in 'keypad' mode. In this mode special keys such as function keys and arrow keys return the name of the key in the parameter *kparam*. The key names are the macros defined in the **curses.h** or **ncurses.h** with the prefix '**KEY_**' removed; see also the description of the parameter **zcurse_keycodes** below. Other keys cause a value to be set in *param* as before. On a successful return only one of *param* or *kparam* contains a non-empty string; the other is set to an empty string.

If *mparam* is also supplied, **input** attempts to handle mouse input. This is only available with the ncurses library; mouse handling can be detected by checking for the exit status of '**zcurse mouse**' with no arguments. If a mouse button is clicked (or double- or triple-clicked, or pressed or released with a configurable delay from being clicked) then **kparam** is set to the string **MOUSE**, and *mparam* is set to an array consisting of the following elements:

- An identifier to discriminate different input devices; this is only rarely useful.
- The x, y and z coordinates of the mouse click relative to the full screen, as three elements in that order (i.e. the y coordinate is, unusually, after the x coordinate). The z coordinate is only available for a few unusual input devices and is otherwise set to zero.
- Any events that occurred as separate items; usually there will be just one. An event consists of **PRESSED**, **RELEASED**, **CLICKED**, **DOUBLE_CLICKED** or **TRIPLE_CLICKED** followed immediately (in the same element) by the number of the button.
- If the shift key was pressed, the string **SHIFT**.
- If the control key was pressed, the string **CTRL**.
- If the alt key was pressed, the string **ALT**.

Not all mouse events may be passed through to the terminal window; most terminal emulators handle some mouse events themselves. Note that the ncurses manual implies that using input both with and without mouse handling may cause the mouse cursor to appear and disappear.

The subcommand **mouse** can be used to configure the use of the mouse. There is no window argument; mouse options are global. '**zcurse mouse**' with no arguments returns status 0 if mouse

handling is possible, else status 1. Otherwise, the possible arguments (which may be combined on the same command line) are as follows. **delay** *num* sets the maximum delay in milliseconds between press and release events to be considered as a click; the value 0 disables click resolution, and the default is one sixth of a second. **motion** preceded by an optional '+' (the default) or - turns on or off reporting of mouse motion in addition to clicks, presses and releases, which are always reported. However, it appears reports for mouse motion are not currently implemented.

The subcommand **timeout** specifies a timeout value for input from *targetwin*. If *intval* is negative, '**zcurse**s **input**' waits indefinitely for a character to be typed; this is the default. If *intval* is zero, '**zcurse**s **input**' returns immediately; if there is typeahead it is returned, else no input is done and status 1 is returned. If *intval* is positive, '**zcurse**s **input**' waits *intval* milliseconds for input and if there is none at the end of that period returns status 1.

The subcommand **querychar** queries the character at the current cursor position. The return values are stored in the array named *param* if supplied, else in the array **reply**. The first value is the character (which may be a multibyte character if the system supports them); the second is the color pair in the usual *fg_collbg_col* notation, or **0** if color is not supported. Any attributes other than color that apply to the character, as set with the subcommand **attr**, appear as additional elements.

The subcommand **resize** resizes **stdscr** and all windows to given dimensions (windows that stick out from the new dimensions are resized down). The underlying curses extension (**resize_term** call) can be unavailable. To verify, zeroes can be used for *height* and *width*. If the result of the subcommand is **0**, *resize_term* is available (**2** otherwise). Tests show that resizing can be normally accomplished by calling **zcurse**s **end** and **zcurse**s **refresh**. The **resize** subcommand is provided for versatility. Multiple system configurations have been checked and **zcurse**s **end** and **zcurse**s **refresh** are still needed for correct terminal state after **resize**. To invoke them with **resize**, use *endwin* argument. Using *nosave* argument will cause new terminal state to not be saved internally by **zcurse**s. This is also provided for versatility and should normally be not needed.

Parameters

ZCURSES_COLORS

Readonly integer. The maximum number of colors the terminal supports. This value is initialised by the curses library and is not available until the first time **zcurse**s **init** is run.

ZCURSES_COLOR_PAIRS

Readonly integer. The maximum number of color pairs *fg_collbg_col* that may be defined in '**zcurse**s **attr**' commands; note this limit applies to all color pairs that have been used whether or not they are currently active. This value is initialised by the curses library and is not available until the first time **zcurse**s **init** is run.

zcurses_attr

Readonly array. The attributes supported by **zsh/curses**; available as soon as the module is loaded.

zcurses_colors

Readonly array. The colors supported by **zsh/curses**; available as soon as the module is loaded.

zcurses_keycodes

Readonly array. The values that may be returned in the second parameter supplied to '**zcurse**s **input**' in the order in which they are defined internally by curses. Not all function keys are listed, only **F0**; curses reserves space for **F0** up to **F63**.

zcurses_windows

Readonly array. The current list of windows, i.e. all windows that have been created with '**zcurse**s **addwin**' and not removed with '**zcurse**s **delwin**'.

THE ZSH/DATETIME MODULE

The **zsh/datetime** module makes available one builtin command:

strftime [**-s** *scalar*] *format* [*epochtime* [*nanoseconds*]]

strftime -r [**-q**] [**-s** *scalar*] *format timestring*

Output the date in the *format* specified. With no *epochtime*, the current system date/time is used; optionally, *epochtime* may be used to specify the number of seconds since the epoch, and *nanoseconds* may additionally be used to specify the number of nanoseconds past the second (otherwise that number is assumed to be 0). See *strftime(3)* for details. The zsh extensions described in the section EXPANSION OF PROMPT SEQUENCES in *zshmisc(1)* are also available.

-q Run quietly; suppress printing of all error messages described below. Errors for invalid *epochtime* values are always printed.

-r With the option **-r** (reverse), use *format* to parse the input string *timestring* and output the number of seconds since the epoch at which the time occurred. The parsing is implemented by the system function **strptime**; see *strptime(3)*. This means that zsh format extensions are not available, but for reverse lookup they are not required.

In most implementations of **strftime** any timezone in the *timestring* is ignored and the local timezone declared by the **TZ** environment variable is used; other parameters are set to zero if not present.

If *timestring* does not match *format* the command returns status 1 and prints an error message. If *timestring* matches *format* but not all characters in *timestring* were used, the conversion succeeds but also prints an error message.

If either of the system functions **strptime** or **mktime** is not available, status 2 is returned and an error message is printed.

-s *scalar*

Assign the date string (or epoch time in seconds if **-r** is given) to *scalar* instead of printing it.

Note that depending on the system's declared integral time type, **strftime** may produce incorrect results for epoch times greater than 2147483647 which corresponds to 2038-01-19 03:14:07 +0000.

The **zsh/datetime** module makes available several parameters; all are readonly:

EPOCHREALTIME

A floating point value representing the number of seconds since the epoch. The notional accuracy is to nanoseconds if the **clock_gettime** call is available and to microseconds otherwise, but in practice the range of double precision floating point and shell scheduling latencies may be significant effects.

EPOCHSECONDS

An integer value representing the number of seconds since the epoch.

epochtime

An array value containing the number of seconds since the epoch in the first element and the remainder of the time since the epoch in nanoseconds in the second element. To ensure the two elements are consistent the array should be copied or otherwise referenced as a single substitution before the values are used. The following idiom may be used:

```
for secs nsecs in $epochtime; do
...
done
```

THE ZSH/DB/GDBM MODULE

The **zsh/db/gdbm** module is used to create "tied" associative arrays that interface to database files. If the GDBM interface is not available, the builtins defined by this module will report an error. This module is also intended as a prototype for creating additional database interfaces, so the **ztie** builtin may move to a more generic module in the future.

The builtins in this module are:

ztie **-d db/gdbm -f filename** [**-r**] *arrayname*

Open the GDBM database identified by *filename* and, if successful, create the associative array *arrayname* linked to the file. To create a local tied array, the parameter must first be declared, so commands similar to the following would be executed inside a function scope:

```
local -A sampledb
ztie -d db/gdbm -f sample.gdbm sampledb
```

The **-r** option opens the database file for reading only, creating a parameter with the readonly attribute. Without this option, using **'ztie'** on a file for which the user does not have write permission is an error. If writable, the database is opened synchronously so fields changed in *arrayname* are immediately written to *filename*.

Changes to the file modes *filename* after it has been opened do not alter the state of *arrayname*, but **'typeset -r arrayname'** works as expected.

zuntie [**-u**] *arrayname* ...

Close the GDBM database associated with each *arrayname* and then unset the parameter. The **-u** option forces an unset of parameters made readonly with **'ztie -r'**.

This happens automatically if the parameter is explicitly unset or its local scope (function) ends. Note that a readonly parameter may not be explicitly unset, so the only way to unset a global parameter created with **'ztie -r'** is to use **'zuntie -u'**.

zgdbmpath *parametername*

Put path to database file assigned to *parametername* into **REPLY** scalar.

zgdbm_tied

Array holding names of all tied parameters.

The fields of an associative array tied to GDBM are neither cached nor otherwise stored in memory, they are read from or written to the database on each reference. Thus, for example, the values in a readonly array may be changed by a second writer of the same database file.

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.

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.

THE ZSH/FILES MODULE

The **zsh/files** module makes available some common commands for file manipulation as builtins; these commands are probably not needed for many normal situations but can be useful in emergency recovery situations with constrained resources. The commands do not implement all features now required by relevant standards committees.

For all commands, a variant beginning **zf_** is also available and loaded automatically. Using the features capability of **zmodload** will let you load only those names you want. Note that it's possible to load only the builtins with **zsh**-specific names using the following command:

```
zmodload -m -F zsh/files b:zf_*
```

The commands loaded by default are:

chgrp [**-hRs**] *group filename ...*

Changes group of files specified. This is equivalent to **chown** with a *user-spec* argument of `‘:group’`.

chmod [**-Rs**] *mode filename ...*

Changes mode of files specified.

The specified *mode* must be in octal.

The **-R** option causes **chmod** to recursively descend into directories, changing the mode of all files in the directory after changing the mode of the directory itself.

The **-s** option is a zsh extension to **chmod** functionality. It enables paranoid behaviour, intended to avoid security problems involving a **chmod** being tricked into affecting files other than the ones intended. It will refuse to follow symbolic links, so that (for example) “**chmod 600 /tmp/foo/passwd**” can’t accidentally **chmod /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 **chmod** of a deep directory tree can’t end up recursively **chmoding /usr** as a result of directories being moved up the tree.

chown [**-hRs**] *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).

If the target is a symbolic link, the **-h** option causes **chown** to set the ownership of the link instead of its target.

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 [**-dfhins**] *filename dest*

ln [**-dfhins**] *filename ... dir*

Creates hard (or, with **-s**, symbolic) links. In the first form, the specified *destination* is created, as a link to the specified *filename*. In the second form, each of the *filenames* is taken in turn, and linked to a pathname in the specified *directory* 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.

The **-h** and **-n** options are identical and both exist for compatibility; either one indicates that if the target is a symlink then it should not be dereferenced. Typically this is used in combination with **-sf** so that if an existing link points to a directory then it will be removed, instead of followed. If this option is used with multiple filenames and the target is a symbolic link pointing to a directory then the result is an error.

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 *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 *destination*. In the second form, each of the *filenames* is taken in turn, and moved to a pathname in the specified *directory* 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 [**-dfiRrs**] *filename* ...

Removes files and directories specified.

Normally, **rm** will not remove directories (except with the **-R** or **-r** options). The **-d** option causes **rm** to try removing directories with **unlink** (see *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** and **-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** and **-r** options cause **rm** to recursively descend into directories, deleting all files in the directory before removing the directory with the **rmdir** system call (see *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 *sync(2)*), which flushes dirty buffers to disk. It might return before the I/O has actually been completed.

THE ZSH/LANGINFO MODULE

The **zsh/langinfo** module makes available one parameter:

langinfo

An associative array that maps langinfo elements to their values.

Your implementation may support a number of the following keys:

CODESET, D_T_FMT, D_FMT, T_FMT, RADIXCHAR, THOUSEP, YESEXPR, NOEXPR, CRNCYSTR, ABDAY_{1..7}, DAY_{1..7}, ABMON_{1..12}, MON_{1..12}, T_FMT_AMPM, AM_STR, PM_STR, ERA, ERA_D_FMT, ERA_D_T_FMT, ERA_T_FMT, ALT_DIGITS

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 effect 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.

A file may conveniently be read into an array as one line per element with the form **'array=("\${(f@)mapfile[filename]}")'**. The double quotes and the **'@'** are necessary to prevent empty lines from being removed. Note that if the file ends with a newline, the shell will split on the final newline, generating an additional empty field; this can be suppressed by using **'array=("\${(f@)\${mapfile[filename]}%\$'\n'}")'**.

Limitations

Although reading and writing of the file in question is efficiently handled, zsh's internal memory management may be arbitrarily baroque; however, **mapfile** is usually very much more efficient than anything involving a loop. 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.

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**, **log2**, **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 *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 functions **min**, **max**, and **sum** are defined not in this module but in the **zmathfunc** autoloadable function, described in the section ‘Mathematical Functions’ in *zshcontrib(1)*.

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.

The function **rand48** is available if your system’s mathematical library has the function **erand48(3)**. It returns a pseudo-random floating point number between 0 and 1. It takes a single string optional argument.

If the argument is not present, the random number seed is initialised by three calls to the **rand(3)** function — this produces the same random numbers as the next three values of **\$RANDOM**.

If the argument is present, it gives the name of a scalar parameter where the current random number seed will be stored. On the first call, the value must contain at least twelve hexadecimal digits (the remainder of the string is ignored), or the seed will be initialised in the same manner as for a call to **rand48** with no argument. Subsequent calls to **rand48(param)** will then maintain the seed in the parameter *param* as a string of twelve hexadecimal digits, with no base signifier. The random number sequences for different parameters are completely independent, and are also independent from that used by calls to **rand48** with no argument.

For example, consider

```
print $(( rand48(seed) ))
print $(( rand48() ))
print $(( rand48(seed) ))
```

Assuming **\$seed** does not exist, it will be initialised by the first call. In the second call, the default seed is initialised; note, however, that because of the properties of **rand()** there is a correlation between the seeds used for the two initialisations, so for more secure uses, you should generate your own 12-byte seed. The third call returns to the same sequence of random numbers used in the first call, unaffected by the intervening **rand48()**.

THE ZSH/NEARCOLOR MODULE

The **zsh/nearcolor** module replaces colours specified as hex triplets with the nearest colour in the 88 or 256 colour palettes that are widely used by terminal emulators. By default, 24-bit true colour escape codes are generated when colours are specified using hex triplets. These are not supported by all terminals. The purpose of this module is to make it easier to define colour preferences in a form that can work across a range of terminal emulators.

Aside from the default colour, the ANSI standard for terminal escape codes provides for eight colours. The bright attribute brings this to sixteen. These basic colours are commonly used in terminal applications due to being widely supported. Expanded 88 and 256 colour palettes are also common and, while the first sixteen colours vary somewhat between terminals and configurations, these add a generally consistent and

predictable set of colours.

In order to use the **zsh/nearcolor** module, it only needs to be loaded. Thereafter, whenever a colour is specified using a hex triplet, it will be compared against each of the available colours and the closest will be selected. The first sixteen colours are never matched in this process due to being unpredictable.

It isn't possible to reliably detect support for true colour in the terminal emulator. It is therefore recommended to be selective in loading the **zsh/nearcolor** module. For example, the following checks the **COLORTERM** environment variable:

```
[[ $COLORTERM = *(24bit{truecolor})* ]] || zmodload zsh/nearcolor
```

Note that some terminals accept the true color escape codes but map them internally to a more limited palette in a similar manner to the **zsh/nearcolor** module.

THE ZSH/NEWUSER MODULE

The **zsh/newuser** module is loaded at boot if it is available, the **RCS** option is set, and the **PRIVILEGED** option is not set (all three are true by default). This takes place immediately after commands in the global **zshenv** file (typically **/etc/zsh/zshenv**), if any, have been executed. If the module is not available it is silently ignored by the shell; the module may safely be removed from **\$MODULE_PATH** by the administrator if it is not required.

On loading, the module tests if any of the start-up files **.zshenv**, **.zprofile**, **.zshrc** or **.zlogin** exist in the directory given by the environment variable **ZDOTDIR**, or the user's home directory if that is not set. The test is not performed and the module halts processing if the shell was in an emulation mode (i.e. had been invoked as some other shell than zsh).

If none of the start-up files were found, the module then looks for the file **newuser** first in a sitewide directory, usually the parent directory of the **site-functions** directory, and if that is not found the module searches in a version-specific directory, usually the parent of the **functions** directory containing version-specific functions. (These directories can be configured when zsh is built using the **--enable-site-scriptdir=dir** and **--enable-scriptdir=dir** flags to **configure**, respectively; the defaults are *prefix/share/zsh* and *prefix/share/zsh/\$ZSH_VERSION* where the default *prefix* is **/usr/local**.)

If the file **newuser** is found, it is then sourced in the same manner as a start-up file. The file is expected to contain code to install start-up files for the user, however any valid shell code will be executed.

The **zsh/newuser** module is then unconditionally unloaded.

Note that it is possible to achieve exactly the same effect as the **zsh/newuser** module by adding code to **/etc/zsh/zshenv**. The module exists simply to allow the shell to make arrangements for new users without the need for intervention by package maintainers and system administrators.

The script supplied with the module invokes the shell function **zsh-newuser-install**. This may be invoked directly by the user even if the **zsh/newuser** module is disabled. Note, however, that if the module is not installed the function will not be installed either. The function is documented in the section User Configuration Functions in *zshcontrib(1)*.

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.

functions_source

This readonly associative array maps names of enabled functions to the name of the file containing the source of the function.

For an autoloading function that has already been loaded, or marked for autoload with an absolute path, or that has had its path resolved with '**functions -r**', this is the file found for autoloading, resolved to an absolute path.

For a function defined within the body of a script or sourced file, this is the name of that file. In this case, this is the exact path originally used to that file, which may be a relative path.

For any other function, including any defined at an interactive prompt or an autoload function whose path has not yet been resolved, this is the empty string. However, the hash element is reported as defined just so long as the function is present: the keys to this hash are the same as those to **\$functions**.

dis_functions_source

Like **functions_source** 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.

patchars

This array contains the enabled pattern characters.

dis_patchars

Like **patchars** but for disabled pattern characters.

aliases This maps the names of the regular aliases currently enabled to their expansions.

dis_aliases

Like **aliases** but for disabled regular aliases.

galiases

Like **aliases**, but for global aliases.

dis_galiases

Like **galiases** but for disabled global aliases.

saliases

Like **raliases**, but for suffix aliases.

dis_saliases

Like **saliases** but for disabled suffix 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 *zshexpn(1)*. 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 autoloading, or aliased. The value says which state the named module is in and is one of the strings **loaded**, **autoloading**, 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. Although it is presented as an associative array, the array of all values (**\${history[@]}**) is guaranteed to be returned in order from most recent to oldest history event, that is, by decreasing history event number.

historywords

A special array containing the words stored in the history. These also appear in most to least recent order.

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).

The keys of the associative arrays are usually valid job numbers, and these are the values output with, for example, **\${(k)jobdirs}**. Non-numeric job references may be used when looking up a value; for example, **\${jobdirs[%+]}** refers to the current job.

jobtexts

This associative array maps job numbers to the texts of the command lines that were used to start the jobs.

Handling of the keys of the associative array is as described for **jobdirs** above.

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 *pids* are, of course, the process IDs and the *state* describes the state of that process.

Handling of the keys of the associative array is as described for **jobdirs** above.

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.

usergroups

This associative array maps names of system groups of which the current user is a member to the corresponding group identifiers. The contents are the same as the groups output by the **id** command.

funcfiletrace

This array contains the absolute line numbers and corresponding file names for the point where the current function, sourced file, or (if **EINVALENENO** is set) **eval** command was called. The array is of the same length as **funcsourcectrace** and **funcctrace**, but differs from **funcsourcectrace** in that the line and file are the point of call, not the point of definition, and differs from **funcctrace** in that all values are absolute line numbers in files, rather than relative to the start of a function, if any.

funcsourcectrace

This array contains the file names and line numbers of the points where the functions, sourced files, and (if **EINVALENENO** is set) **eval** commands currently being executed were defined. The line number is the line where the **'function name'** or **'name ()'** started. In the case of an autoloaded function the line number is reported as zero. The format of each element is *filename:lineno*.

For functions autoloaded from a file in native zsh format, where only the body of the function occurs in the file, or for files that have been executed by the **source** or **'.'** builtins, the trace information is shown as *filename:0*, since the entire file is the definition. The source file name is resolved to an absolute path when the function is loaded or the path to it otherwise resolved.

Most users will be interested in the information in the **funcfiletrace** array instead.

funcstack

This array contains the names of the functions, sourced files, and (if **EINVALENENO** is set) **eval** commands. currently being executed. The first element is the name of the function using the parameter.

The standard shell array **zsh_eval_context** can be used to determine the type of shell construct being executed at each depth: note, however, that is in the opposite order, with the most recent item last, and it is more detailed, for example including an entry for **toplevel**, the main shell code being executed either interactively or from a script, which is not present in **\$funcstack**.

funcctrace

This array contains the names and line numbers of the callers corresponding to the functions currently being executed. The format of each element is *name:lineno*. Callers are also shown for sourced files; the caller is the point where the **source** or **'.'** command was executed.

THE ZSH/PCRE MODULE

The **zsh/pcre** module makes some commands available as builtins:

pcre_compile [**-aimxs**] *PCRE*

Compiles a perl-compatible regular expression.

Option **-a** will force the pattern to be anchored. Option **-i** will compile a case-insensitive pattern. Option **-m** will compile a multi-line pattern; that is, **^** and **\$** will match newlines within the pattern. Option **-x** will compile an extended pattern, wherein whitespace and **#** comments are ignored. Option **-s** makes the dot metacharacter match all characters, including those that indicate newline.

pcre_study

Studies the previously-compiled PCRE which may result in faster matching.

pcre_match [**-v** *var*] [**-a** *arr*] [**-n** *offset*] [**-b**] *string*

Returns successfully if **string** matches the previously-compiled PCRE.

Upon successful match, if the expression captures substrings within parentheses, **pcre_match** will set the array **match** to those substrings, unless the **-a** option is given, in which case it will set the array *arr*. Similarly, the variable **MATCH** will be set to the entire matched portion of the string, unless the **-v** option is given, in which case the variable *var* will be set. No variables are altered if there is no successful match. A **-n** option starts searching for a match from the byte *offset* position in *string*. If the **-b** option is given, the variable **ZPCRE_OP** will be set to an offset pair string, representing the byte offset positions of the entire matched portion within the *string*. For example, a **ZPCRE_OP** set to "32 45" indicates that the matched portion began on byte offset 32

and ended on byte offset 44. Here, byte offset position 45 is the position directly after the matched portion. Keep in mind that the byte position isn't necessarily the same as the character position when UTF-8 characters are involved. Consequently, the byte offset positions are only to be relied on in the context of using them for subsequent searches on *string*, using an offset position as an argument to the `-n` option. This is mostly used to implement the "find all non-overlapping matches" functionality.

A simple example of "find all non-overlapping matches":

```
string="The following zip codes: 78884 90210 99513"
pcre_compile -m "\d{5}"
accum=()
pcre_match -b -- $string
while [[ $? -eq 0 ]] do
    b=(=${ZPCRE_OP})
    accum+=${MATCH}
    pcre_match -b -n ${b[2]} -- $string
done
print -l $accum
```

The `zsh/pcre` module makes available the following test condition:

`expr -pcre-match pcre`

Matches a string against a perl-compatible regular expression.

For example,

```
[[ "$text" -pcre-match ^d+$ ]] &&
print text variable contains only "d's".
```

If the `REMATCH_PCRE` option is set, the `=~` operator is equivalent to `-pcre-match`, and the `NO_CASE_MATCH` option may be used. Note that `NO_CASE_MATCH` never applies to the `pcre_match` builtin, instead use the `-i` switch of `pcre_compile`.

THE ZSH/PARAM/PRIVATE MODULE

The `zsh/param/private` module is used to create parameters whose scope is limited to the current function body, and *not* to other functions called by the current function.

This module provides a single autoloaded builtin:

private [{+|-}AHUahlprtux] [{+|-}EFLRZi [*n*]] [*name*[=*value*] ...]

The `private` builtin accepts all the same options and arguments as `local` (`zshbuiltins(1)`) except for the `-T` option. Tied parameters may not be made private.

If used at the top level (outside a function scope), `private` creates a normal parameter in the same manner as `declare` or `typeset`. A warning about this is printed if `WARN_CREATE_GLOBAL` is set (`zshoptions(1)`). Used inside a function scope, `private` creates a local parameter similar to one declared with `local`, except having special properties noted below.

Special parameters which expose or manipulate internal shell state, such as `ARGC`, `argv`, `COLUMNS`, `LINES`, `UID`, `EUID`, `IFS`, `PROMPT`, `RANDOM`, `SECONDS`, etc., cannot be made private unless the `-h` option is used to hide the special meaning of the parameter. This may change in the future.

As with other `typeset` equivalents, `private` is both a builtin and a reserved word, so arrays may be assigned with parenthesized word list `name=(value...)` syntax. However, the reserved word `'private'` is not available until `zsh/param/private` is loaded, so care must be taken with order of execution and parsing for function definitions which use `private`. To compensate for this, the module also adds the option `'-P'` to the `'local'` builtin to declare private parameters.

For example, this construction fails if `zsh/param/private` has not yet been loaded when `'bad_declaration'` is defined:

```

bad_declaration() {
  zmodload zsh/param/private
  private array=( one two three )
}

```

This construction works because **local** is already a keyword, and the module is loaded before the statement is executed:

```

good_declaration() {
  zmodload zsh/param/private
  local -P array=( one two three )
}

```

The following is usable in scripts but may have trouble with **autoload**:

```

zmodload zsh/param/private
iffy_declaration() {
  private array=( one two three )
}

```

The **private** builtin may always be used with scalar assignments and for declarations without assignments.

Parameters declared with **private** have the following properties:

- Within the function body where it is declared, the parameter behaves as a local, except as noted above for tied or special parameters.
- The type of a parameter declared private cannot be changed in the scope where it was declared, even if the parameter is unset. Thus an array cannot be assigned to a private scalar, etc.
- Within any other function called by the declaring function, the private parameter does *NOT* hide other parameters of the same name, so for example a global parameter of the same name is visible and may be assigned or unset. This includes calls to anonymous functions, although that may also change in the future.
- An exported private remains in the environment of inner scopes but appears unset for the current shell in those scopes. Generally, exporting private parameters should be avoided.

Note that this differs from the static scope defined by compiled languages derived from C, in that the a new call to the same function creates a new scope, i.e., the parameter is still associated with the call stack rather than with the function definition. It differs from ksh '**typeset -S**' because the syntax used to define the function has no bearing on whether the parameter scope is respected.

THE ZSH/REGEX MODULE

The **zsh/regex** module makes available the following test condition:

expr -regex-match regex

Matches a string against a POSIX extended regular expression. On successful match, matched portion of the string will normally be placed in the **MATCH** variable. If there are any capturing parentheses within the regex, then the **match** array variable will contain those. If the match is not successful, then the variables will not be altered.

For example,

```

[[ alphabetical -regex-match ^a([a]+)a([a]+)a ]] &&
print -l $MATCH X $match

```

If the option **REMATCH_PCRE** is not set, then the **=~** operator will automatically load this module as needed and will invoke the **-regex-match** operator.

If **BASH_REMATCH** is set, then the array **BASH_REMATCH** will be set instead of **MATCH** and **match**.

THE ZSH/SCHED MODULE

The **zsh/sched** module makes available one builtin command and one parameter.

sched [-o] [+]hh:mm[:ss] *command* ...

sched [-o] [+]seconds *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, and either as hours, minutes and (optionally) seconds separated by a colon, or seconds alone. An absolute number of seconds indicates the time since the epoch (1970/01/01 00:00); this is useful in combination with the features in the **zsh/datetime** module, see the **zsh/datetime** module entry in *zshmodules(1)*.

With no arguments, prints the list of scheduled commands. If the scheduled command has the **-o** flag set, this is shown at the start of the command.

With the argument *'-item'*, removes the given item from the list. The numbering of the list is continuous and entries are in time order, so the numbering can change when entries are added or deleted.

Commands are executed either immediately before a prompt, or while the shell's line editor is waiting for input. In the latter case it is useful to be able to produce output that does not interfere with the line being edited. Providing the option **-o** causes the shell to clear the command line before the event and redraw it afterwards. This should be used with any scheduled event that produces visible output to the terminal; it is not needed, for example, with output that updates a terminal emulator's title bar.

To effect changes to the editor buffer when an event executes, use the **'zle'** command with no arguments to test whether the editor is active, and if it is, then use **'zle widget'** to access the editor via the named *widget*.

The **sched** builtin is not made available by default when the shell starts in a mode emulating another shell. It can be made available with the command **'zmodload -F zsh/sched b:sched'**.

zsh_scheduled_events

A readonly array corresponding to the events scheduled by the **sched** builtin. The indices of the array correspond to the numbers shown when **sched** is run with no arguments (provided that the **KSH_ARRAYS** option is not set). The value of the array consists of the scheduled time in seconds since the epoch (see the section 'The **zsh/datetime** Module' for facilities for using this number), followed by a colon, followed by any options (which may be empty but will be preceded by a *'-'* otherwise), followed by a colon, followed by the command to be executed.

The **sched** builtin should be used for manipulating the events. Note that this will have an immediate effect on the contents of the array, so that indices may become invalid.

THE ZSH/NET/SOCKET MODULE

The **zsh/net/socket** module makes available one builtin command:

zsocket [-altv] [-d fd] [*args*]

zsocket is implemented as a builtin to allow full use of shell command line editing, file I/O, and job control mechanisms.

Outbound Connections

zsocket [-v] [-d fd] *filename*

Open a new Unix domain connection to *filename*. The shell parameter **REPLY** will be set to the file descriptor associated with that connection. Currently, only stream connections are supported.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

In order to elicit more verbose output, use **-v**.

File descriptors can be closed with normal shell syntax when no longer needed, for example:

```
exec {REPLY}>&-
```

Inbound Connections

zsocket **-l** [**-v**] [**-d** *fd*] *filename*

zsocket **-l** will open a socket listening on *filename*. The shell parameter **REPLY** will be set to the file descriptor associated with that listener. The file descriptor remains open in subshells and forked external executables.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

In order to elicit more verbose output, use **-v**.

zsocket **-a** [**-tv**] [**-d** *targetfd*] *listenfd*

zsocket **-a** will accept an incoming connection to the socket associated with *listenfd*. The shell parameter **REPLY** will be set to the file descriptor associated with the inbound connection. The file descriptor remains open in subshells and forked external executables.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

If **-t** is specified, **zsocket** will return if no incoming connection is pending. Otherwise it will wait for one.

In order to elicit more verbose output, use **-v**.

THE ZSH/STAT MODULE

The **zsh/stat** module makes available one builtin command under two possible names:

zstat [**-gnNolLtTrs**] [**-f** *fd*] [**-H** *hash*] [**-A** *array*] [**-F** *fmt*]
[**+element**] [*file ...*]

stat ... The command acts as a front end to the **stat** system call (see *stat(2)*). The same command is provided with two names; as the name **stat** is often used by an external command it is recommended that only the **zstat** form of the command is used. This can be arranged by loading the module with the command '**zmodload -F zsh/stat b:zstat**'.

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 (“**zstat +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 *fnt* Supplies a **strftime** (see *strftime(3)*) string for the formatting of the time elements. The format string supports all of the zsh extensions described in the section EXPANSION OF PROMPT SEQUENCES in *zshmisc(1)*. 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; arguments, and options other than **-A**, are ignored.

-L Perform an **lstat** (see *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. It’s important to note that this is the exact opposite from *ls(1)*, etc.

-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.

THE ZSH/SYSTEM MODULE

The **zsh/system** module makes available various builtin commands and parameters.

Builtins

syserror [**-e** *errvar*] [**-p** *prefix*] [*errno* | *errname*]

This command prints out the error message associated with *errno*, a system error number, followed by a newline to standard error.

Instead of the error number, a name *errname*, for example **ENOENT**, may be used. The set of names is the same as the contents of the array **errnos**, see below.

If the string *prefix* is given, it is printed in front of the error message, with no intervening space.

If *errvar* is supplied, the entire message, without a newline, is assigned to the parameter names *errvar* and nothing is output.

A return status of 0 indicates the message was successfully printed (although it may not be useful if the error number was out of the system's range), a return status of 1 indicates an error in the parameters, and a return status of 2 indicates the error name was not recognised (no message is printed for this).

sysopen [**-arw**] [**-m** *permissions*] [**-o** *options*]

-u *fd file*

This command opens a file. The **-r**, **-w** and **-a** flags indicate whether the file should be opened for reading, writing and appending, respectively. The **-m** option allows the initial permissions to use when creating a file to be specified in octal form. The file descriptor is specified with **-u**. Either an explicit file descriptor in the range 0 to 9 can be specified or a variable name can be given to which the file descriptor number will be assigned.

The **-o** option allows various system specific options to be specified as a comma-separated list. The following is a list of possible options. Note that, depending on the system, some may not be available.

cloexec mark file to be closed when other programs are executed (else the file descriptor remains open in subshells and forked external executables)

create

creat create file if it does not exist

excl create file, error if it already exists

noatime

suppress updating of the file atime

nofollow

fail if *file* is a symbolic link

sync request that writes wait until data has been physically written

truncate

trunc truncate file to size 0

To close the file, use one of the following:

exec {*fd*}<&-

exec {*fd*}>&-

sysread [**-c** *countvar*] [**-i** *infd*] [**-o** *outfd*]

[**-s** *bufsize*] [**-t** *timeout*] [*param*]

Perform a single system read from file descriptor *infd*, or zero if that is not given. The result of the read is stored in *param* or **REPLY** if that is not given. If *countvar* is given, the number of bytes read is assigned to the parameter named by *countvar*.

The maximum number of bytes read is *bufsize* or 8192 if that is not given, however the command returns as soon as any number of bytes was successfully read.

If *timeout* is given, it specifies a timeout in seconds, which may be zero to poll the file descriptor.

This is handled by the **poll** system call if available, otherwise the **select** system call if available.

If *outfd* is given, an attempt is made to write all the bytes just read to the file descriptor *outfd*. If this fails, because of a system error other than **EINTR** or because of an internal zsh error during an interrupt, the bytes read but not written are stored in the parameter named by *param* if supplied (no default is used in this case), and the number of bytes read but not written is stored in the parameter named by *countvar* if that is supplied. If it was successful, *countvar* contains the full number of bytes transferred, as usual, and *param* is not set.

The error **EINTR** (interrupted system call) is handled internally so that shell interrupts are transparent to the caller. Any other error causes a return.

The possible return statuses are

- 0 At least one byte of data was successfully read and, if appropriate, written.
- 1 There was an error in the parameters to the command. This is the only error for which a message is printed to standard error.
- 2 There was an error on the read, or on polling the input file descriptor for a timeout. The parameter **ERRNO** gives the error.
- 3 Data were successfully read, but there was an error writing them to *outfd*. The parameter **ERRNO** gives the error.
- 4 The attempt to read timed out. Note this does not set **ERRNO** as this is not a system error.
- 5 No system error occurred, but zero bytes were read. This usually indicates end of file. The parameters are set according to the usual rules; no write to *outfd* is attempted.

sysseek [**-u** *fd*] [**-w** *start|end|current*] *offset*

The current file position at which future reads and writes will take place is adjusted to the specified byte offset. The *offset* is evaluated as a math expression. The **-u** option allows the file descriptor to be specified. By default the offset is specified relative to the start or the file but, with the **-w** option, it is possible to specify that the offset should be relative to the current position or the end of the file.

syswrite [**-c** *countvar*] [**-o** *outfd*] *data*

The data (a single string of bytes) are written to the file descriptor *outfd*, or 1 if that is not given, using the **write** system call. Multiple write operations may be used if the first does not write all the data.

If *countvar* is given, the number of byte written is stored in the parameter named by *countvar*; this may not be the full length of *data* if an error occurred.

The error **EINTR** (interrupted system call) is handled internally by retrying; otherwise an error causes the command to return. For example, if the file descriptor is set to non-blocking output, an error **EAGAIN** (on some systems, **EWOULDBLOCK**) may result in the command returning early.

The return status may be 0 for success, 1 for an error in the parameters to the command, or 2 for an error on the write; no error message is printed in the last case, but the parameter **ERRNO** will reflect the error that occurred.

zsystem flock [**-t** *timeout*] [**-f** *var*] [**-er**] *file*

zsystem flock **-u** *fd_expr*

The builtin **zsystem**'s subcommand **flock** performs advisory file locking (via the *fcntl(2)* system call) over the entire contents of the given file. This form of locking requires the processes accessing the file to cooperate; its most obvious use is between two instances of the shell itself.

In the first form the named *file*, which must already exist, is locked by opening a file descriptor to the file and applying a lock to the file descriptor. The lock terminates when the shell process that created the lock exits; it is therefore often convenient to create file locks within subshells, since the

lock is automatically released when the subshell exits. Note that use of the **print** builtin with the **-u** option will, as a side effect, release the lock, as will redirection to the file in the shell holding the lock. To work around this use a subshell, e.g. **(print message) >> file**. Status 0 is returned if the lock succeeds, else status 1.

In the second form the file descriptor given by the arithmetic expression *fd_expr* is closed, releasing a lock. The file descriptor can be queried by using the **-f var** form during the lock; on a successful lock, the shell variable *var* is set to the file descriptor used for locking. The lock will be released if the file descriptor is closed by any other means, for example using **exec {var}>&-**; however, the form described here performs a safety check that the file descriptor is in use for file locking.

By default the shell waits indefinitely for the lock to succeed. The option **-t timeout** specifies a timeout for the lock in seconds; currently this must be an integer. The shell will attempt to lock the file once a second during this period. If the attempt times out, status 2 is returned.

If the option **-e** is given, the file descriptor for the lock is preserved when the shell uses **exec** to start a new process; otherwise it is closed at that point and the lock released.

If the option **-r** is given, the lock is only for reading, otherwise it is for reading and writing. The file descriptor is opened accordingly.

zsystem supports subcommand

The builtin **zsystem**'s subcommand **supports** tests whether a given subcommand is supported. It returns status 0 if so, else status 1. It operates silently unless there was a syntax error (i.e. the wrong number of arguments), in which case status 255 is returned. Status 1 can indicate one of two things: *subcommand* is known but not supported by the current operating system, or *subcommand* is not known (possibly because this is an older version of the shell before it was implemented).

Math Functions

systell(fd)

The **systell** math function returns the current file position for the file descriptor passed as an argument.

Parameters

errno A readonly array of the names of errors defined on the system. These are typically macros defined in C by including the system header file **errno.h**. The index of each name (assuming the option **KSH_ARRAYS** is unset) corresponds to the error number. Error numbers *num* before the last known error which have no name are given the name **Enum** in the array.

Note that aliases for errors are not handled; only the canonical name is used.

sysparams

A readonly associative array. The keys are:

pid Returns the process ID of the current process, even in subshells. Compare **\$\$**, which returns the process ID of the main shell process.

ppid Returns the process ID of the parent of the current process, even in subshells. Compare **\$PPID**, which returns the process ID of the parent of the main shell process.

procsubstpid

Returns the process ID of the last process started for process substitution, i.e. the **<(…)** and **>(…)** expansions.

THE ZSH/NET/TCP MODULE

The **zsh/net/tcp** module makes available one builtin command:

ztcp [**-acfltv**] [**-d fd**] [*args*]

ztcp is implemented as a builtin to allow full use of shell command line editing, file I/O, and job control mechanisms.

If **ztcp** is run with no options, it will output the contents of its session table.

If it is run with only the option **-L**, it will output the contents of the session table in a format suitable for automatic parsing. The option is ignored if given with a command to open or close a session. The output consists of a set of lines, one per session, each containing the following elements separated by spaces:

File descriptor

The file descriptor in use for the connection. For normal inbound (**I**) and outbound (**O**) connections this may be read and written by the usual shell mechanisms. However, it should only be close with '**ztcp -c**'.

Connection type

A letter indicating how the session was created:

- Z** A session created with the **zftp** command.
- L** A connection opened for listening with '**ztcp -l**'.
- I** An inbound connection accepted with '**ztcp -a**'.
- O** An outbound connection created with '**ztcp host ...**'.

The local host

This is usually set to an all-zero IP address as the address of the localhost is irrelevant.

The local port

This is likely to be zero unless the connection is for listening.

The remote host

This is the fully qualified domain name of the peer, if available, else an IP address. It is an all-zero IP address for a session opened for listening.

The remote port

This is zero for a connection opened for listening.

Outbound Connections

ztcp [**-v**] [**-d** *fd*] *host* [*port*]

Open a new TCP connection to *host*. If the *port* is omitted, it will default to port 23. The connection will be added to the session table and the shell parameter **REPLY** will be set to the file descriptor associated with that connection.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

In order to elicit more verbose output, use **-v**.

Inbound Connections

ztcp -l [**-v**] [**-d** *fd*] *port*

ztcp -l will open a socket listening on TCP *port*. The socket will be added to the session table and the shell parameter **REPLY** will be set to the file descriptor associated with that listener.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

In order to elicit more verbose output, use **-v**.

ztcp -a [**-tv**] [**-d** *targetfd*] *listenfd*

ztcp -a will accept an incoming connection to the port associated with *listenfd*. The connection will be added to the session table and the shell parameter **REPLY** will be set to the file descriptor associated with the inbound connection.

If **-d** is specified, its argument will be taken as the target file descriptor for the connection.

If **-t** is specified, **ztcp** will return if no incoming connection is pending. Otherwise it will wait for one.

In order to elicit more verbose output, use **-v**.

Closing Connections**ztcp -cf** [-v] [*fd*]**ztcp -c** [-v] [*fd*]

ztcp -c will close the socket associated with *fd*. The socket will be removed from the session table. If *fd* is not specified, **ztcp** will close everything in the session table.

Normally, sockets registered by **zftp** (see *zshmodules(1)*) cannot be closed this way. In order to force such a socket closed, use **-f**.

In order to elicit more verbose output, use **-v**.

Example

Here is how to create a TCP connection between two instances of **zsh**. We need to pick an unassigned port; here we use the randomly chosen 5123.

On **host1**,

zmodload zsh/net/tcp**ztcp -l 5123****listenfd=\$REPLY****ztcp -a \$listenfd****fd=\$REPLY**

The second from last command blocks until there is an incoming connection.

Now create a connection from **host2** (which may, of course, be the same machine):

zmodload zsh/net/tcp**ztcp host1 5123****fd=\$REPLY**

Now on each host, **\$fd** contains a file descriptor for talking to the other. For example, on **host1**:

print This is a message >&\$fd

and on **host2**:

read -r line <&\$fd; print -r - \$line

prints 'This is a message'.

To tidy up, on **host1**:

ztcp -c \$listenfd**ztcp -c \$fd**

and on **host2**

ztcp -c \$fd**THE ZSH/TERMCAP MODULE**

The **zsh/termcap** module makes available one builtin command:

echohc 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.

THE ZSH/TERMINFO MODULE

The **zsh/terminfo** module makes available one builtin command:

echohi cap [*arg*]

Output the terminfo value corresponding to the capability *cap*, instantiated with *arg* if applicable.

The **zsh/terminfo** module makes available one parameter:

terminfo

An associative array that maps terminfo capability names to their values.

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 *zshzftpsys(1)*. 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.

Subcommands

open *host[:port]* [*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. If the argument is in the form *host:port*, open a connection to TCP port *port* instead of the standard FTP port 21. This may be the name of a TCP service or a number: see the description of **ZFTP_PORT** below for more information.

If IPv6 addresses in colon format are used, the *host* should be surrounded by quoted square brackets to distinguish it from the *port*, for example '[fe80::203:baff:fe02:8b56]'. For consistency this is allowed with all forms of *host*.

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_PORT**, **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 status 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 [*arg ...*]

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 [*arg ...*]

Give a (short) listing of the remote directory. With no *arg*, 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 bytes, 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 *file ...*

local [*file ...*]

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 *files* 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 *arg* ...

Send a host-specific command to the server. You will probably only need this if instructed by the server to use it.

quote *arg* ...

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_PORT**, **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.

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_PORT

Readonly. The number of the remote TCP port to which the connection is open (even if the port was originally specified as a named service). Usually this is the standard FTP port, 21.

In the unlikely event that your system does not have the appropriate conversion functions, this appears in network byte order. If your system is little-endian, the port then consists of two swapped bytes and the standard port will be reported as 5376. In that case, numeric ports passed to **zftp open** will also need to be in this format.

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 '**T**'. 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.

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.

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.

THE ZSH/ZLE MODULE

The **zsh/zle** module contains the Zsh Line Editor. See *zshzle(1)*.

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 *zshzle(1)*).

keymaps

This array contains the names of the keymaps currently defined.

widgets

This associative array contains one entry per widget. 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,

a string of the form **'completion:type:name'**

for completion widgets,

or a null value if the widget is not yet fully defined. In the penultimate case, *type* is the name of the builtin widget the completion widget imitates in its behavior and *name* is the name of the shell function implementing the completion widget.

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 descendants, the average time in milliseconds spent in the function and its descendants per call and the percentage of time spent in all shell functions used in this function and its descendants. 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 descendants. 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 descendants only for the times when it was called from that particular calling function. Likewise, for a called function, this column lists the total time spent in the called function and its descendants 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.

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.

The shell parameter **REPLY** is set to the file descriptor assigned to the master side of the pseudo-terminal. This allows the terminal to be monitored with ZLE descriptor handlers (see *zsh-zle(1)*) or manipulated with **sysread** and **syswrite** (see THE ZSH/SYSTEM MODULE in *zsh-modules(1)*). *Warning*: Use of **sysread** and **syswrite** is *not* recommended; use **zpty -r** and **zpty -w** unless you know exactly what you are doing.

zpty -d [*name* ...]

The second form, with the **-d** option, is used to delete commands previously started, by supplying a list of their *names*. If no *name* is given, all commands are deleted. Deleting a command causes the HUP signal to be sent to the corresponding process.

zpty -w [**-n**] *name* [*string* ...]

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 *string* is 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. The exact input is always copied: the **-n** option is not applied.

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 [**-mt**] *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 status 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 status 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 status is zero if the string read matches the pattern, or if the command has exited but at least one character could still be read. If the option **-m** is present, the return status is zero only if the pattern matches. 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 status is non-zero.

In all cases, the return status 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 status **1**. When used with a *pattern*, the behaviour on a failed poll is similar to when the command has exited: the return value is zero if at least one character could still be read even if the pattern failed to match.

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 status 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.

THE ZSH/ZSELECT MODULE

The **zsh/zselect** module makes available one builtin command:

zselect [**-rwe**] [**-t** *timeout*] [**-a** *array*] [**-A** *assoc*] [*fd* ...]

The **zselect** builtin is a front-end to the ‘select’ system call, which blocks until a file descriptor is ready for reading or writing, or has an error condition, with an optional timeout. If this is not available on your system, the command prints an error message and returns status 2 (normal errors return status 1). For more information, see your systems documentation for *select*(3). Note there is no connection with the shell builtin of the same name.

Arguments and options may be intermingled in any order. Non-option arguments are file descriptors, which must be decimal integers. By default, file descriptors are to be tested for reading, i.e.

zselect will return when data is available to be read from the file descriptor, or more precisely, when a read operation from the file descriptor will not block. After a **-r**, **-w** and **-e**, the given file descriptors are to be tested for reading, writing, or error conditions. These options and an arbitrary list of file descriptors may be given in any order.

(The presence of an ‘error condition’ is not well defined in the documentation for many implementations of the select system call. According to recent versions of the POSIX specification, it is really an *exception* condition, of which the only standard example is out-of-band data received on a socket. So zsh users are unlikely to find the **-e** option useful.)

The option ‘**-t timeout**’ specifies a timeout in hundredths of a second. This may be zero, in which case the file descriptors will simply be polled and **zselect** will return immediately. It is possible to call **zselect** with no file descriptors and a non-zero timeout for use as a finer-grained replacement for ‘sleep’; note, however, the return status is always 1 for a timeout.

The option ‘**-a array**’ indicates that *array* should be set to indicate the file descriptor(s) which are ready. If the option is not given, the array **reply** will be used for this purpose. The array will contain a string similar to the arguments for **zselect**. For example,

```
zselect -t 0 -r 0 -w 1
```

might return immediately with status 0 and **\$reply** containing ‘**-r 0 -w 1**’ to show that both file descriptors are ready for the requested operations.

The option ‘**-A assoc**’ indicates that the associative array *assoc* should be set to indicate the file descriptor(s) which are ready. This option overrides the option **-a**, nor will **reply** be modified. The keys of **assoc** are the file descriptors, and the corresponding values are any of the characters ‘**rwe**’ to indicate the condition.

The command returns status 0 if some file descriptors are ready for reading. If the operation timed out, or a timeout of 0 was given and no file descriptors were ready, or there was an error, it returns status 1 and the array will not be set (nor modified in any way). If there was an error in the select operation the appropriate error message is printed.

THE ZSH/ZUTIL MODULE

The **zsh/zutil** module only adds some builtins:

```
zstyle [ -L [ metapattern [ style ] ] ]
zstyle [ -e | - | -- ] pattern style string ...
zstyle -d [ pattern [ style ... ] ]
zstyle -g name [ pattern [ style ] ]
zstyle -{a|b|s} context style name [ sep ]
zstyle -{T|t} context style [ string ... ]
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 matched against the patterns. The definition stored for the most specific pattern that matches will be returned.

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 ‘*’. A ‘*’ in the pattern will match zero or more characters in the context; colons are not treated specially in this regard. If two patterns are equally specific, the tie is broken in favour of the pattern that was defined first.

Example

For example, to define your preferred form of precipitation depending on which city you’re in, you might set the following in your **zshrc**:

```
zstyle 'weather:europa:*' preferred-precipitation rain
```

```
zstyle ':weather:europa:germany:* preferred-precipitation none
zstyle ':weather:europa:germany:*:munich' preferred-precipitation snow
```

Then, the fictional **'weather'** plugin might run under the hood a command such as

```
zstyle -s ":weather:${continent}:${country}:${county}:${city}" preferred-precipitation REPLY
```

in order to retrieve your preference into the scalar variable **\$REPLY**.

Usage

The forms that operate on patterns are the following.

zstyle [**-L** [*metapattern* [*style*]]]

Without arguments, lists style definitions. Styles are shown in alphabetic order and patterns are shown in the order **zstyle** will test them.

If the **-L** option is given, listing is done in the form of calls to **zstyle**. The optional first argument, *metapattern*, is a pattern which will be matched against the string supplied as *pattern* when the style was defined. Note: this means, for example, **'zstyle -L ":completion:*"'** will match any supplied pattern beginning **'completion:'**, not just **":completion:*"**: use **':completion:*'** to match that. The optional second argument limits the output to a specific *style* (not a pattern). **-L** is not compatible with any other options.

zstyle [**-** | **--** | **-e**] *pattern style string ...*

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* [*style ...*]]

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 styles for a given context.

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.

Return **0** if the style is set, **1** otherwise.

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'**.

Return **0** if *name* is set to **'yes'**, **1** otherwise.

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.

Return **0** if the style is set, **1** otherwise.

zstyle -t *context style [string ...]*

zstyle -T *context style [string ...]*

Test the value of a style, i.e. the **-t** option only returns a status (sets **\$?**). Without any *string* 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 defined but doesn't match, the return status is **1**. If the style is not defined, the status is **2**.

The **-T** option tests the values of the style like **-t**, but it returns status 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 spec ...*

zformat -a *array sep spec ...*

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.maxc'*, 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 *%*-escapes also understand ternary expressions in the form used by prompts. The *%* is followed by a *'(* and then an ordinary format specifier character as described above. There may be a set of digits either before or after the *'(*; these specify a test number, which defaults to zero. Negative numbers are also allowed. An arbitrary delimiter character follows the format specifier, which is followed by a piece of *'true'* text, the delimiter character again, a piece of *'false'* text, and a closing parenthesis. The complete expression (without the digits) thus looks like *'%(X.text1.text2)'*, except that the *'.'* character is arbitrary. The value given for the format specifier in the *char:string* expressions is evaluated as a mathematical expression, and compared with the test number. If they are the same, *text1* is output, else *text2* is output. A parenthesis may be escaped in *text2* as *%*). Either of *text1* or *text2* may contain nested *%*-escapes.

For example:

```
zformat -f REPLY "The answer is '%3(c.yes.no)'." c:3
```

outputs "The answer is 'yes'." to **REPLY** since the value for the format specifier **c** is 3, agreeing with the digit argument to the ternary expression.

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. A colon in the *left* string can be escaped with a backslash. The resulting strings are stored in the *array*.

zregexparse

This implements some internals of the `_regex_arguments` function.

zparseopts [**-D -E -F -K -M**] [**-a array**] [**-A assoc**] [**-**] *spec* ...

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, which should be declared as a normal array and never as an associative 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 `'--'`. See also **-F**.

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.

In all cases, option-arguments must appear either immediately following the option in the same positional parameter or in the next one. Even an optional argument may appear in the next parameter, unless it begins with a `'-'`. There is no special handling of `'='` as with GNU-style argument parsers; given the *spec* `'-foo:'`, the positional parameter `'--foo=bar'` is parsed as `'--foo'` with an argument of `'=bar'`.

When the names of two options that take no arguments overlap, the longest one wins, so that parsing for the *specs* `'-foo -foobar'` (for example) is unambiguous. However, due to the aforementioned handling of option-arguments, ambiguities may arise when at least one overlapping *spec* takes an argument, as in `'-foo: -foobar'`. In that case, the last matching *spec* wins.

The options of **zparseopts** itself cannot be stacked because, for example, the stack `'-DEK'` is indistinguishable from a *spec* for the GNU-style long option `'--DEK'`. 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*. If the first such parameter is ‘-’ or ‘--’, it is removed as well. This is similar to using the **shift** builtin.
- E**
This changes the parsing rules to *not* stop at the first string that isn’t described by one of the *specs*. 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. As indicated above, parsing still stops at the first ‘-’ or ‘--’ not described by a *spec*, but it is not removed when used with **-D**.
- F**
If this option is given, **zparseopts** immediately stops at the first option-like parameter not described by one of the *specs*, prints an error message, and returns status 1. Removal (**-D**) and extraction (**-E**) are not performed, and option arrays are not updated. This provides basic validation for the given options.

Note that the appearance in the positional parameters of an option without its required argument always aborts parsing and returns an error as described above regardless of whether this option is used.
- K**
With this option, the arrays specified with the **-a** option and with the ‘=array’ forms are kept unchanged when none of the *specs* for them is used. Otherwise the entire array is replaced when any of the *specs* is used. Individual elements of associative arrays specified with the **-A** option are preserved by **-K**. This allows assignment of default values to arrays before calling **zparseopts**.
- M**
This changes the assignment rules to implement a map among equivalent option names. If any *spec* uses the ‘=array’ form, the string *array* is interpreted as the name of another *spec*, which is used to choose where to store the values. If no other *spec* is found, the values are stored as usual. This changes only the way the values are stored, not the way \$* is parsed, so results may be unpredictable if the ‘name+’ specifier is used inconsistently.

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**.

The **-M** option can be used like this:

```
set -- -a -bx -c y -cz baz -cend
zparseopts -A bar -M a=foo b+: c:=b
```

to have the effect of

```
foo=(-a)  
bar=(-a ” -b xyz)
```