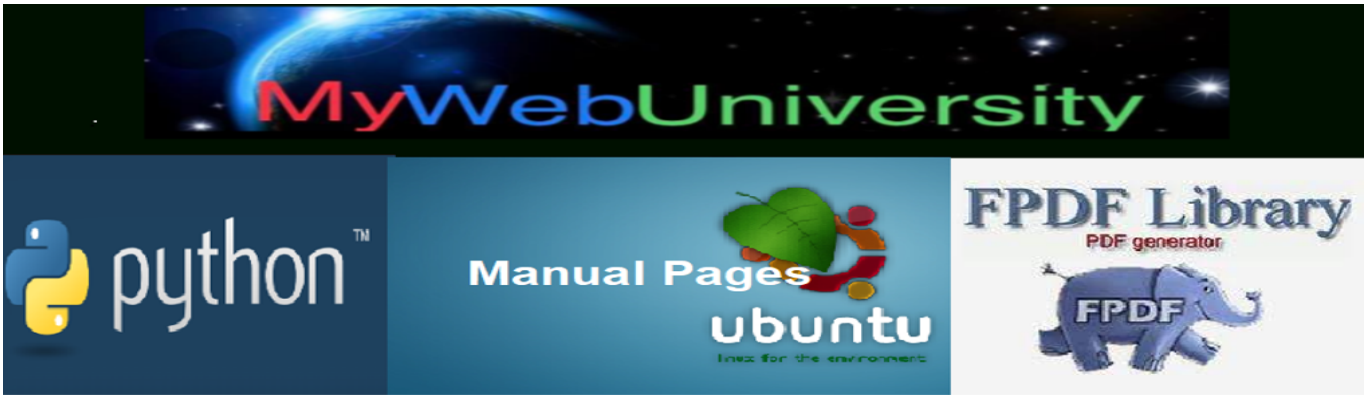

Rocky Enterprise Linux 9.2 Manual Pages on command 'zshcompsys.1'

$ man zshcompsys.1

ZSHCOMPSYS(1) General Commands Manual ZSHCOMPSYS(1)

NAME

zshcompsys - zsh completion system

DESCRIPTION

This describes the shell code for the `new' completion system, referred to as compsys. It is written in shell functions based on the features described in zshcompwid(1).

The features are contextual, sensitive to the point at which completion is started. Many completions are already provided. For this reason, a user can perform a great many tasks without knowing any details beyond how to initialize the system, which is described below in INITIALIZATION.

The context that decides what completion is to be performed may be

? an argument or option position: these describe the position on the command line at which completion is requested. For example `first argument to rmdir, the word be? ing completed names a directory';

? a special context, denoting an element in the shell's syntax. For example `a word in command position' or `an array subscript'.

A full context specification contains other elements, as we shall describe.

Besides commands names and contexts, the system employs two more concepts, styles and tags. These provide ways for the user to configure the system's behaviour.

Tags play a dual role. They serve as a classification system for the matches, typically indicating a class of object that the user may need to distinguish. For example, when completing arguments of the ls command the user may prefer to try files before directo? ries, so both of these are tags. They also appear as the rightmost element in a context

specification.

Styles modify various operations of the completion system, such as output formatting, but also what kinds of completers are used (and in what order), or which tags are examined. Styles may accept arguments and are manipulated using the zstyle command described in see zshmodules(1).

In summary, tags describe what the completion objects are, and style how they are to be completed. At various points of execution, the completion system checks what styles and/or tags are defined for the current context, and uses that to modify its behavior.

The full description of context handling, which determines how tags and other elements of the context influence the behaviour of styles, is described below in COMPLETION SYSTEM CONFIGURATION.

When a completion is requested, a dispatcher function is called; see the description of _main_complete in the list of control functions below. This dispatcher decides which func? tion should be called to produce the completions, and calls it. The result is passed to one or more completers, functions that implement individual completion strategies: simple completion, error correction, completion with error correction, menu selection, etc.

More generally, the shell functions contained in the completion system are of two types:

? those beginning `comp' are to be called directly; there are only a few of these;

? those beginning `_' are called by the completion code. The shell functions of this set, which implement completion behaviour and may be bound to keystrokes, are re? ferred to as `widgets'. These proliferate as new completions are required.

INITIALIZATION

If the system was installed completely, it should be enough to call the shell function compinit from your initialization file; see the next section. However, the function compinstall can be run by a user to configure various aspects of the completion system. Usually, compinstall will insert code into .zshrc, although if that is not writable it will save it in another file and tell you that file's location. Note that it is up to you to make sure that the lines added to .zshrc are actually run; you may, for example, need to move them to an earlier place in the file if .zshrc usually returns early. So long as you keep them all together (including the comment lines at the start and finish), you can rerun compinstall and it will correctly locate and modify these lines. Note, however, that any code you add to this section by hand is likely to be lost if you rerun compin? stall, although lines using the command `zstyle' should be gracefully handled.

The new code will take effect next time you start the shell, or run .zshrc by hand; there is also an option to make them take effect immediately. However, if compinstall has re? moved definitions, you will need to restart the shell to see the changes.

To run compinstall you will need to make sure it is in a directory mentioned in your fpath parameter, which should already be the case if zsh was properly configured as long as your startup files do not remove the appropriate directories from fpath. Then it must be au? toloaded (`autoload -U compinstall' is recommended). You can abort the installation any time you are being prompted for information, and your .zshrc will not be altered at all; changes only take place right at the end, where you are specifically asked for confirma? tion.

Use of compinit

This section describes the use of compinit to initialize completion for the current ses? sion when called directly; if you have run compinstall it will be called automatically from your .zshrc.

To initialize the system, the function compinit should be in a directory mentioned in the fpath parameter, and should be autoloaded (`autoload -U compinit' is recommended), and then run simply as `compinit'. This will define a few utility functions, arrange for all the necessary shell functions to be autoloaded, and will then re-define all widgets that do completion to use the new system. If you use the menu-select widget, which is part of the zsh/complist module, you should make sure that that module is loaded before the call to compinit so that that widget is also re-defined. If completion styles (see below) are set up to perform expansion as well as completion by default, and the TAB key is bound to expand-or-complete, compinit will rebind it to complete-word; this is necessary to use the correct form of expansion.

Should you need to use the original completion commands, you can still bind keys to the old widgets by putting a `.' in front of the widget name, e.g. `.expand-or-complete'.

To speed up the running of compinit, it can be made to produce a dumped configuration that will be read in on future invocations; this is the default, but can be turned off by call? ing compinit with the option -D. The dumped file is .zcompdump in the same directory as the startup files (i.e. $ZDOTDIR or $HOME); alternatively, an explicit file name can be given by `compinit -d dumpfile'. The next invocation of compinit will read the dumped file instead of performing a full initialization.

If the number of completion files changes, compinit will recognise this and produce a new

dump file. However, if the name of a function or the arguments in the first line of a #compdef function (as described below) change, it is easiest to delete the dump file by hand so that compinit will re-create it the next time it is run. The check performed to see if there are new functions can be omitted by giving the option -C. In this case the dump file will only be created if there isn't one already.

The dumping is actually done by another function, compdump, but you will only need to run this yourself if you change the configuration (e.g. using compdef) and then want to dump the new one. The name of the old dumped file will be remembered for this purpose.

If the parameter _compdir is set, compinit uses it as a directory where completion func? tions can be found; this is only necessary if they are not already in the function search path.

For security reasons compinit also checks if the completion system would use files not owned by root or by the current user, or files in directories that are world- or group-writable or that are not owned by root or by the current user. If such files or di? rectories are found, compinit will ask if the completion system should really be used. To avoid these tests and make all files found be used without asking, use the option -u, and to make compinit silently ignore all insecure files and directories use the option -i. This security check is skipped entirely when the -C option is given.

The security check can be retried at any time by running the function compaudit. This is the same check used by compinit, but when it is executed directly any changes to fpath are made local to the function so they do not persist. The directories to be checked may be passed as arguments; if none are given, compaudit uses fpath and _compdir to find comple? tion system directories, adding missing ones to fpath as necessary. To force a check of exactly the directories currently named in fpath, set _compdir to an empty string before calling compaudit or compinit.

The function bashcompinit provides compatibility with bash's programmable completion sys? tem. When run it will define the functions, compgen and complete which correspond to the bash builtins with the same names. It will then be possible to use completion specifica? tions and functions written for bash.

Autoloaded files

The convention for autoloaded functions used in completion is that they start with an un? derscore; as already mentioned, the fpath/FPATH parameter must contain the directory in which they are stored. If zsh was properly installed on your system, then fpath/FPATH au?

tomatically contains the required directories for the standard functions.

For incomplete installations, if compinit does not find enough files beginning with an un? derscore (fewer than twenty) in the search path, it will try to find more by adding the directory _compdir to the search path. If that directory has a subdirectory named Base, all subdirectories will be added to the path. Furthermore, if the subdirectory Base has a subdirectory named Core, compinit will add all subdirectories of the subdirectories to the path: this allows the functions to be in the same format as in the zsh source distribu? tion.

When compinit is run, it searches all such files accessible via fpath/FPATH and reads the first line of each of them. This line should contain one of the tags described below. Files whose first line does not start with one of these tags are not considered to be part of the completion system and will not be treated specially.

The tags are:

#compdef name ... [-{p|P} pattern ... [-N name ...]]

The file will be made autoloadable and the function defined in it will be called when completing names, each of which is either the name of a command whose argu? ments are to be completed or one of a number of special contexts in the form -con? text- described below.

Each name may also be of the form `cmd=service'. When completing the command cmd, the function typically behaves as if the command (or special context) service was being completed instead. This provides a way of altering the behaviour of func? tions that can perform many different completions. It is implemented by setting the parameter $service when calling the function; the function may choose to inter? pret this how it wishes, and simpler functions will probably ignore it.

If the #compdef line contains one of the options -p or -P, the words following are taken to be patterns. The function will be called when completion is attempted for a command or context that matches one of the patterns. The options -p and -P are used to specify patterns to be tried before or after other completions respec? tively. Hence -P may be used to specify default actions.

The option -N is used after a list following -p or -P; it specifies that remaining words no longer define patterns. It is possible to toggle between the three op? tions as many times as necessary.

#compdef -k style key-sequence ...

This option creates a widget behaving like the builtin widget style and binds it to the given key-sequences, if any. The style must be one of the builtin widgets that perform completion, namely complete-word, delete-char-or-list, expand-or-complete, expand-or-complete-prefix, list-choices, menu-complete, menu-expand-or-complete, or reverse-menu-complete. If the zsh/complist module is loaded (see zshmodules(1)) the widget menu-select is also available.

When one of the key-sequences is typed, the function in the file will be invoked to generate the matches. Note that a key will not be re-bound if it already was (that is, was bound to something other than undefined-key). The widget created has the same name as the file and can be bound to any other keys using bindkey as usual.

#compdef -K widget-name style key-sequence [name style seq ...]

This is similar to -k except that only one key-sequence argument may be given for each widget-name style pair. However, the entire set of three arguments may be re? peated with a different set of arguments. Note in particular that the widget-name must be distinct in each set. If it does not begin with `_' this will be added.

The widget-name should not clash with the name of any existing widget: names based on the name of the function are most useful. For example,

```
#compdef -K _foo_complete complete-word "^X^C" \
  _foo_list list-choices "^X^D"
```

(all on one line) defines a widget _foo_complete for completion, bound to `^X^C', and a widget _foo_list for listing, bound to `^X^D'.

#autoload [options]

Functions with the #autoload tag are marked for autoloading but are not otherwise treated specially. Typically they are to be called from within one of the comple? tion functions. Any options supplied will be passed to the autoload builtin; a typical use is +X to force the function to be loaded immediately. Note that the -U and -z flags are always added implicitly.

The # is part of the tag name and no white space is allowed after it. The #compdef tags use the compdef function described below; the main difference is that the name of the function is supplied implicitly.

The special contexts for which completion functions can be defined are:

-array-value-

The right hand side of an array-assignment (`name=(...)')

-brace-parameter-

The name of a parameter expansion within braces (`${...}')

-assign-parameter-

The name of a parameter in an assignment, i.e. on the left hand side of an `='

-command-

A word in command position

-condition-

A word inside a condition (`[[...]]')

-default-

Any word for which no other completion is defined

-equal-

A word beginning with an equals sign

-first-

This is tried before any other completion function. The function called may set the _compskip parameter to one of various values: all: no further completion is at? tempted; a string containing the substring patterns: no pattern completion func? tions will be called; a string containing default: the function for the `-default-' context will not be called, but functions defined for commands will be.

-math- Inside mathematical contexts, such as `((...))'

-parameter-

The name of a parameter expansion (`$...')

-redirect-

The word after a redirection operator.

-subscript-

The contents of a parameter subscript.

-tilde-

After an initial tilde (`~'), but before the first slash in the word.

-value-

On the right hand side of an assignment.

Default implementations are supplied for each of these contexts. In most cases the con? text -context- is implemented by a corresponding function _context, for example the con? text `-tilde-' and the function `_tilde').

The contexts -redirect- and -value- allow extra context-specific information. (Inter?

nally, this is handled by the functions for each context calling the function _dispatch.)

The extra information is added separated by commas.

For the -redirect- context, the extra information is in the form `-redirect-,op,command', where op is the redirection operator and command is the name of the command on the line. If there is no command on the line yet, the command field will be empty.

For the -value- context, the form is `-value-,name,command', where name is the name of the parameter on the left hand side of the assignment. In the case of elements of an associa? tive array, for example `assoc=(key <TAB>', name is expanded to `name-key'. In certain special contexts, such as completing after `make CFLAGS=', the command part gives the name of the command, here make; otherwise it is empty.

It is not necessary to define fully specific completions as the functions provided will try to generate completions by progressively replacing the elements with `-default-'. For example, when completing after `foo=<TAB>', _value will try the names `-value-,foo,' (note the empty command part), `-value-,foo,-default-' and`-value-,-default-,-default-', in that order, until it finds a function to handle the context.

As an example:

```
compdef '_files -g "*.log"' '-redirect-,2>,-default-'
```

completes files matching `*.log' after `2> <TAB>' for any command with no more specific handler defined.

Also:

```
compdef _foo -value-,-default-,-default-
```

specifies that _foo provides completions for the values of parameters for which no special function has been defined. This is usually handled by the function _value itself.

The same lookup rules are used when looking up styles (as described below); for example

```
zstyle ':completion:*:*:-redirect-,2>,*:*' file-patterns '*.log'
```

is another way to make completion after `2> <TAB>' complete files matching `*.log'.

Functions

The following function is defined by compinit and may be called directly.

compdef [-ane] function name ... [-{p|P} pattern ... [-N name ...]]

compdef -d name ...

compdef -k [-an] function style key-sequence [key-sequence ...]

compdef -K [-an] function name style key-seq [name style seq ...]

The first form defines the function to call for completion in the given contexts as

described for the #compdef tag above.

Alternatively, all the arguments may have the form `cmd=service'. Here service should already have been defined by `cmd1=service' lines in #compdef files, as described above. The argument for cmd will be completed in the same way as service.

The function argument may alternatively be a string containing almost any shell code. If the string contains an equal sign, the above will take precedence. The option -e may be used to specify the first argument is to be evaluated as shell code even if it contains an equal sign. The string will be executed using the eval builtin command to generate completions. This provides a way of avoiding having to define a new completion function. For example, to complete files ending in `.h' as arguments to the command foo:

```
compdef '_files -g "*.h"' foo
```

The option -n prevents any completions already defined for the command or context from being overwritten.

The option -d deletes any completion defined for the command or contexts listed.

The names may also contain -p, -P and -N options as described for the #compdef tag. The effect on the argument list is identical, switching between definitions of patterns tried initially, patterns tried finally, and normal commands and contexts.

The parameter $_compskip may be set by any function defined for a pattern context. If it is set to a value containing the substring `patterns' none of the pattern-functions will be called; if it is set to a value containing the substring `all', no other function will be called. Setting $_compskip in this manner is of particular utility when using the -p option, as otherwise the dispatcher will move on to additional functions (likely the default one) after calling the pattern-context one, which can mangle the display of completion possibilities if not handled properly.

The form with -k defines a widget with the same name as the function that will be called for each of the key-sequences; this is like the #compdef -k tag. The function should generate the completions needed and will otherwise behave like the builtin widget whose name is given as the style argument. The widgets usable for this are: complete-word, delete-char-or-list, expand-or-complete, expand-or-complete-prefix, list-choices, menu-complete, menu-expand-or-complete, and reverse-menu-complete, as well as menu-select if the zsh/complist module is loaded.

The option -n prevents the key being bound if it is already to bound to something other than undefined-key.

The form with -K is similar and defines multiple widgets based on the same func? tion, each of which requires the set of three arguments name, style and key-se? quence, where the latter two are as for -k and the first must be a unique widget name beginning with an underscore.

Wherever applicable, the -a option makes the function autoloadable, equivalent to autoload -U function.

The function compdef can be used to associate existing completion functions with new com? mands. For example,

compdef _pids foo

uses the function _pids to complete process IDs for the command foo.

Note also the _gnu_generic function described below, which can be used to complete options for commands that understand the `--help' option.

COMPLETION SYSTEM CONFIGURATION

This section gives a short overview of how the completion system works, and then more de? tail on how users can configure how and when matches are generated.

Overview

When completion is attempted somewhere on the command line the completion system begins building the context. The context represents everything that the shell knows about the meaning of the command line and the significance of the cursor position. This takes ac? count of a number of things including the command word (such as `grep' or `zsh') and op? tions to which the current word may be an argument (such as the `-o' option to zsh which takes a shell option as an argument).

The context starts out very generic ("we are beginning a completion") and becomes more specific as more is learned ("the current word is in a position that is usually a command name" or "the current word might be a variable name" and so on). Therefore the context will vary during the same call to the completion system.

This context information is condensed into a string consisting of multiple fields sepa? rated by colons, referred to simply as `the context' in the remainder of the documenta? tion. Note that a user of the completion system rarely needs to compose a context string, unless for example a new function is being written to perform completion for a new com? mand. What a user may need to do is compose a style pattern, which is matched against a

context when needed to look up context-sensitive options that configure the completion system.

The next few paragraphs explain how a context is composed within the completion function suite. Following that is discussion of how styles are defined. Styles determine such things as how the matches are generated, similarly to shell options but with much more control. They are defined with the zstyle builtin command (see zshmodules(1)).

The context string always consists of a fixed set of fields, separated by colons and with a leading colon before the first. Fields which are not yet known are left empty, but the surrounding colons appear anyway. The fields are always in the order :completion:func? tion:completer:command:argument:tag. These have the following meaning:

? The literal string completion, saying that this style is used by the completion system. This distinguishes the context from those used by, for example, zle wid? gets and ZFTP functions.

? The function, if completion is called from a named widget rather than through the normal completion system. Typically this is blank, but it is set by special wid? gets such as predict-on and the various functions in the Widget directory of the distribution to the name of that function, often in an abbreviated form.

? The completer currently active, the name of the function without the leading under? score and with other underscores converted to hyphens. A `completer' is in overall control of how completion is to be performed; `complete' is the simplest, but other completers exist to perform related tasks such as correction, or to modify the be? haviour of a later completer. See the section `Control Functions' below for more information.

? The command or a special -context-, just at it appears following the #compdef tag or the compdef function. Completion functions for commands that have sub-commands usually modify this field to contain the name of the command followed by a minus sign and the sub-command. For example, the completion function for the cvs command sets this field to cvs-add when completing arguments to the add subcommand.

? The argument; this indicates which command line or option argument we are complet? ing. For command arguments this generally takes the form argument-n, where n is the number of the argument, and for arguments to options the form option-opt-n where n is the number of the argument to option opt. However, this is only the case if the command line is parsed with standard UNIX-style options and arguments,

so many completions do not set this.

? The tag. As described previously, tags are used to discriminate between the types of matches a completion function can generate in a certain context. Any completion function may use any tag name it likes, but a list of the more common ones is given below.

The context is gradually put together as the functions are executed, starting with the main entry point, which adds :completion: and the function element if necessary. The com? pleter then adds the completer element. The contextual completion adds the command and argument options. Finally, the tag is added when the types of completion are known. For example, the context name

:completion::complete:dvips:option-o-1:files

says that normal completion was attempted as the first argument to the option -o of the command dvips:

dvips -o ...

and the completion function will generate filenames.

Usually completion will be tried for all possible tags in an order given by the completion function. However, this can be altered by using the tag-order style. Completion is then restricted to the list of given tags in the given order.

The _complete_help bindable command shows all the contexts and tags available for comple? tion at a particular point. This provides an easy way of finding information for tag-or? der and other styles. It is described in the section `Bindable Commands' below.

When looking up styles the completion system uses full context names, including the tag. Looking up the value of a style therefore consists of two things: the context, which is matched to the most specific (best fitting) style pattern, and the name of the style it? self, which must be matched exactly. The following examples demonstrate that style pat? terns may be loosely defined for styles that apply broadly, or as tightly defined as de? sired for styles that apply in narrower circumstances.

For example, many completion functions can generate matches in a simple and a verbose form and use the verbose style to decide which form should be used. To make all such functions use the verbose form, put

zstyle ':completion:*' verbose yes

in a startup file (probably .zshrc). This gives the verbose style the value yes in every context inside the completion system, unless that context has a more specific definition.

It is best to avoid giving the context as `*' in case the style has some meaning outside the completion system.

Many such general purpose styles can be configured simply by using the compinstall func? tion.

A more specific example of the use of the verbose style is by the completion for the kill builtin. If the style is set, the builtin lists full job texts and process command lines; otherwise it shows the bare job numbers and PIDs. To turn the style off for this use only:

```
zstyle ':completion:*:*:kill:*:*' verbose no
```

For even more control, the style can use one of the tags `jobs' or `processes'. To turn off verbose display only for jobs:

```
zstyle ':completion:*:*:kill:*:jobs' verbose no
```

The -e option to zstyle even allows completion function code to appear as the argument to a style; this requires some understanding of the internals of completion functions (see see zshcompwid(1))). For example,

```
zstyle -e ':completion:*' hosts 'reply=($myhosts)'
```

This forces the value of the hosts style to be read from the variable myhosts each time a host name is needed; this is useful if the value of myhosts can change dynamically. For another useful example, see the example in the description of the file-list style below.

This form can be slow and should be avoided for commonly examined styles such as menu and list-rows-first.

Note that the order in which styles are defined does not matter; the style mechanism uses the most specific possible match for a particular style to determine the set of values.

More precisely, strings are preferred over patterns (for example, `:completion::com? plete:::foo' is more specific than `:completion::complete:::*'), and longer patterns are preferred over shorter patterns.

A good rule of thumb is that any completion style pattern that needs to include more than one wildcard (*) and that does not end in a tag name, should include all six colons (:), possibly surrounding additional wildcards.

Style names like those of tags are arbitrary and depend on the completion function. How? ever, the following two sections list some of the most common tags and styles.

Standard Tags

Some of the following are only used when looking up particular styles and do not refer to

a type of match.

accounts

used to look up the users-hosts style

all-expansions

used by the _expand completer when adding the single string containing all possible expansions

all-files

for the names of all files (as distinct from a particular subset, see the globbed-files tag).

arguments

for arguments to a command

arrays for names of array parameters

association-keys

for keys of associative arrays; used when completing inside a subscript to a param? eter of this type

bookmarks

when completing bookmarks (e.g. for URLs and the zftp function suite)

builtins

for names of builtin commands

characters

for single characters in arguments of commands such as stty. Also used when com? pleting character classes after an opening bracket

colormapids

for X colormap ids

colors for color names

commands

for names of external commands. Also used by complex commands such as cvs when completing names subcommands.

contexts

for contexts in arguments to the zstyle builtin command

corrections

used by the _approximate and _correct completers for possible corrections

cursors

for cursor names used by X programs

default

used in some contexts to provide a way of supplying a default when more specific tags are also valid. Note that this tag is used when only the function field of the context name is set

descriptions

used when looking up the value of the format style to generate descriptions for types of matches

devices

for names of device special files

directories

for names of directories -- local-directories is used instead when completing argu? ments of cd and related builtin commands when the cdpath array is set

directory-stack

for entries in the directory stack

displays

for X display names

domains

for network domains

email-plugin

for email addresses from the `_email-plugin' backend of _email_addresses

expansions

used by the _expand completer for individual words (as opposed to the complete set of expansions) resulting from the expansion of a word on the command line

extensions

for X server extensions

file-descriptors

for numbers of open file descriptors

files the generic file-matching tag used by functions completing filenames

fonts for X font names

fstypes

for file system types (e.g. for the mount command)

functions

names of functions -- normally shell functions, although certain commands may un?

derstand other kinds of function

globbed-files

for filenames when the name has been generated by pattern matching

groups for names of user groups

history-words

for words from the history

hosts for hostnames

indexes

for array indexes

jobs for jobs (as listed by the `jobs' builtin)

interfaces

for network interfaces

keymaps

for names of zsh keymaps

keysyms

for names of X keysyms

libraries

for names of system libraries

limits for system limits

local-directories

for names of directories that are subdirectories of the current working directory

when completing arguments of cd and related builtin commands (compare path-directo?

ries) -- when the cdpath array is unset, directories is used instead

manuals

for names of manual pages

mailboxes

for e-mail folders

maps for map names (e.g. NIS maps)

messages

used to look up the format style for messages

modifiers

for names of X modifiers

modules

for modules (e.g. zsh modules)

my-accounts

used to look up the users-hosts style

named-directories

for named directories (you wouldn't have guessed that, would you?)

names for all kinds of names

newsgroups

for USENET groups

nicknames

for nicknames of NIS maps

options

for command options

original

used by the _approximate, _correct and _expand completers when offering the origi?

nal string as a match

other-accounts

used to look up the users-hosts style

other-files

for the names of any non-directory files. This is used instead of all-files when

the list-dirs-first style is in effect.

packages

for packages (e.g. rpm or installed Debian packages)

parameters

for names of parameters

path-directories

for names of directories found by searching the cdpath array when completing argu?

ments of cd and related builtin commands (compare local-directories)

paths used to look up the values of the expand, ambiguous and special-dirs styles

pods   for perl pods (documentation files)

ports for communication ports

prefixes

for prefixes (like those of a URL)

printers

for print queue names

processes

for process identifiers

processes-names

used to look up the command style when generating the names of processes for kil?

lall

sequences

for sequences (e.g. mh sequences)

sessions

for sessions in the zftp function suite

signals

for signal names

strings

for strings (e.g. the replacement strings for the cd builtin command)

styles for styles used by the zstyle builtin command

suffixes

for filename extensions

tags for tags (e.g. rpm tags)

targets

for makefile targets

time-zones

for time zones (e.g. when setting the TZ parameter)

types for types of whatever (e.g. address types for the xhost command)

urls used to look up the urls and local styles when completing URLs

users for usernames

values for one of a set of values in certain lists

variant

used by _pick_variant to look up the command to run when determining what program

is installed for a particular command name.

visuals

for X visuals

warnings

used to look up the format style for warnings

widgets

for zsh widget names

windows

for IDs of X windows

zsh-options

for shell options

Standard Styles

Note that the values of several of these styles represent boolean values. Any of the strings `true', `on', `yes', and `1' can be used for the value `true' and any of the strings `false', `off', `no', and `0' for the value `false'. The behavior for any other value is undefined except where explicitly mentioned. The default value may be either `true' or `false' if the style is not set.

Some of these styles are tested first for every possible tag corresponding to a type of match, and if no style was found, for the default tag. The most notable styles of this type are menu, list-colors and styles controlling completion listing such as list-packed and last-prompt. When tested for the default tag, only the function field of the context will be set so that a style using the default tag will normally be defined along the lines of:

zstyle ':completion:*:default' menu ...

accept-exact

This is tested for the default tag in addition to the tags valid for the current context. If it is set to `true' and any of the trial matches is the same as the string on the command line, this match will immediately be accepted (even if it would otherwise be considered ambiguous).

When completing pathnames (where the tag used is `paths') this style accepts any number of patterns as the value in addition to the boolean values. Pathnames matching one of these patterns will be accepted immediately even if the command line contains some more partially typed pathname components and these match no file under the directory accepted.

This style is also used by the _expand completer to decide if words beginning with a tilde or parameter expansion should be expanded. For example, if there are pa? rameters foo and foobar, the string `$foo' will only be expanded if accept-exact is

set to `true'; otherwise the completion system will be allowed to complete $foo to $foobar. If the style is set to `continue', _expand will add the expansion as a match and the completion system will also be allowed to continue.

accept-exact-dirs

This is used by filename completion. Unlike accept-exact it is a boolean. By de? fault, filename completion examines all components of a path to see if there are completions of that component, even if the component matches an existing directory. For example, when completion after /usr/bin/, the function examines possible com? pletions to /usr.

When this style is `true', any prefix of a path that matches an existing directory is accepted without any attempt to complete it further. Hence, in the given exam? ple, the path /usr/bin/ is accepted immediately and completion tried in that direc? tory.

This style is also useful when completing after directories that magically appear when referenced, such as ZFS .zfs directories or NetApp .snapshot directories. When the style is set the shell does not check for the existence of the directory within the parent directory.

If you wish to inhibit this behaviour entirely, set the path-completion style (see below) to `false'.

add-space

This style is used by the _expand completer. If it is `true' (the default), a space will be inserted after all words resulting from the expansion, or a slash in the case of directory names. If the value is `file', the completer will only add a space to names of existing files. Either a boolean `true' or the value `file' may be combined with `subst', in which case the completer will not add a space to words generated from the expansion of a substitution of the form `$(...)' or `${...}'.

The _prefix completer uses this style as a simple boolean value to decide if a space should be inserted before the suffix.

ambiguous

This applies when completing non-final components of filename paths, in other words those with a trailing slash. If it is set, the cursor is left after the first am? biguous component, even if menu completion is in use. The style is always tested with the paths tag.

assign-list

When completing after an equals sign that is being treated as an assignment, the completion system normally completes only one filename. In some cases the value may be a list of filenames separated by colons, as with PATH and similar parame? ters. This style can be set to a list of patterns matching the names of such pa? rameters.

The default is to complete lists when the word on the line already contains a colon.

auto-description

If set, this style's value will be used as the description for options that are not described by the completion functions, but that have exactly one argument. The se? quence `%d' in the value will be replaced by the description for this argument. Depending on personal preferences, it may be useful to set this style to something like `specify: %d'. Note that this may not work for some commands.

avoid-completer

This is used by the _all_matches completer to decide if the string consisting of all matches should be added to the list currently being generated. Its value is a list of names of completers. If any of these is the name of the completer that generated the matches in this completion, the string will not be added.

The default value for this style is `_expand _old_list _correct _approximate', i.e. it contains the completers for which a string with all matches will almost never be wanted.

cache-path

This style defines the path where any cache files containing dumped completion data are stored. It defaults to `$ZDOTDIR/.zcompcache', or `$HOME/.zcompcache' if $ZDOTDIR is not defined. The completion cache will not be used unless the use-cache style is set.

cache-policy

This style defines the function that will be used to determine whether a cache needs rebuilding. See the section on the _cache_invalid function below.

call-command

This style is used in the function for commands such as make and ant where calling the command directly to generate matches suffers problems such as being slow or, as

in the case of make can potentially cause actions in the makefile to be executed. If it is set to `true' the command is called to generate matches. The default value of this style is `false'.

command

In many places, completion functions need to call external commands to generate the list of completions. This style can be used to override the command that is called in some such cases. The elements of the value are joined with spaces to form a command line to execute. The value can also start with a hyphen, in which case the usual command will be added to the end; this is most useful for putting `builtin' or `command' in front to make sure the appropriate version of a command is called, for example to avoid calling a shell function with the same name as an external command.

As an example, the completion function for process IDs uses this style with the processes tag to generate the IDs to complete and the list of processes to display (if the verbose style is `true'). The list produced by the command should look like the output of the ps command. The first line is not displayed, but is searched for the string `PID' (or `pid') to find the position of the process IDs in the following lines. If the line does not contain `PID', the first numbers in each of the other lines are taken as the process IDs to complete.

Note that the completion function generally has to call the specified command for each attempt to generate the completion list. Hence care should be taken to spec? ify only commands that take a short time to run, and in particular to avoid any that may never terminate.

command-path

This is a list of directories to search for commands to complete. The default for this style is the value of the special parameter path.

commands

This is used by the function completing sub-commands for the system initialisation scripts (residing in /etc/init.d or somewhere not too far away from that). Its values give the default commands to complete for those commands for which the com? pletion function isn't able to find them out automatically. The default for this style are the two strings `start' and `stop'.

This is used by the _expand_alias function when invoked as a bindable command. If set to `true' and the word on the command line is not the name of an alias, matching alias names will be completed.

complete-options

This is used by the completer for cd, chdir and pushd. For these commands a - is used to introduce a directory stack entry and completion of these is far more common than completing options. Hence unless the value of this style is `true' options will not be completed, even after an initial -. If it is `true', options will be completed after an initial - unless there is a preceding -- on the command line.

completer

The strings given as the value of this style provide the names of the completer functions to use. The available completer functions are described in the section `Control Functions' below.

Each string may be either the name of a completer function or a string of the form `function:name'. In the first case the completer field of the context will contain the name of the completer without the leading underscore and with all other underscores replaced by hyphens. In the second case the function is the name of the completer to call, but the context will contain the user-defined name in the completer field of the context. If the name starts with a hyphen, the string for the context will be build from the name of the completer function as in the first case with the name appended to it. For example:

```
zstyle ':completion:*' completer _complete _complete:-foo
```

Here, completion will call the _complete completer twice, once using `complete' and once using `complete-foo' in the completer field of the context. Normally, using the same completer more than once only makes sense when used with the `functions:name' form, because otherwise the context name will be the same in all calls to the completer; possible exceptions to this rule are the _ignored and _prefix completers.

The default value for this style is `_complete _ignored': only completion will be done, first using the ignored-patterns style and the $fignore array and then without ignoring matches.

condition

This style is used by the _list completer function to decide if insertion of matches should be delayed unconditionally. The default is `true'.

delimiters

This style is used when adding a delimiter for use with history modifiers or glob qualifiers that have delimited arguments. It is an array of preferred delimiters to add. Non-special characters are preferred as the completion system may other? wise become confused. The default list is :, +, /, -, %. The list may be empty to force a delimiter to be typed.

disabled

If this is set to `true', the _expand_alias completer and bindable command will try to expand disabled aliases, too. The default is `false'.

domains

A list of names of network domains for completion. If this is not set, domain names will be taken from the file /etc/resolv.conf.

environ

The environ style is used when completing for `sudo'. It is set to an array of `VAR=value' assignments to be exported into the local environment before the com? pletion for the target command is invoked.

```
zstyle ':completion:*:sudo::' environ \
  PATH="/sbin:/usr/sbin:$PATH" HOME="/root"
```

expand This style is used when completing strings consisting of multiple parts, such as path names.

If one of its values is the string `prefix', the partially typed word from the line will be expanded as far as possible even if trailing parts cannot be completed.

If one of its values is the string `suffix', matching names for components after the first ambiguous one will also be added. This means that the resulting string is the longest unambiguous string possible. However, menu completion can be used to cycle through all matches.

fake This style may be set for any completion context. It specifies additional strings that will always be completed in that context. The form of each string is `value:description'; the colon and description may be omitted, but any literal colons in value must be quoted with a backslash. Any description provided is shown alongside the value in completion listings.

It is important to use a sufficiently restrictive context when specifying fake strings. Note that the styles fake-files and fake-parameters provide additional features when completing files or parameters.

fake-always

This works identically to the fake style except that the ignored-patterns style is not applied to it. This makes it possible to override a set of matches completely by setting the ignored patterns to `*'.

The following shows a way of supplementing any tag with arbitrary data, but having it behave for display purposes like a separate tag. In this example we use the features of the tag-order style to divide the named-directories tag into two when performing completion with the standard completer complete for arguments of cd. The tag named-directories-normal behaves as normal, but the tag named-directo? ries-mine contains a fixed set of directories. This has the effect of adding the match group `extra directories' with the given completions.

```
zstyle ':completion::complete:cd:*' tag-order \
  'named-directories:-mine:extra\ directories
  named-directories:-normal:named\ directories *'
zstyle ':completion::complete:cd:*:named-directories-mine' \
  fake-always mydir1 mydir2
zstyle ':completion::complete:cd:*:named-directories-mine' \
  ignored-patterns '*'
```

fake-files

This style is used when completing files and looked up without a tag. Its values are of the form `dir:names...'. This will add the names (strings separated by spa? ces) as possible matches when completing in the directory dir, even if no such files really exist. The dir may be a pattern; pattern characters or colons in dir should be quoted with a backslash to be treated literally.

This can be useful on systems that support special file systems whose top-level pathnames can not be listed or generated with glob patterns (but see accept-ex? act-dirs for a more general way of dealing with this problem). It can also be used for directories for which one does not have read permission.

The pattern form can be used to add a certain `magic' entry to all directories on a particular file system.

fake-parameters

This is used by the completion function for parameter names. Its values are names of parameters that might not yet be set but should be completed nonetheless. Each name may also be followed by a colon and a string specifying the type of the param? eter (like `scalar', `array' or `integer'). If the type is given, the name will only be completed if parameters of that type are required in the particular con? text. Names for which no type is specified will always be completed.

file-list

This style controls whether files completed using the standard builtin mechanism are to be listed with a long list similar to ls -l. Note that this feature uses the shell module zsh/stat for file information; this loads the builtin stat which will replace any external stat executable. To avoid this the following code can be included in an initialization file:

```
zmodload -i zsh/stat
disable stat
```

The style may either be set to a `true' value (or `all'), or one of the values `in? sert' or `list', indicating that files are to be listed in long format in all cir? cumstances, or when attempting to insert a file name, or when listing file names without attempting to insert one.

More generally, the value may be an array of any of the above values, optionally followed by =num. If num is present it gives the maximum number of matches for which long listing style will be used. For example,

```
zstyle ':completion:*' file-list list=20 insert=10
```

specifies that long format will be used when listing up to 20 files or inserting a file with up to 10 matches (assuming a listing is to be shown at all, for example on an ambiguous completion), else short format will be used.

```
zstyle -e ':completion:*' file-list \
    '(( ${+NUMERIC} )) && reply=(true)'
```

specifies that long format will be used any time a numeric argument is supplied, else short format.

file-patterns

This is used by the standard function for completing filenames, _files. If the style is unset up to three tags are offered, `globbed-files',`directories' and

`all-files', depending on the types of files expected by the caller of _files.

The first two (`globbed-files' and `directories') are normally offered together to make it easier to complete files in sub-directories.

The file-patterns style provides alternatives to the default tags, which are not used. Its value consists of elements of the form `pattern:tag'; each string may contain any number of such specifications separated by spaces.

The pattern is a pattern that is to be used to generate filenames. Any occurrence of the sequence `%p' is replaced by any pattern(s) passed by the function calling _files. Colons in the pattern must be preceded by a backslash to make them distin? guishable from the colon before the tag. If more than one pattern is needed, the patterns can be given inside braces, separated by commas.

The tags of all strings in the value will be offered by _files and used when look? ing up other styles. Any tags in the same word will be offered at the same time and before later words. If no `:tag' is given the `files' tag will be used.

The tag may also be followed by an optional second colon and a description, which will be used for the `%d' in the value of the format style (if that is set) instead of the default description supplied by the completion function. If the description given here contains itself a `%d', that is replaced with the description supplied by the completion function.

For example, to make the rm command first complete only names of object files and then the names of all files if there is no matching object file:

```
zstyle ':completion:*:*:rm:*:*' file-patterns \
  '*.o:object-files' '%p:all-files'
```

To alter the default behaviour of file completion -- offer files matching a pattern and directories on the first attempt, then all files -- to offer only matching files on the first attempt, then directories, and finally all files:

```
zstyle ':completion:*' file-patterns \
  '%p:globbed-files' '*(-/):directories' '*:all-files'
```

This works even where there is no special pattern: _files matches all files using the pattern `*' at the first step and stops when it sees this pattern. Note also it will never try a pattern more than once for a single completion attempt.

During the execution of completion functions, the EXTENDED_GLOB option is in ef? fect, so the characters `#', `~' and `^' have special meanings in the patterns.

file-sort

The standard filename completion function uses this style without a tag to deter? mine in which order the names should be listed; menu completion will cycle through them in the same order. The possible values are: `size' to sort by the size of the file; `links' to sort by the number of links to the file; `modification' (or `time' or `date') to sort by the last modification time; `access' to sort by the last ac? cess time; and `inode' (or `change') to sort by the last inode change time. If the style is set to any other value, or is unset, files will be sorted alphabetically by name. If the value contains the string `reverse', sorting is done in the oppo? site order. If the value contains the string `follow', timestamps are associated with the targets of symbolic links; the default is to use the timestamps of the links themselves.

file-split-chars

A set of characters that will cause all file completions for the given context to be split at the point where any of the characters occurs. A typical use is to set the style to :; then everything up to and including the last : in the string so far is ignored when completing files. As this is quite heavy-handed, it is usually preferable to update completion functions for contexts where this behaviour is use? ful.

filter The ldap plugin of email address completion (see _email_addresses) uses this style to specify the attributes to match against when filtering entries. So for example, if the style is set to `sn', matching is done against surnames. Standard LDAP fil? tering is used so normal completion matching is bypassed. If this style is not set, the LDAP plugin is skipped. You may also need to set the command style to specify how to connect to your LDAP server.

force-list

This forces a list of completions to be shown at any point where listing is done, even in cases where the list would usually be suppressed. For example, normally the list is only shown if there are at least two different matches. By setting this style to `always', the list will always be shown, even if there is only a sin? gle match that will immediately be accepted. The style may also be set to a num? ber. In this case the list will be shown if there are at least that many matches, even if they would all insert the same string.

This style is tested for the default tag as well as for each tag valid for the cur? rent completion. Hence the listing can be forced only for certain types of match.

format If this is set for the descriptions tag, its value is used as a string to display above matches in completion lists. The sequence `%d' in this string will be re? placed with a short description of what these matches are. This string may also contain the output attribute sequences understood by compadd -X (see zshcomp? wid(1)).

The style is tested with each tag valid for the current completion before it is tested for the descriptions tag. Hence different format strings can be defined for different types of match.

Note also that some completer functions define additional `%'-sequences. These are described for the completer functions that make use of them.

Some completion functions display messages that may be customised by setting this style for the messages tag. Here, the `%d' is replaced with a message given by the completion function.

Finally, the format string is looked up with the warnings tag, for use when no matches could be generated at all. In this case the `%d' is replaced with the de? scriptions for the matches that were expected separated by spaces. The sequence `%D' is replaced with the same descriptions separated by newlines.

It is possible to use printf-style field width specifiers with `%d' and similar es? cape sequences. This is handled by the zformat builtin command from the zsh/zutil module, see zshmodules(1).

glob This is used by the _expand completer. If it is set to `true' (the default), glob? bing will be attempted on the words resulting from a previous substitution (see the substitute style) or else the original string from the line.

global If this is set to `true' (the default), the _expand_alias completer and bindable command will try to expand global aliases.

group-name

The completion system can group different types of matches, which appear in sepa? rate lists. This style can be used to give the names of groups for particular tags. For example, in command position the completion system generates names of builtin and external commands, names of aliases, shell functions and parameters and reserved words as possible completions. To have the external commands and shell

functions listed separately:

zstyle ':completion:*:*:-command-:*:commands' \

group-name commands

zstyle ':completion:*:*:-command-:*:functions' \

group-name functions

As a consequence, any match with the same tag will be displayed in the same group.

If the name given is the empty string the name of the tag for the matches will be used as the name of the group. So, to have all different types of matches dis? played separately, one can just set:

zstyle ':completion:*' group-name ''

All matches for which no group name is defined will be put in a group named -de? fault-.

group-order

This style is additional to the group-name style to specify the order for display of the groups defined by that style (compare tag-order, which determines which com? pletions appear at all). The groups named are shown in the given order; any other groups are shown in the order defined by the completion function.

For example, to have names of builtin commands, shell functions and external com? mands appear in that order when completing in command position:

zstyle ':completion:*:*:-command-:*:*' group-order \

builtins functions commands

groups A list of names of UNIX groups. If this is not set, group names are taken from the YP database or the file `/etc/group'.

hidden If this is set to `true', matches for the given context will not be listed, al? though any description for the matches set with the format style will be shown. If it is set to `all', not even the description will be displayed.

Note that the matches will still be completed; they are just not shown in the list.

To avoid having matches considered as possible completions at all, the tag-order style can be modified as described below.

hosts A list of names of hosts that should be completed. If this is not set, hostnames are taken from the file `/etc/hosts'.

hosts-ports

This style is used by commands that need or accept hostnames and network ports.

The strings in the value should be of the form `host:port'. Valid ports are deter? mined by the presence of hostnames; multiple ports for the same host may appear.

ignore-line

This is tested for each tag valid for the current completion. If it is set to `true', none of the words that are already on the line will be considered as possi? ble completions. If it is set to `current', the word the cursor is on will not be considered as a possible completion. The value `current-shown' is similar but only applies if the list of completions is currently shown on the screen. Finally, if the style is set to `other', all words on the line except for the current one will be excluded from the possible completions.

The values `current' and `current-shown' are a bit like the opposite of the ac? cept-exact style: only strings with missing characters will be completed.

Note that you almost certainly don't want to set this to `true' or `other' for a general context such as `:completion:*'. This is because it would disallow comple? tion of, for example, options multiple times even if the command in question ac? cepts the option more than once.

ignore-parents

The style is tested without a tag by the function completing pathnames in order to determine whether to ignore the names of directories already mentioned in the cur? rent word, or the name of the current working directory. The value must include one or both of the following strings:

parent The name of any directory whose path is already contained in the word on the line is ignored. For example, when completing after foo/../, the directory foo will not be considered a valid completion.

pwd The name of the current working directory will not be completed; hence, for example, completion after ../ will not use the name of the current direc? tory.

In addition, the value may include one or both of:

.. Ignore the specified directories only when the word on the line contains the substring `../'.

directory

Ignore the specified directories only when names of directories are com? pleted, not when completing names of files.

Excluded values act in a similar fashion to values of the ignored-patterns style, so they can be restored to consideration by the _ignored completer.

extra-verbose

If set, the completion listing is more verbose at the cost of a probable decrease in completion speed. Completion performance will suffer if this style is set to `true'.

ignored-patterns

A list of patterns; any trial completion matching one of the patterns will be ex? cluded from consideration. The _ignored completer can appear in the list of com? pleters to restore the ignored matches. This is a more configurable version of the shell parameter $fignore.

Note that the EXTENDED_GLOB option is set during the execution of completion func? tions, so the characters `#', `~' and `^' have special meanings in the patterns.

insert This style is used by the _all_matches completer to decide whether to insert the list of all matches unconditionally instead of adding the list as another match.

insert-ids

When completing process IDs, for example as arguments to the kill and wait builtins the name of a command may be converted to the appropriate process ID. A problem arises when the process name typed is not unique. By default (or if this style is set explicitly to `menu') the name will be converted immediately to a set of possi? ble IDs, and menu completion will be started to cycle through them.

If the value of the style is `single', the shell will wait until the user has typed enough to make the command unique before converting the name to an ID; attempts at completion will be unsuccessful until that point. If the value is any other string, menu completion will be started when the string typed by the user is longer than the common prefix to the corresponding IDs.

insert-tab

If this is set to `true', the completion system will insert a TAB character (assum? ing that was used to start completion) instead of performing completion when there is no non-blank character to the left of the cursor. If it is set to `false', com? pletion will be done even there.

The value may also contain the substrings `pending' or `pending=val'. In this case, the typed character will be inserted instead of starting completion when

there is unprocessed input pending. If a val is given, completion will not be done if there are at least that many characters of unprocessed input. This is often useful when pasting characters into a terminal. Note however, that it relies on the $PENDING special parameter from the zsh/zle module being set properly which is not guaranteed on all platforms.

The default value of this style is `true' except for completion within vared builtin command where it is `false'.

insert-unambiguous

This is used by the _match and _approximate completers. These completers are often used with menu completion since the word typed may bear little resemblance to the final completion. However, if this style is `true', the completer will start menu completion only if it could find no unambiguous initial string at least as long as the original string typed by the user.

In the case of the _approximate completer, the completer field in the context will already have been set to one of correct-num or approximate-num, where num is the number of errors that were accepted.

In the case of the _match completer, the style may also be set to the string `pat? tern'. Then the pattern on the line is left unchanged if it does not match unam? biguously.

gain-privileges

If set to true, this style enables the use of commands like sudo or doas to gain extra privileges when retrieving information for completion. This is only done when a command such as sudo appears on the command-line. To force the use of, e.g. sudo or to override any prefix that might be added due to gain-privileges, the command style can be used with a value that begins with a hyphen.

keep-prefix

This style is used by the _expand completer. If it is `true', the completer will try to keep a prefix containing a tilde or parameter expansion. Hence, for exam? ple, the string `~/f*' would be expanded to `~/foo' instead of `/home/user/foo'.

If the style is set to `changed' (the default), the prefix will only be left un? changed if there were other changes between the expanded words and the original word from the command line. Any other value forces the prefix to be expanded un? conditionally.

The behaviour of _expand when this style is `true' is to cause _expand to give up when a single expansion with the restored prefix is the same as the original; hence any remaining completers may be called.

last-prompt

This is a more flexible form of the ALWAYS_LAST_PROMPT option. If it is `true', the completion system will try to return the cursor to the previous command line after displaying a completion list. It is tested for all tags valid for the cur? rent completion, then the default tag. The cursor will be moved back to the previ? ous line if this style is `true' for all types of match. Note that unlike the AL? WAYS_LAST_PROMPT option this is independent of the numeric argument.

known-hosts-files

This style should contain a list of files to search for host names and (if the use-ip style is set) IP addresses in a format compatible with ssh known_hosts files. If it is not set, the files /etc/ssh/ssh_known_hosts and ~/.ssh/known_hosts are used.

list This style is used by the _history_complete_word bindable command. If it is set to `true' it has no effect. If it is set to `false' matches will not be listed. This overrides the setting of the options controlling listing behaviour, in particular AUTO_LIST. The context always starts with `:completion:history-words'.

list-colors

If the zsh/complist module is loaded, this style can be used to set color specifi? cations. This mechanism replaces the use of the ZLS_COLORS and ZLS_COLOURS parame? ters described in the section `The zsh/complist Module' in zshmodules(1), but the syntax is the same.

If this style is set for the default tag, the strings in the value are taken as specifications that are to be used everywhere. If it is set for other tags, the specifications are used only for matches of the type described by the tag. For this to work best, the group-name style must be set to an empty string.

In addition to setting styles for specific tags, it is also possible to use group names specified explicitly by the group-name tag together with the `(group)' syntax allowed by the ZLS_COLORS and ZLS_COLOURS parameters and simply using the default tag.

It is possible to use any color specifications already set up for the GNU version

of the ls command:

```
zstyle ':completion:*:default' list-colors \
    ${(s.:.)LS_COLORS}
```

The default colors are the same as for the GNU ls command and can be obtained by setting the style to an empty string (i.e. '').

list-dirs-first

This is used by file completion. If set, directories to be completed are listed separately from and before completion for other files, regardless of tag ordering. In addition, the tag other-files is used in place of all-files for the remaining files, to indicate that no directories are presented with that tag.

list-grouped

If this style is `true' (the default), the completion system will try to make certain completion listings more compact by grouping matches. For example, options for commands that have the same description (shown when the verbose style is set to `true') will appear as a single entry. However, menu selection can be used to cycle through all the matches.

list-packed

This is tested for each tag valid in the current context as well as the default tag. If it is set to `true', the corresponding matches appear in listings as if the LIST_PACKED option were set. If it is set to `false', they are listed normally.

list-prompt

If this style is set for the default tag, completion lists that don't fit on the screen can be scrolled (see the description of the zsh/complist module in zshmodules(1)). The value, if not the empty string, will be displayed after every screenful and the shell will prompt for a key press; if the style is set to the empty string, a default prompt will be used.

The value may contain the escape sequences: `%l' or `%L', which will be replaced by the number of the last line displayed and the total number of lines; `%m' or `%M', the number of the last match shown and the total number of matches; and `%p' and `%P', `Top' when at the beginning of the list, `Bottom' when at the end and the position shown as a percentage of the total length otherwise. In each case the form with the uppercase letter will be replaced by a string of fixed width, padded to

the right with spaces, while the lowercase form will be replaced by a variable width string. As in other prompt strings, the escape sequences `%S', `%s', `%B', `%b', `%U', `%u' for entering and leaving the display modes standout, bold and un? derline, and `%F', `%f', `%K', `%k' for changing the foreground background colour, are also available, as is the form `%{...%}' for enclosing escape sequences which display with zero (or, with a numeric argument, some other) width.

After deleting this prompt the variable LISTPROMPT should be unset for the removal to take effect.

list-rows-first

This style is tested in the same way as the list-packed style and determines whether matches are to be listed in a rows-first fashion as if the LIST_ROWS_FIRST option were set.

list-suffixes

This style is used by the function that completes filenames. If it is `true', and completion is attempted on a string containing multiple partially typed pathname components, all ambiguous components will be shown. Otherwise, completion stops at the first ambiguous component.

list-separator

The value of this style is used in completion listing to separate the string to complete from a description when possible (e.g. when completing options). It de? faults to `--' (two hyphens).

local This is for use with functions that complete URLs for which the corresponding files are available directly from the file system. Its value should consist of three strings: a hostname, the path to the default web pages for the server, and the di? rectory name used by a user placing web pages within their home area.

For example:

```
zstyle ':completion:*' local toast \
    /var/http/public/toast public_html
```

Completion after `http://toast/stuff/' will look for files in the directory /var/http/public/toast/stuff, while completion after `http://toast/~yousir/' will look for files in the directory ~yousir/public_html.

mail-directory

If set, zsh will assume that mailbox files can be found in the directory specified.

It defaults to `~/Mail'.

match-original

This is used by the _match completer. If it is set to only, _match will try to generate matches without inserting a `*' at the cursor position. If set to any other non-empty value, it will first try to generate matches without inserting the `*' and if that yields no matches, it will try again with the `*' inserted. If it is unset or set to the empty string, matching will only be performed with the `*' inserted.

matcher

This style is tested separately for each tag valid in the current context. Its value is placed before any match specifications given by the matcher-list style so can override them via the use of an x: specification. The value should be in the form described in the section `Completion Matching Control' in zshcompwid(1). For examples of this, see the description of the tag-order style.

For notes comparing the use of this and the matcher-list style, see under the de? scription of the tag-order style.

matcher-list

This style can be set to a list of match specifications that are to be applied ev? erywhere. Match specifications are described in the section `Completion Matching Control' in zshcompwid(1). The completion system will try them one after another for each completer selected. For example, to try first simple completion and, if that generates no matches, case-insensitive completion:

```
zstyle ':completion:*' matcher-list '' 'm:{a-zA-Z}={A-Za-z}'
```

By default each specification replaces the previous one; however, if a specifica? tion is prefixed with +, it is added to the existing list. Hence it is possible to create increasingly general specifications without repetition:

```
zstyle ':completion:*' matcher-list \
    '' '+m:{a-z}={A-Z}' '+m:{A-Z}={a-z}'
```

It is possible to create match specifications valid for particular completers by using the third field of the context. This applies only to completers that over? ride the global matcher-list, which as of this writing includes only _prefix and _ignored. For example, to use the completers _complete and _prefix but allow case-insensitive completion only with _complete:

```
zstyle ':completion:*' completer _complete _prefix
zstyle ':completion:*:complete:*:*:*' matcher-list \
    '' 'm:{a-zA-Z}={A-Za-z}'
```

User-defined names, as explained for the completer style, are available. This makes it possible to try the same completer more than once with different match specifications each time. For example, to try normal completion without a match specification, then normal completion with case-insensitive matching, then correc? tion, and finally partial-word completion:

```
zstyle ':completion:*' completer \
   _complete _correct _complete:foo
zstyle ':completion:*:complete:*:*:*' matcher-list \
   '' 'm:{a-zA-Z}={A-Za-z}'
zstyle ':completion:*:foo:*:*:*' matcher-list \
   'm:{a-zA-Z}={A-Za-z} r:|[-_./]=* r:|=*'
```

If the style is unset in any context no match specification is applied. Note also that some completers such as _correct and _approximate do not use the match speci? fications at all, though these completers will only ever be called once even if the matcher-list contains more than one element.

Where multiple specifications are useful, note that the entire completion is done for each element of matcher-list, which can quickly reduce the shell's performance. As a rough rule of thumb, one to three strings will give acceptable performance. On the other hand, putting multiple space-separated values into the same string does not have an appreciable impact on performance.

If there is no current matcher or it is empty, and the option NO_CASE_GLOB is in effect, the matching for files is performed case-insensitively in any case. How? ever, any matcher must explicitly specify case-insensitive matching if that is re? quired.

For notes comparing the use of this and the matcher style, see under the descrip? tion of the tag-order style.

max-errors

This is used by the _approximate and _correct completer functions to determine the maximum number of errors to allow. The completer will try to generate completions by first allowing one error, then two errors, and so on, until either a match or

matches were found or the maximum number of errors given by this style has been reached.

If the value for this style contains the string `numeric', the completer function will take any numeric argument as the maximum number of errors allowed. For exam? ple, with

zstyle ':completion:*:approximate:::' max-errors 2 numeric

two errors are allowed if no numeric argument is given, but with a numeric argument of six (as in `ESC-6 TAB'), up to six errors are accepted. Hence with a value of `0 numeric', no correcting completion will be attempted unless a numeric argument is given.

If the value contains the string `not-numeric', the completer will not try to gen? erate corrected completions when given a numeric argument, so in this case the num? ber given should be greater than zero. For example, `2 not-numeric' specifies that correcting completion with two errors will usually be performed, but if a numeric argument is given, correcting completion will not be performed.

The default value for this style is `2 numeric'.

max-matches-width

This style is used to determine the trade off between the width of the display used for matches and the width used for their descriptions when the verbose style is in effect. The value gives the number of display columns to reserve for the matches. The default is half the width of the screen.

This has the most impact when several matches have the same description and so will be grouped together. Increasing the style will allow more matches to be grouped together; decreasing it will allow more of the description to be visible.

menu If this is `true' in the context of any of the tags defined for the current comple? tion menu completion will be used. The value for a specific tag will take prece? dence over that for the `default' tag.

If none of the values found in this way is `true' but at least one is set to `auto', the shell behaves as if the AUTO_MENU option is set.

If one of the values is explicitly set to `false', menu completion will be explic? itly turned off, overriding the MENU_COMPLETE option and other settings.

In the form `yes=num', where `yes' may be any of the `true' values (`yes', `true', `on' and `1'), menu completion will be turned on if there are at least num matches.

In the form `yes=long', menu completion will be turned on if the list does not fit on the screen. This does not activate menu completion if the widget normally only lists completions, but menu completion can be activated in that case with the value `yes=long-list' (Typically, the value `select=long-list' described later is more useful as it provides control over scrolling.)

Similarly, with any of the `false' values (as in `no=10'), menu completion will not be used if there are num or more matches.

The value of this widget also controls menu selection, as implemented by the zsh/complist module. The following values may appear either alongside or instead of the values above.

If the value contains the string `select', menu selection will be started uncondi? tionally.

In the form `select=num', menu selection will only be started if there are at least num matches. If the values for more than one tag provide a number, the smallest number is taken.

Menu selection can be turned off explicitly by defining a value containing the string`no-select'.

It is also possible to start menu selection only if the list of matches does not fit on the screen by using the value `select=long'. To start menu selection even if the current widget only performs listing, use the value `select=long-list'.

To turn on menu completion or menu selection when there are a certain number of matches or the list of matches does not fit on the screen, both of `yes=' and `se? lect=' may be given twice, once with a number and once with `long' or `long-list'.

Finally, it is possible to activate two special modes of menu selection. The word `interactive' in the value causes interactive mode to be entered immediately when menu selection is started; see the description of the zsh/complist module in zsh? modules(1) for a description of interactive mode. Including the string `search' does the same for incremental search mode. To select backward incremental search, include the string `search-backward'.

muttrc If set, gives the location of the mutt configuration file. It defaults to `~/.mut? trc'.

numbers

This is used with the jobs tag. If it is `true', the shell will complete job num?

bers instead of the shortest unambiguous prefix of the job command text. If the value is a number, job numbers will only be used if that many words from the job descriptions are required to resolve ambiguities. For example, if the value is `1', strings will only be used if all jobs differ in the first word on their com? mand lines.

old-list

This is used by the _oldlist completer. If it is set to `always', then standard widgets which perform listing will retain the current list of matches, however they were generated; this can be turned off explicitly with the value `never', giving the behaviour without the _oldlist completer. If the style is unset, or any other value, then the existing list of completions is displayed if it is not already; otherwise, the standard completion list is generated; this is the default behaviour of _oldlist. However, if there is an old list and this style contains the name of the completer function that generated the list, then the old list will be used even if it was generated by a widget which does not do listing.

For example, suppose you type ^Xc to use the _correct_word widget, which generates a list of corrections for the word under the cursor. Usually, typing ^D would gen? erate a standard list of completions for the word on the command line, and show that. With _oldlist, it will instead show the list of corrections already gener? ated.

As another example consider the _match completer: with the insert-unambiguous style set to `true' it inserts only a common prefix string, if there is any. However, this may remove parts of the original pattern, so that further completion could produce more matches than on the first attempt. By using the _oldlist completer and setting this style to _match, the list of matches generated on the first at? tempt will be used again.

old-matches

This is used by the _all_matches completer to decide if an old list of matches should be used if one exists. This is selected by one of the `true' values or by the string `only'. If the value is `only', _all_matches will only use an old list and won't have any effect on the list of matches currently being generated.

If this style is set it is generally unwise to call the _all_matches completer un? conditionally. One possible use is for either this style or the completer style to

be defined with the -e option to zstyle to make the style conditional.

old-menu

This is used by the _oldlist completer. It controls how menu completion behaves when a completion has already been inserted and the user types a standard comple? tion key such as TAB. The default behaviour of _oldlist is that menu completion always continues with the existing list of completions. If this style is set to `false', however, a new completion is started if the old list was generated by a different completion command; this is the behaviour without the _oldlist completer. For example, suppose you type ^Xc to generate a list of corrections, and menu com? pletion is started in one of the usual ways. Usually, or with this style set to `false', typing TAB at this point would start trying to complete the line as it now appears. With _oldlist, it instead continues to cycle through the list of correc? tions.

original

This is used by the _approximate and _correct completers to decide if the original string should be added as a possible completion. Normally, this is done only if there are at least two possible corrections, but if this style is set to `true', it is always added. Note that the style will be examined with the completer field in the context name set to correct-num or approximate-num, where num is the number of errors that were accepted.

packageset

This style is used when completing arguments of the Debian `dpkg' program. It con? tains an override for the default package set for a given context. For example,

```
zstyle ':completion:*:complete:dpkg:option--status-1:*' \
        packageset avail
```

causes available packages, rather than only installed packages, to be completed for `dpkg --status'.

path The function that completes color names uses this style with the colors tag. The value should be the pathname of a file containing color names in the format of an X11 rgb.txt file. If the style is not set but this file is found in one of various standard locations it will be used as the default.

path-completion

This is used by filename completion. By default, filename completion examines all

components of a path to see if there are completions of that component. For exam?
ple, /u/b/z can be completed to /usr/bin/zsh. Explicitly setting this style to
`false' inhibits this behaviour for path components up to the / before the cursor;
this overrides the setting of accept-exact-dirs.
Even with the style set to `false', it is still possible to complete multiple paths
by setting the option COMPLETE_IN_WORD and moving the cursor back to the first com?
ponent in the path to be completed. For example, /u/b/z can be completed to
/usr/bin/zsh if the cursor is after the /u.

pine-directory

If set, specifies the directory containing PINE mailbox files. There is no de?
fault, since recursively searching this directory is inconvenient for anyone who
doesn't use PINE.

ports A list of Internet service names (network ports) to complete. If this is not set,
service names are taken from the file `/etc/services'.

prefix-hidden

This is used for certain completions which share a common prefix, for example com?
mand options beginning with dashes. If it is `true', the prefix will not be shown
in the list of matches.
The default value for this style is `false'.

prefix-needed

This style is also relevant for matches with a common prefix. If it is set to
`true' this common prefix must be typed by the user to generate the matches.
The style is applicable to the options, signals, jobs, functions, and parameters
completion tags.
For command options, this means that the initial `-', `+', or `--' must be typed
explicitly before option names will be completed.
For signals, an initial `-' is required before signal names will be completed.
For jobs, an initial `%' is required before job names will be completed.
For function and parameter names, an initial `_' or `.' is required before function
or parameter names starting with those characters will be completed.
The default value for this style is `false' for function and parameter completions,
and `true' otherwise.

preserve-prefix

This style is used when completing path names. Its value should be a pattern matching an initial prefix of the word to complete that should be left unchanged under all circumstances. For example, on some Unices an initial `//' (double slash) has a special meaning; setting this style to the string `//' will preserve it. As another example, setting this style to `?:/' under Cygwin would allow com? pletion after `a:/...' and so on.

range This is used by the _history completer and the _history_complete_word bindable com? mand to decide which words should be completed.

If it is a single number, only the last N words from the history will be completed.

If it is a range of the form `max:slice', the last slice words will be completed; then if that yields no matches, the slice words before those will be tried and so on. This process stops either when at least one match has been found, or max words have been tried.

The default is to complete all words from the history at once.

recursive-files

If this style is set, its value is an array of patterns to be tested against `$PWD/': note the trailing slash, which allows directories in the pattern to be de? limited unambiguously by including slashes on both sides. If an ordinary file com? pletion fails and the word on the command line does not yet have a directory part to its name, the style is retrieved using the same tag as for the completion just attempted, then the elements tested against $PWD/ in turn. If one matches, then the shell reattempts completion by prepending the word on the command line with each directory in the expansion of **/*(/) in turn. Typically the elements of the style will be set to restrict the number of directories beneath the current one to a manageable number, for example `*/.git/*'.

For example,

```
zstyle ':completion:*' recursive-files '*/zsh/*'
```

If the current directory is /home/pws/zsh/Src, then zle_trTAB can be completed to Zle/zle_tricky.c.

regular

This style is used by the _expand_alias completer and bindable command. If set to `true' (the default), regular aliases will be expanded but only in command posi? tion. If it is set to `false', regular aliases will never be expanded. If it is

set to `always', regular aliases will be expanded even if not in command position.

rehash If this is set when completing external commands, the internal list (hash) of com?
mands will be updated for each search by issuing the rehash command. There is a
speed penalty for this which is only likely to be noticeable when directories in
the path have slow file access.

remote-access

If set to `false', certain commands will be prevented from making Internet connec?
tions to retrieve remote information. This includes the completion for the CVS
command.

It is not always possible to know if connections are in fact to a remote site, so
some may be prevented unnecessarily.

remove-all-dups

The _history_complete_word bindable command and the _history completer use this to
decide if all duplicate matches should be removed, rather than just consecutive du?
plicates.

select-prompt

If this is set for the default tag, its value will be displayed during menu selec?
tion (see the menu style above) when the completion list does not fit on the screen
as a whole. The same escapes as for the list-prompt style are understood, except
that the numbers refer to the match or line the mark is on. A default prompt is
used when the value is the empty string.

select-scroll

This style is tested for the default tag and determines how a completion list is
scrolled during a menu selection (see the menu style above) when the completion
list does not fit on the screen as a whole. If the value is `0' (zero), the list
is scrolled by half-screenfuls; if it is a positive integer, the list is scrolled
by the given number of lines; if it is a negative number, the list is scrolled by a
screenful minus the absolute value of the given number of lines. The default is to
scroll by single lines.

separate-sections

This style is used with the manuals tag when completing names of manual pages. If
it is `true', entries for different sections are added separately using tag names
of the form `manual.X', where X is the section number. When the group-name style

is also in effect, pages from different sections will appear separately. This style is also used similarly with the words style when completing words for the dict command. It allows words from different dictionary databases to be added sepa? rately. The default for this style is `false'.

show-ambiguity

If the zsh/complist module is loaded, this style can be used to highlight the first ambiguous character in completion lists. The value is either a color indication such as those supported by the list-colors style or, with a value of `true', a de? fault of underlining is selected. The highlighting is only applied if the comple? tion display strings correspond to the actual matches.

show-completer

Tested whenever a new completer is tried. If it is `true', the completion system outputs a progress message in the listing area showing what completer is being tried. The message will be overwritten by any output when completions are found and is removed after completion is finished.

single-ignored

This is used by the _ignored completer when there is only one match. If its value is `show', the single match will be displayed but not inserted. If the value is `menu', then the single match and the original string are both added as matches and menu completion is started, making it easy to select either of them.

sort This allows the standard ordering of matches to be overridden.

If its value is `true' or `false', sorting is enabled or disabled. Additionally the values associated with the `-o' option to compadd can also be listed: match, nosort, numeric, reverse. If it is not set for the context, the standard behaviour of the calling widget is used.

The style is tested first against the full context including the tag, and if that fails to produce a value against the context without the tag.

In many cases where a calling widget explicitly selects a particular ordering in lieu of the default, a value of `true' is not honoured. An example of where this is not the case is for command history where the default of sorting matches chrono? logically may be overridden by setting the style to `true'.

In the _expand completer, if it is set to `true', the expansions generated will al? ways be sorted. If it is set to `menu', then the expansions are only sorted when

they are offered as single strings but not in the string containing all possible expansions.

special-dirs

Normally, the completion code will not produce the directory names `.' and `..' as possible completions. If this style is set to `true', it will add both `.' and `..' as possible completions; if it is set to `..', only `..' will be added.

The following example sets special-dirs to `..' when the current prefix is empty, is a single `.', or consists only of a path beginning with `../'. Otherwise the value is `false'.

```
zstyle -e ':completion:*' special-dirs \
  '[[ $PREFIX = (../)#(|.|..) ]] && reply=(..)'
```

squeeze-slashes

If set to `true', sequences of slashes in filename paths (for example in `foo//bar') will be treated as a single slash. This is the usual behaviour of UNIX paths. However, by default the file completion function behaves as if there were a `*' between the slashes.

stop If set to `true', the _history_complete_word bindable command will stop once when reaching the beginning or end of the history. Invoking _history_complete_word will then wrap around to the opposite end of the history. If this style is set to `false' (the default), _history_complete_word will loop immediately as in a menu completion.

strip-comments

If set to `true', this style causes non-essential comment text to be removed from completion matches. Currently it is only used when completing e-mail addresses where it removes any display name from the addresses, cutting them down to plain user@host form.

subst-globs-only

This is used by the _expand completer. If it is set to `true', the expansion will only be used if it resulted from globbing; hence, if expansions resulted from the use of the substitute style described below, but these were not further changed by globbing, the expansions will be rejected.

The default for this style is `false'.

substitute

This boolean style controls whether the _expand completer will first try to expand all substitutions in the string (such as `$(...)' and `${...}').

The default is `true'.

suffix This is used by the _expand completer if the word starts with a tilde or contains a parameter expansion. If it is set to `true', the word will only be expanded if it doesn't have a suffix, i.e. if it is something like `~foo' or `$foo' rather than `~foo/' or `$foo/bar', unless that suffix itself contains characters eligible for expansion. The default for this style is `true'.

tag-order

This provides a mechanism for sorting how the tags available in a particular con? text will be used.

The values for the style are sets of space-separated lists of tags. The tags in each value will be tried at the same time; if no match is found, the next value is used. (See the file-patterns style for an exception to this behavior.)

For example:

```
zstyle ':completion:*:complete:-command-:*:*' tag-order \
    'commands functions'
```

specifies that completion in command position first offers external commands and shell functions. Remaining tags will be tried if no completions are found.

In addition to tag names, each string in the value may take one of the following forms:

- If any value consists of only a hyphen, then only the tags specified in the other values are generated. Normally all tags not explicitly selected are tried last if the specified tags fail to generate any matches. This means that a single value consisting only of a single hyphen turns off completion.

! tags...

A string starting with an exclamation mark specifies names of tags that are not to be used. The effect is the same as if all other possible tags for the context had been listed.

tag:label ...

Here, tag is one of the standard tags and label is an arbitrary name. Matches are generated as normal but the name label is used in contexts in? stead of tag. This is not useful in words starting with !.

If the label starts with a hyphen, the tag is prepended to the label to form the name used for lookup. This can be used to make the completion system try a certain tag more than once, supplying different style settings for each attempt; see below for an example.

tag:label:description

As before, but description will replace the `%d' in the value of the format style instead of the default description supplied by the completion func? tion. Spaces in the description must be quoted with a backslash. A `%d' appearing in description is replaced with the description given by the com? pletion function.

In any of the forms above the tag may be a pattern or several patterns in the form `{pat1,pat2...}'. In this case all matching tags will be used except for any given explicitly in the same string.

One use of these features is to try one tag more than once, setting other styles differently on each attempt, but still to use all the other tags without having to repeat them all. For example, to make completion of function names in command po? sition ignore all the completion functions starting with an underscore the first time completion is tried:

```
zstyle ':completion:*:*:-command-:*:*' tag-order \
    'functions:-non-comp *' functions
zstyle ':completion:*:functions-non-comp' \
    ignored-patterns '_*'
```

On the first attempt, all tags will be offered but the functions tag will be re? placed by functions-non-comp. The ignored-patterns style is set for this tag to exclude functions starting with an underscore. If there are no matches, the second value of the tag-order style is used which completes functions using the default tag, this time presumably including all function names.

The matches for one tag can be split into different groups. For example:

```
zstyle ':completion:*' tag-order \
    'options:-long:long\ options
     options:-short:short\ options
     options:-single-letter:single\ letter\ options'
zstyle ':completion:*:options-long' \
```

```
        ignored-patterns '[-+](|-|[^-]*)'
    zstyle ':completion:*:options-short' \
        ignored-patterns '--*' '[-+]?'
    zstyle ':completion:*:options-single-letter' \
        ignored-patterns '???*'
```

With the group-names style set, options beginning with `--', options beginning with a single `-' or `+' but containing multiple characters, and single-letter options will be displayed in separate groups with different descriptions.

Another use of patterns is to try multiple match specifications one after another. The matcher-list style offers something similar, but it is tested very early in the completion system and hence can't be set for single commands nor for more specific contexts. Here is how to try normal completion without any match specification and, if that generates no matches, try again with case-insensitive matching, re? stricting the effect to arguments of the command foo:

```
    zstyle ':completion:*:*:foo:*:*' tag-order '*' '*:-case'
    zstyle ':completion:*-case' matcher 'm:{a-z}={A-Z}'
```

First, all the tags offered when completing after foo are tried using the normal tag name. If that generates no matches, the second value of tag-order is used, which tries all tags again except that this time each has -case appended to its name for lookup of styles. Hence this time the value for the matcher style from the second call to zstyle in the example is used to make completion case-insensi? tive.

It is possible to use the -e option of the zstyle builtin command to specify condi? tions for the use of particular tags. For example:

```
    zstyle -e '*:-command-:*' tag-order '
      if [[ -n $PREFIX$SUFFIX ]]; then
        reply=( )
      else
        reply=( - )
      fi'
```

Completion in command position will be attempted only if the string typed so far is not empty. This is tested using the PREFIX special parameter; see zshcompwid for a description of parameters which are special inside completion widgets. Setting re?

ply to an empty array provides the default behaviour of trying all tags at once; setting it to an array containing only a hyphen disables the use of all tags and hence of all completions.

If no tag-order style has been defined for a context, the strings `(|*-)argument-* (|*-)option-* values' and `options' plus all tags offered by the completion func? tion will be used to provide a sensible default behavior that causes arguments (whether normal command arguments or arguments of options) to be completed before option names for most commands.

urls This is used together with the urls tag by functions completing URLs.

If the value consists of more than one string, or if the only string does not name a file or directory, the strings are used as the URLs to complete.

If the value contains only one string which is the name of a normal file the URLs are taken from that file (where the URLs may be separated by white space or new? lines).

Finally, if the only string in the value names a directory, the directory hierarchy rooted at this directory gives the completions. The top level directory should be the file access method, such as `http', `ftp', `bookmark' and so on. In many cases the next level of directories will be a filename. The directory hierarchy can de? scend as deep as necessary.

For example,

```
zstyle ':completion:*' urls ~/.urls
mkdir -p ~/.urls/ftp/ftp.zsh.org/pub
```

allows completion of all the components of the URL ftp://ftp.zsh.org/pub after suitable commands such as `netscape' or `lynx'. Note, however, that access methods and files are completed separately, so if the hosts style is set hosts can be com? pleted without reference to the urls style.

See the description in the function _urls itself for more information (e.g. `more $^fpath/_urls(N)').

use-cache

If this is set, the completion caching layer is activated for any completions which use it (via the _store_cache, _retrieve_cache, and _cache_invalid functions). The directory containing the cache files can be changed with the cache-path style.

use-compctl

If this style is set to a string not equal to false, 0, no, and off, the completion system may use any completion specifications defined with the compctl builtin command. If the style is unset, this is done only if the zsh/compctl module is loaded. The string may also contain the substring `first' to use completions defined with `compctl -T', and the substring `default' to use the completion defined with `compctl -D'.

Note that this is only intended to smooth the transition from compctl to the new completion system and may disappear in the future.

Note also that the definitions from compctl will only be used if there is no specific completion function for the command in question. For example, if there is a function _foo to complete arguments to the command foo, compctl will never be invoked for foo. However, the compctl version will be tried if foo only uses default completion.

use-ip By default, the function _hosts that completes host names strips IP addresses from entries read from host databases such as NIS and ssh files. If this style is `true', the corresponding IP addresses can be completed as well. This style is not use in any context where the hosts style is set; note also it must be set before the cache of host names is generated (typically the first completion attempt).

users This may be set to a list of usernames to be completed. If it is not set all user names will be completed. Note that if it is set only that list of users will be completed; this is because on some systems querying all users can take a prohibitive amount of time.

users-hosts

The values of this style should be of the form `user@host' or `user:host'. It is used for commands that need pairs of user- and hostnames. These commands will complete usernames from this style (only), and will restrict subsequent hostname completion to hosts paired with that user in one of the values of the style.

It is possible to group values for sets of commands which allow a remote login, such as rlogin and ssh, by using the my-accounts tag. Similarly, values for sets of commands which usually refer to the accounts of other people, such as talk and finger, can be grouped by using the other-accounts tag. More ambivalent commands may use the accounts tag.

users-hosts-ports

Like users-hosts but used for commands like telnet and containing strings of the form `user@host:port'.

verbose

If set, as it is by default, the completion listing is more verbose. In particular many commands show descriptions for options if this style is `true'.

word This is used by the _list completer, which prevents the insertion of completions until a second completion attempt when the line has not changed. The normal way of finding out if the line has changed is to compare its entire contents between the two occasions. If this style is `true', the comparison is instead performed only on the current word. Hence if completion is performed on another word with the same contents, completion will not be delayed.

CONTROL FUNCTIONS

The initialization script compinit redefines all the widgets which perform completion to call the supplied widget function _main_complete. This function acts as a wrapper calling the so-called `completer' functions that generate matches. If _main_complete is called with arguments, these are taken as the names of completer functions to be called in the order given. If no arguments are given, the set of functions to try is taken from the completer style. For example, to use normal completion and correction if that doesn't generate any matches:

zstyle ':completion:*' completer _complete _correct

after calling compinit. The default value for this style is `_complete _ignored', i.e. normally only ordinary completion is tried, first with the effect of the ignored-patterns style and then without it. The _main_complete function uses the return status of the com? pleter functions to decide if other completers should be called. If the return status is zero, no other completers are tried and the _main_complete function returns.

If the first argument to _main_complete is a single hyphen, the arguments will not be taken as names of completers. Instead, the second argument gives a name to use in the completer field of the context and the other arguments give a command name and arguments to call to generate the matches.

The following completer functions are contained in the distribution, although users may write their own. Note that in contexts the leading underscore is stripped, for example basic completion is performed in the context `:completion::complete:...'.

_all_matches

This completer can be used to add a string consisting of all other matches. As it influences later completers it must appear as the first completer in the list. The list of all matches is affected by the avoid-completer and old-matches styles de? scribed above.

It may be useful to use the _generic function described below to bind _all_matches to its own keystroke, for example:

```
zle -C all-matches complete-word _generic
bindkey '^Xa' all-matches
zstyle ':completion:all-matches:*' old-matches only
zstyle ':completion:all-matches::::' completer _all_matches
```

Note that this does not generate completions by itself: first use any of the stan? dard ways of generating a list of completions, then use ^Xa to show all matches.

It is possible instead to add a standard completer to the list and request that the list of all matches should be directly inserted:

```
zstyle ':completion:all-matches::::' completer \
    _all_matches _complete
zstyle ':completion:all-matches:*' insert true
```

In this case the old-matches style should not be set.

_approximate

This is similar to the basic _complete completer but allows the completions to un? dergo corrections. The maximum number of errors can be specified by the max-errors style; see the description of approximate matching in zshexpn(1) for how errors are counted. Normally this completer will only be tried after the normal _complete completer:

```
zstyle ':completion:*' completer _complete _approximate
```

This will give correcting completion if and only if normal completion yields no possible completions. When corrected completions are found, the completer will normally start menu completion allowing you to cycle through these strings.

This completer uses the tags corrections and original when generating the possible corrections and the original string. The format style for the former may contain the additional sequences `%e' and `%o' which will be replaced by the number of er? rors accepted to generate the corrections and the original string, respectively.

The completer progressively increases the number of errors allowed up to the limit

by the max-errors style, hence if a completion is found with one error, no comple?
tions with two errors will be shown, and so on. It modifies the completer name in
the context to indicate the number of errors being tried: on the first try the com?
pleter field contains `approximate-1', on the second try `approximate-2', and so
on.

When _approximate is called from another function, the number of errors to accept
may be passed with the -a option. The argument is in the same format as the
max-errors style, all in one string.

Note that this completer (and the _correct completer mentioned below) can be quite
expensive to call, especially when a large number of errors are allowed. One way
to avoid this is to set up the completer style using the -e option to zstyle so
that some completers are only used when completion is attempted a second time on
the same string, e.g.:

```
zstyle -e ':completion:*' completer '
  if [[ $_last_try != "$HISTNO$BUFFER$CURSOR" ]]; then
    _last_try="$HISTNO$BUFFER$CURSOR"
    reply=(_complete _match _prefix)
  else
    reply=(_ignored _correct _approximate)
  fi'
```

This uses the HISTNO parameter and the BUFFER and CURSOR special parameters that
are available inside zle and completion widgets to find out if the command line
hasn't changed since the last time completion was tried. Only then are the _ig?
nored, _correct and _approximate completers called.

_canonical_paths [-A var] [-N] [-MJV12nfX] tag descr [paths ...]

This completion function completes all paths given to it, and also tries to offer
completions which point to the same file as one of the paths given (relative path
when an absolute path is given, and vice versa; when ..'s are present in the word
to be completed; and some paths got from symlinks).

-A, if specified, takes the paths from the array variable specified. Paths can also
be specified on the command line as shown above. -N, if specified, prevents canon?
icalizing the paths given before using them for completion, in case they are al?
ready so. The options -M, -J, -V, -1, -2, -n, -F, -X are passed to compadd.

See _description for a description of tag and descr.

_cmdambivalent

Completes the remaining positional arguments as an external command. The external command and its arguments are completed as separate arguments (in a manner appro? priate for completing /usr/bin/env) if there are two or more remaining positional arguments on the command line, and as a quoted command string (in the manner of system(...)) otherwise. See also _cmdstring and _precommand.

This function takes no arguments.

_cmdstring

Completes an external command as a single argument, as for system(...).

_complete

This completer generates all possible completions in a context-sensitive manner, i.e. using the settings defined with the compdef function explained above and the current settings of all special parameters. This gives the normal completion be? haviour.

To complete arguments of commands, _complete uses the utility function _normal, which is in turn responsible for finding the particular function; it is described below. Various contexts of the form -context- are handled specifically. These are all mentioned above as possible arguments to the #compdef tag.

Before trying to find a function for a specific context, _complete checks if the parameter `compcontext' is set. Setting `compcontext' allows the usual completion dispatching to be overridden which is useful in places such as a function that uses vared for input. If it is set to an array, the elements are taken to be the possi? ble matches which will be completed using the tag `values' and the description `value'. If it is set to an associative array, the keys are used as the possible completions and the values (if non-empty) are used as descriptions for the matches.

If `compcontext' is set to a string containing colons, it should be of the form `tag:descr:action'. In this case the tag and descr give the tag and description to use and the action indicates what should be completed in one of the forms accepted by the _arguments utility function described below.

Finally, if `compcontext' is set to a string without colons, the value is taken as the name of the context to use and the function defined for that context will be called. For this purpose, there is a special context named -command-line- that

completes whole command lines (commands and their arguments). This is not used by the completion system itself but is nonetheless handled when explicitly called.

_correct

Generate corrections, but not completions, for the current word; this is similar to _approximate but will not allow any number of extra characters at the cursor as that completer does. The effect is similar to spell-checking. It is based on _ap? proximate, but the completer field in the context name is correct.

For example, with:

```
zstyle ':completion:::::' completer \
    _complete _correct _approximate
zstyle ':completion:*:correct:::' max-errors 2 not-numeric
zstyle ':completion:*:approximate:::' max-errors 3 numeric
```

correction will accept up to two errors. If a numeric argument is given, correc? tion will not be performed, but correcting completion will be, and will accept as many errors as given by the numeric argument. Without a numeric argument, first correction and then correcting completion will be tried, with the first one accept? ing two errors and the second one accepting three errors.

When _correct is called as a function, the number of errors to accept may be given following the -a option. The argument is in the same form a values to the accept style, all in one string.

This completer function is intended to be used without the _approximate completer or, as in the example, just before it. Using it after the _approximate completer is useless since _approximate will at least generate the corrected strings gener? ated by the _correct completer -- and probably more.

_expand

This completer function does not really perform completion, but instead checks if the word on the command line is eligible for expansion and, if it is, gives de? tailed control over how this expansion is done. For this to happen, the completion system needs to be invoked with complete-word, not expand-or-complete (the default binding for TAB), as otherwise the string will be expanded by the shell's internal mechanism before the completion system is started. Note also this completer should be called before the _complete completer function.

The tags used when generating expansions are all-expansions for the string contain?

ing all possible expansions, expansions when adding the possible expansions as sin?
gle matches and original when adding the original string from the line. The order
in which these strings are generated, if at all, can be controlled by the group-or?
der and tag-order styles, as usual.

The format string for all-expansions and for expansions may contain the sequence
`%o' which will be replaced by the original string from the line.

The kind of expansion to be tried is controlled by the substitute, glob and
subst-globs-only styles.

It is also possible to call _expand as a function, in which case the different
modes may be selected with options: -s for substitute, -g for glob and -o for
subst-globs-only.

_expand_alias

If the word the cursor is on is an alias, it is expanded and no other completers
are called. The types of aliases which are to be expanded can be controlled with
the styles regular, global and disabled.

This function is also a bindable command, see the section `Bindable Commands' be?
low.

_extensions

If the cursor follows the string `*.', filename extensions are completed. The ex?
tensions are taken from files in current directory or a directory specified at the
beginning of the current word. For exact matches, completion continues to allow
other completers such as _expand to expand the pattern. The standard add-space and
prefix-hidden styles are observed.

_external_pwds

Completes current directories of other zsh processes belonging to the current user.
This is intended to be used via _generic, bound to a custom key combination. Note
that pattern matching is enabled so matching is performed similar to how it works
with the _match completer.

_history

Complete words from the shell's command history. This completer can be controlled
by the remove-all-dups, and sort styles as for the _history_complete_word bindable
command, see the section `Bindable Commands' below and the section `Completion Sys?
tem Configuration' above.

_ignored

The ignored-patterns style can be set to a list of patterns which are compared against possible completions; matching ones are removed. With this completer those matches can be reinstated, as if no ignored-patterns style were set. The completer actually generates its own list of matches; which completers are invoked is deter? mined in the same way as for the _prefix completer. The single-ignored style is also available as described above.

_list This completer allows the insertion of matches to be delayed until completion is attempted a second time without the word on the line being changed. On the first attempt, only the list of matches will be shown. It is affected by the styles con? dition and word, see the section `Completion System Configuration' above.

_match This completer is intended to be used after the _complete completer. It behaves similarly but the string on the command line may be a pattern to match against trial completions. This gives the effect of the GLOB_COMPLETE option.

Normally completion will be performed by taking the pattern from the line, insert? ing a `*' at the cursor position and comparing the resulting pattern with the pos? sible completions generated. This can be modified with the match-original style described above.

The generated matches will be offered in a menu completion unless the insert-unam? biguous style is set to `true'; see the description above for other options for this style.

Note that matcher specifications defined globally or used by the completion func? tions (the styles matcher-list and matcher) will not be used.

_menu This completer was written as simple example function to show how menu completion can be enabled in shell code. However, it has the notable effect of disabling menu selection which can be useful with _generic based widgets. It should be used as the first completer in the list. Note that this is independent of the setting of the MENU_COMPLETE option and does not work with the other menu completion widgets such as reverse-menu-complete, or accept-and-menu-complete.

_oldlist

This completer controls how the standard completion widgets behave when there is an existing list of completions which may have been generated by a special completion (i.e. a separately-bound completion command). It allows the ordinary completion

keys to continue to use the list of completions thus generated, instead of produc?
ing a new list of ordinary contextual completions. It should appear in the list of
completers before any of the widgets which generate matches. It uses two styles:
old-list and old-menu, see the section `Completion System Configuration' above.

_precommand

Complete an external command in word-separated arguments, as for exec and
/usr/bin/env.

_prefix

This completer can be used to try completion with the suffix (everything after the
cursor) ignored. In other words, the suffix will not be considered to be part of
the word to complete. The effect is similar to the expand-or-complete-prefix com?
mand.

The completer style is used to decide which other completers are to be called to
generate matches. If this style is unset, the list of completers set for the cur?
rent context is used -- except, of course, the _prefix completer itself. Further?
more, if this completer appears more than once in the list of completers only those
completers not already tried by the last invocation of _prefix will be called.

For example, consider this global completer style:

```
zstyle ':completion:*' completer \
    _complete _prefix _correct _prefix:foo
```

Here, the _prefix completer tries normal completion but ignoring the suffix. If
that doesn't generate any matches, and neither does the call to the _correct com?
pleter after it, _prefix will be called a second time and, now only trying correc?
tion with the suffix ignored. On the second invocation the completer part of the
context appears as `foo'.

To use _prefix as the last resort and try only normal completion when it is in?
voked:

```
zstyle ':completion:*' completer _complete ... _prefix
zstyle ':completion::prefix:*' completer _complete
```

The add-space style is also respected. If it is set to `true' then _prefix will
insert a space between the matches generated (if any) and the suffix.

Note that this completer is only useful if the COMPLETE_IN_WORD option is set; oth?
erwise, the cursor will be moved to the end of the current word before the comple?

tion code is called and hence there will be no suffix.

_user_expand

This completer behaves similarly to the _expand completer but instead performs expansions defined by users. The styles add-space and sort styles specific to the _expand completer are usable with _user_expand in addition to other styles handled more generally by the completion system. The tag all-expansions is also available. The expansion depends on the array style user-expand being defined for the current context; remember that the context for completers is less specific than that for contextual completion as the full context has not yet been determined. Elements of the array may have one of the following forms:

$hash

hash is the name of an associative array. Note this is not a full parameter expression, merely a $, suitably quoted to prevent immediate expansion, followed by the name of an associative array. If the trial expansion word matches a key in hash, the resulting expansion is the corresponding value.

_func

_func is the name of a shell function whose name must begin with _ but is not otherwise special to the completion system. The function is called with the trial word as an argument. If the word is to be expanded, the function should set the array reply to a list of expansions. Optionally, it can set REPLY to a word that will be used as a description for the set of expansions. The return status of the function is irrelevant.

BINDABLE COMMANDS

In addition to the context-dependent completions provided, which are expected to work in an intuitively obvious way, there are a few widgets implementing special behaviour which can be bound separately to keys. The following is a list of these and their default bindings.

_bash_completions

This function is used by two widgets, _bash_complete-word and _bash_list-choices. It exists to provide compatibility with completion bindings in bash. The last character of the binding determines what is completed: `!', command names; `$', environment variables; `@', host names; `/', file names; `~' user names. In bash, the binding preceded by `\e' gives completion, and preceded by `^X' lists options.

As some of these bindings clash with standard zsh bindings, only `\e~' and `^X~' are bound by default. To add the rest, the following should be added to .zshrc af? ter compinit has been run:

```
for key in '!' '$' '@' '/' '~'; do
  bindkey "\e$key" _bash_complete-word
  bindkey "^X$key" _bash_list-choices
done
```

This includes the bindings for `~' in case they were already bound to something else; the completion code does not override user bindings.

_correct_filename (^XC)

Correct the filename path at the cursor position. Allows up to six errors in the name. Can also be called with an argument to correct a filename path, indepen? dently of zle; the correction is printed on standard output.

_correct_word (^Xc)

Performs correction of the current argument using the usual contextual completions as possible choices. This stores the string `correct-word' in the function field of the context name and then calls the _correct completer.

_expand_alias (^Xa)

This function can be used as a completer and as a bindable command. It expands the word the cursor is on if it is an alias. The types of alias expanded can be con? trolled with the styles regular, global and disabled.

When used as a bindable command there is one additional feature that can be se? lected by setting the complete style to `true'. In this case, if the word is not the name of an alias, _expand_alias tries to complete the word to a full alias name without expanding it. It leaves the cursor directly after the completed word so that invoking _expand_alias once more will expand the now-complete alias name.

_expand_word (^Xe)

Performs expansion on the current word: equivalent to the standard expand-word command, but using the _expand completer. Before calling it, the function field of the context is set to `expand-word'.

_generic

This function is not defined as a widget and not bound by default. However, it can be used to define a widget and will then store the name of the widget in the func?

tion field of the context and call the completion system. This allows custom com? pletion widgets with their own set of style settings to be defined easily. For ex? ample, to define a widget that performs normal completion and starts menu selec? tion:

```
zle -C foo complete-word _generic
bindkey '...' foo
zstyle ':completion:foo:*' menu yes select=1
```

Note in particular that the completer style may be set for the context in order to change the set of functions used to generate possible matches. If _generic is called with arguments, those are passed through to _main_complete as the list of completers in place of those defined by the completer style.

_history_complete_word (\e/)

Complete words from the shell's command history. This uses the list, re? move-all-dups, sort, and stop styles.

_most_recent_file (^Xm)

Complete the name of the most recently modified file matching the pattern on the command line (which may be blank). If given a numeric argument N, complete the Nth most recently modified file. Note the completion, if any, is always unique.

_next_tags (^Xn)

This command alters the set of matches used to that for the next tag, or set of tags, either as given by the tag-order style or as set by default; these matches would otherwise not be available. Successive invocations of the command cycle through all possible sets of tags.

_read_comp (^X^R)

Prompt the user for a string, and use that to perform completion on the current word. There are two possibilities for the string. First, it can be a set of words beginning `_', for example `_files -/', in which case the function with any argu? ments will be called to generate the completions. Unambiguous parts of the func? tion name will be completed automatically (normal completion is not available at this point) until a space is typed.

Second, any other string will be passed as a set of arguments to compadd and should hence be an expression specifying what should be completed.

A very restricted set of editing commands is available when reading the string:

`DEL' and `^H' delete the last character; `^U' deletes the line, and `^C' and `^G' abort the function, while `RET' accepts the completion. Note the string is used verbatim as a command line, so arguments must be quoted in accordance with standard shell rules.

Once a string has been read, the next call to _read_comp will use the existing string instead of reading a new one. To force a new string to be read, call _read_comp with a numeric argument.

_complete_debug (^X?)

This widget performs ordinary completion, but captures in a temporary file a trace of the shell commands executed by the completion system. Each completion attempt gets its own file. A command to view each of these files is pushed onto the editor buffer stack.

_complete_help (^Xh)

This widget displays information about the context names, the tags, and the comple? tion functions used when completing at the current cursor position. If given a nu? meric argument other than 1 (as in `ESC-2 ^Xh'), then the styles used and the con? texts for which they are used will be shown, too.

Note that the information about styles may be incomplete; it depends on the infor? mation available from the completion functions called, which in turn is determined by the user's own styles and other settings.

_complete_help_generic

Unlike other commands listed here, this must be created as a normal ZLE widget rather than a completion widget (i.e. with zle -N). It is used for generating help with a widget bound to the _generic widget that is described above.

If this widget is created using the name of the function, as it is by default, then when executed it will read a key sequence. This is expected to be bound to a call to a completion function that uses the _generic widget. That widget will be exe? cuted, and information provided in the same format that the _complete_help widget displays for contextual completion.

If the widget's name contains debug, for example if it is created as `zle -N _com? plete_debug_generic _complete_help_generic', it will read and execute the keystring for a generic widget as before, but then generate debugging information as done by _complete_debug for contextual completion.

If the widget's name contains noread, it will not read a keystring but instead ar?
range that the next use of a generic widget run in the same shell will have the ef?
fect as described above.

The widget works by setting the shell parameter ZSH_TRACE_GENERIC_WIDGET which is read by _generic. Unsetting the parameter cancels any pending effect of the noread form.

For example, after executing the following:

```
zle -N _complete_debug_generic _complete_help_generic
bindkey '^x:' _complete_debug_generic
```

typing `C-x :' followed by the key sequence for a generic widget will cause trace output for that widget to be saved to a file.

_complete_tag (^Xt)

This widget completes symbol tags created by the etags or ctags programmes (note there is no connection with the completion system's tags) stored in a file TAGS, in the format used by etags, or tags, in the format created by ctags. It will look back up the path hierarchy for the first occurrence of either file; if both exist, the file TAGS is preferred. You can specify the full path to a TAGS or tags file by setting the parameter $TAGSFILE or $tagsfile respectively. The corresponding completion tags used are etags and vtags, after emacs and vi respectively.

UTILITY FUNCTIONS

Descriptions follow for utility functions that may be useful when writing completion func?
tions. If functions are installed in subdirectories, most of these reside in the Base subdirectory. Like the example functions for commands in the distribution, the utility functions generating matches all follow the convention of returning status zero if they generated completions and non-zero if no matching completions could be added.

_absolute_command_paths

This function completes external commands as absolute paths (unlike _command_names -e which completes their basenames). It takes no arguments.

_all_labels [-x] [-12VJ] tag name descr [command arg ...]

This is a convenient interface to the _next_label function below, implementing the loop shown in the _next_label example. The command and its arguments are called to generate the matches. The options stored in the parameter name will automatically be inserted into the args passed to the command. Normally, they are put directly

after the command, but if one of the args is a single hyphen, they are inserted di?
rectly before that. If the hyphen is the last argument, it will be removed from
the argument list before the command is called. This allows _all_labels to be used
in almost all cases where the matches can be generated by a single call to the com?
padd builtin command or by a call to one of the utility functions.

For example:

```
local expl
...
if _requested foo; then
  ...
  _all_labels foo expl '...' compadd ... - $matches
fi
```

Will complete the strings from the matches parameter, using compadd with additional
options which will take precedence over those generated by _all_labels.

_alternative [-O name] [-C name] spec ...

This function is useful in simple cases where multiple tags are available. Essen?
tially it implements a loop like the one described for the _tags function below.
The tags to use and the action to perform if a tag is requested are described using
the specs which are of the form: `tag:descr:action'. The tags are offered using
_tags and if the tag is requested, the action is executed with the given descrip?
tion descr. The actions are those accepted by the _arguments function (described
below), excluding the `->state' and `=...' forms.

For example, the action may be a simple function call:

```
_alternative \
    'users:user:_users' \
    'hosts:host:_hosts'
```

offers usernames and hostnames as possible matches, generated by the _users and
_hosts functions respectively.

Like _arguments, this function uses _all_labels to execute the actions, which will
loop over all sets of tags. Special handling is only required if there is an addi?
tional valid tag, for example inside a function called from _alternative.

The option `-O name' is used in the same way as by the _arguments function. In
other words, the elements of the name array will be passed to compadd when execut?

ing an action.

Like _tags this function supports the -C option to give a different name for the argument context field.

_arguments [-nswWCRS] [-A pat] [-O name] [-M matchspec]

[:] spec ...

_arguments [opt ...] -- [-l] [-i pats] [-s pair]

[helpspec ...]

This function can be used to give a complete specification for completion for a command whose arguments follow standard UNIX option and argument conventions.

Options Overview

Options to _arguments itself must be in separate words, i.e. -s -w, not -sw. The options are followed by specs that describe options and arguments of the analyzed command. To avoid ambiguity, all options to _arguments itself may be separated from the spec forms by a single colon.

The `--' form is used to intuit spec forms from the help output of the command be? ing analyzed, and is described in detail below. The opts for the `--' form are otherwise the same options as the first form. Note that `-s' following `--' has a distinct meaning from `-s' preceding `--', and both may appear.

The option switches -s, -S, -A, -w, and -W affect how _arguments parses the ana? lyzed command line's options. These switches are useful for commands with standard argument parsing.

The options of _arguments have the following meanings:

-n With this option, _arguments sets the parameter NORMARG to the position of the first normal argument in the $words array, i.e. the position after the end of the options. If that argument has not been reached, NORMARG is set to -1. The caller should declare `integer NORMARG' if the -n option is passed; otherwise the parameter is not used.

-s Enable option stacking for single-letter options, whereby multiple sin? gle-letter options may be combined into a single word. For example, the two options `-x' and `-y' may be combined into a single word `-xy'. By default, every word corresponds to a single option name (`-xy' is a single option named `xy').

Options beginning with a single hyphen or plus sign are eligible for stack?

ing; words beginning with two hyphens are not.

Note that -s after -- has a different meaning, which is documented in the segment entitled `Deriving spec forms from the help output'.

-w In combination with -s, allow option stacking even if one or more of the options take arguments. For example, if -x takes an argument, with no -s, `-xy' is considered as a single (unhandled) option; with -s, -xy is an option with the argument `y'; with both -s and -w, -xy is the option -x and the option -y with arguments to -x (and to -y, if it takes arguments) still to come in subsequent words.

-W This option takes -w a stage further: it is possible to complete single-letter options even after an argument that occurs in the same word. However, it depends on the action performed whether options will really be completed at this point. For more control, use a utility function like _guard as part of the action.

-C Modify the curcontext parameter for an action of the form `->state'. This is discussed in detail below.

-R Return status 300 instead of zero when a $state is to be handled, in the `->string' syntax.

-S Do not complete options after a `--' appearing on the line, and ignore the `--'. For example, with -S, in the line

```
foobar -x -- -y
```

the `-x' is considered an option, the `-y' is considered an argument, and the `--' is considered to be neither.

-A pat Do not complete options after the first non-option argument on the line. pat is a pattern matching all strings which are not to be taken as arguments. For example, to make _arguments stop completing options after the first normal argument, but ignoring all strings starting with a hyphen even if they are not described by one of the optspecs, the form is `-A "-*"'.

-O name

Pass the elements of the array name as arguments to functions called to execute actions. This is discussed in detail below.

-M matchspec

Use the match specification matchspec for completing option names and val?

ues. The default matchspec allows partial word completion after `_' and `-', such as completing `-f-b' to `-foo-bar'. The default matchspec is:

r:|[_-]=* r:|=*

specs: overview

Each of the following forms is a spec describing individual sets of options or ar? guments on the command line being analyzed.

n:message:action

n::message:action

This describes the n'th normal argument. The message will be printed above the matches generated and the action indicates what can be completed in this position (see below). If there are two colons before the message the argu? ment is optional. If the message contains only white space, nothing will be printed above the matches unless the action adds an explanation string it? self.

:message:action

::message:action

Similar, but describes the next argument, whatever number that happens to be. If all arguments are specified in this form in the correct order the numbers are unnecessary.

*:message:action

*::message:action

*:::message:action

This describes how arguments (usually non-option arguments, those not begin? ning with - or +) are to be completed when neither of the first two forms was provided. Any number of arguments can be completed in this fashion. With two colons before the message, the words special array and the CURRENT special parameter are modified to refer only to the normal arguments when the action is executed or evaluated. With three colons before the message they are modified to refer only to the normal arguments covered by this de? scription.

optspec

optspec:...

This describes an option. The colon indicates handling for one or more ar?

guments to the option; if it is not present, the option is assumed to take no arguments.

The following forms are available for the initial optspec, whether or not the option has arguments.

*optspec

Here optspec is one of the remaining forms below. This indicates the following optspec may be repeated. Otherwise if the corresponding option is already present on the command line to the left of the cur? sor it will not be offered again.

-optname

+optname

In the simplest form the optspec is just the option name beginning with a minus or a plus sign, such as `-foo'. The first argument for the option (if any) must follow as a separate word directly after the option.

Either of `-+optname' and `+-optname' can be used to specify that -optname and +optname are both valid.

In all the remaining forms, the leading `-' may be replaced by or paired with `+' in this way.

-optname-

The first argument of the option must come directly after the option name in the same word. For example, `-foo-:...' specifies that the completed option and argument will look like `-fooarg'.

-optname+

The first argument may appear immediately after optname in the same word, or may appear as a separate word after the option. For exam? ple, `-foo+:...' specifies that the completed option and argument will look like either `-fooarg' or `-foo arg'.

-optname=

The argument may appear as the next word, or in same word as the op? tion name provided that it is separated from it by an equals sign, for example `-foo=arg' or `-foo arg'.

-optname=-

The argument to the option must appear after an equals sign in the same word, and may not be given in the next argument.

optspec[explanation]

An explanation string may be appended to any of the preceding forms of optspec by enclosing it in brackets, as in `-q[query operation]'.

The verbose style is used to decide whether the explanation strings are displayed with the option in a completion listing.

If no bracketed explanation string is given but the auto-description style is set and only one argument is described for this optspec, the value of the style is displayed, with any appearance of the sequence `%d' in it replaced by the message of the first optarg that follows the optspec; see below.

It is possible for options with a literal `+' or `=' to appear, but that character must be quoted, for example `-\+'.

Each optarg following an optspec must take one of the following forms:

:message:action

::message:action

An argument to the option; message and action are treated as for or? dinary arguments. In the first form, the argument is mandatory, and in the second form it is optional.

This group may be repeated for options which take multiple arguments. In other words, :message1:action1:message2:action2 specifies that the option takes two arguments.

:*pattern:message:action

:*pattern::message:action

:*pattern:::message:action

This describes multiple arguments. Only the last optarg for an op? tion taking multiple arguments may be given in this form. If the pattern is empty (i.e. :*:), all the remaining words on the line are to be completed as described by the action; otherwise, all the words up to and including a word matching the pattern are to be completed using the action.

Multiple colons are treated as for the `*:...' forms for ordinary ar?

guments: when the message is preceded by two colons, the words spe? cial array and the CURRENT special parameter are modified during the execution or evaluation of the action to refer only to the words af? ter the option. When preceded by three colons, they are modified to refer only to the words covered by this description.

Any literal colon in an optname, message, or action must be preceded by a back? slash, `\:'.

Each of the forms above may be preceded by a list in parentheses of option names and argument numbers. If the given option is on the command line, the options and arguments indicated in parentheses will not be offered. For example, `(-two -three 1)-one:...' completes the option `-one'; if this appears on the command line, the options -two and -three and the first ordinary argument will not be completed after it. `(-foo):...' specifies an ordinary argument completion; -foo will not be com? pleted if that argument is already present.

Other items may appear in the list of excluded options to indicate various other items that should not be applied when the current specification is matched: a sin? gle star (*) for the rest arguments (i.e. a specification of the form `*:...'); a colon (:) for all normal (non-option-) arguments; and a hyphen (-) for all options. For example, if `(*)' appears before an option and the option appears on the com? mand line, the list of remaining arguments (those shown in the above table begin? ning with `*:') will not be completed.

To aid in reuse of specifications, it is possible to precede any of the forms above with `!'; then the form will no longer be completed, although if the option or ar? gument appears on the command line they will be skipped as normal. The main use for this is when the arguments are given by an array, and _arguments is called re? peatedly for more specific contexts: on the first call `_arguments $global_options' is used, and on subsequent calls `_arguments !$^global_options'.

specs: actions

In each of the forms above the action determines how completions should be gener? ated. Except for the `->string' form below, the action will be executed by calling the _all_labels function to process all tag labels. No special handling of tags is needed unless a function call introduces a new one.

The functions called to execute actions will be called with the elements of the ar?

ray named by the `-O name' option as arguments. This can be used, for example, to pass the same set of options for the compadd builtin to all actions.

The forms for action are as follows.

(single unquoted space)

This is useful where an argument is required but it is not possible or desirable to generate matches for it. The message will be displayed but no completions listed. Note that even in this case the colon at the end of the message is needed; it may only be omitted when neither a message nor an action is given.

(item1 item2 ...)

One of a list of possible matches, for example:

:foo:(foo bar baz)

((item1\:desc1 ...))

Similar to the above, but with descriptions for each possible match. Note the backslash before the colon. For example,

:foo:((a\:bar b\:baz))

The matches will be listed together with their descriptions if the descrip? tion style is set with the values tag in the context.

->string

In this form, _arguments processes the arguments and options and then returns control to the calling function with parameters set to indicate the state of processing; the calling function then makes its own arrangements for generating completions. For example, functions that implement a state machine can use this type of action.

Where _arguments encounters action in the `->string' format, it will strip all leading and trailing whitespace from string and set the array state to the set of all strings for which an action is to be performed. The elements of the array state_descr are assigned the corresponding message field from each optarg containing such an action.

By default and in common with all other well behaved completion functions, _arguments returns status zero if it was able to add matches and non-zero otherwise. However, if the -R option is given, _arguments will instead return a status of 300 to indicate that $state is to be handled.

In addition to $state and $state_descr, _arguments also sets the global pa? rameters `context', `line' and `opt_args' as described below, and does not reset any changes made to the special parameters such as PREFIX and words. This gives the calling function the choice of resetting these parameters or propagating changes in them.

A function calling _arguments with at least one action containing a `->string' must therefore declare appropriate local parameters:

```
local context state state_descr line
typeset -A opt_args
```

to prevent _arguments from altering the global environment.

{eval-string}

A string in braces is evaluated as shell code to generate matches. If the eval-string itself does not begin with an opening parenthesis or brace it is split into separate words before execution.

= action

If the action starts with `= ' (an equals sign followed by a space), _argu? ments will insert the contents of the argument field of the current context as the new first element in the words special array and increment the value of the CURRENT special parameter. This has the effect of inserting a dummy word onto the completion command line while not changing the point at which completion is taking place.

This is most useful with one of the specifiers that restrict the words on the command line on which the action is to operate (the two- and three-colon forms above). One particular use is when an action itself causes _arguments on a restricted range; it is necessary to use this trick to insert an appro? priate command name into the range for the second call to _arguments to be able to parse the line.

word...

word...

This covers all forms other than those above. If the action starts with a space, the remaining list of words will be invoked unchanged.

Otherwise it will be invoked with some extra strings placed after the first word; these are to be passed down as options to the compadd builtin. They

ensure that the state specified by _arguments, in particular the descrip?
tions of options and arguments, is correctly passed to the completion com?
mand. These additional arguments are taken from the array parameter `expl';
this will be set up before executing the action and hence may be referred to
inside it, typically in an expansion of the form `$expl[@]' which preserves
empty elements of the array.

During the performance of the action the array `line' will be set to the normal ar?
guments from the command line, i.e. the words from the command line after the com?
mand name excluding all options and their arguments. Options are stored in the as?
sociative array `opt_args' with option names as keys and their arguments as the
values. For options that have more than one argument these are given as one
string, separated by colons. All colons and backslashes in the original arguments
are preceded with backslashes.

The parameter `context' is set when returning to the calling function to perform an
action of the form `->string'. It is set to an array of elements corresponding to
the elements of $state. Each element is a suitable name for the argument field of
the context: either a string of the form `option-opt-n' for the n'th argument of
the option -opt, or a string of the form `argument-n' for the n'th argument. For
`rest' arguments, that is those in the list at the end not handled by position, n
is the string `rest'. For example, when completing the argument of the -o option,
the name is `option-o-1', while for the second normal (non-option-) argument it is
`argument-2'.

Furthermore, during the evaluation of the action the context name in the curcontext
parameter is altered to append the same string that is stored in the context param?
eter.

The option -C tells _arguments to modify the curcontext parameter for an action of
the form `->state'. This is the standard parameter used to keep track of the cur?
rent context. Here it (and not the context array) should be made local to the
calling function to avoid passing back the modified value and should be initialised
to the current value at the start of the function:

local curcontext="$curcontext"

This is useful where it is not possible for multiple states to be valid together.

Grouping Options

Options can be grouped to simplify exclusion lists. A group is introduced with `+' followed by a name for the group in the subsequent word. Whole groups can then be referenced in an exclusion list or a group name can be used to disambiguate between two forms of the same option. For example:

```
_arguments \
  '(group2--x)-a' \
 + group1 \
  -m \
  '(group2)-n' \
 + group2 \
  -x -y
```

If the name of a group is specified in the form `(name)' then only one value from that group will ever be completed; more formally, all specifications are mutually exclusive to all other specifications in that group. This is useful for defining options that are aliases for each other. For example:

```
_arguments \
  -a -b \
 + '(operation)' \
  {-c,--compress}'[compress]' \
  {-d,--decompress}'[decompress]' \
  {-l,--list}'[list]'
```

If an option in a group appears on the command line, it is stored in the associa? tive array `opt_args' with 'group-option' as a key. In the example above, a key `operation--c' is used if the option `-c' is present on the command line.

Specifying Multiple Sets of Arguments

It is possible to specify multiple sets of options and arguments with the sets sep? arated by single hyphens. This differs from groups in that sets are considered to be mutually exclusive of each other.

Specifications before the first set and from any group are common to all sets. For example:

```
_arguments \
  -a \
 - set1 \
```

```
  -c \
- set2 \
  -d \
  ':arg:(x2 y2)'
```

This defines two sets. When the command line contains the option `-c', the `-d' option and the argument will not be considered possible completions. When it con? tains `-d' or an argument, the option `-c' will not be considered. However, after `-a' both sets will still be considered valid.

As for groups, the name of a set may appear in exclusion lists, either alone or preceding a normal option or argument specification.

The completion code has to parse the command line separately for each set. This can be slow so sets should only be used when necessary. A useful alternative is often an option specification with rest-arguments (as in `-foo:*:...'); here the option -foo swallows up all remaining arguments as described by the optarg definitions.

Deriving spec forms from the help output

The option `--' allows _arguments to work out the names of long options that sup? port the `--help' option which is standard in many GNU commands. The command word is called with the argument `--help' and the output examined for option names. Clearly, it can be dangerous to pass this to commands which may not support this option as the behaviour of the command is unspecified.

In addition to options, `_arguments --' will try to deduce the types of arguments available for options when the form `--opt=val' is valid. It is also possible to provide hints by examining the help text of the command and adding helpspec of the form `pattern:message:action'; note that other _arguments spec forms are not used. The pattern is matched against the help text for an option, and if it matches the message and action are used as for other argument specifiers. The special case of `*:' means both message and action are empty, which has the effect of causing op? tions having no description in the help output to be ordered in listings ahead of options that have a description.

For example:

```
_arguments -- '*\*:toggle:(yes no)' \
           '*=FILE*:file:_files' \
           '*=DIR*:directory:_files -/' \
```

'*=PATH*:directory:_files -/'

Here, `yes' and `no' will be completed as the argument of options whose description ends in a star; file names will be completed for options that contain the substring `=FILE' in the description; and directories will be completed for options whose de? scription contains `=DIR' or `=PATH'. The last three are in fact the default and so need not be given explicitly, although it is possible to override the use of these patterns. A typical help text which uses this feature is:

-C, --directory=DIR change to directory DIR

so that the above specifications will cause directories to be completed after `--directory', though not after `-C'.

Note also that _arguments tries to find out automatically if the argument for an option is optional. This can be specified explicitly by doubling the colon before the message.

If the pattern ends in `(-)', this will be removed from the pattern and the action will be used only directly after the `=', not in the next word. This is the behav? iour of a normal specification defined with the form `=-'.

By default, the command (with the option `--help') is run after resetting all the locale categories (except for LC_CTYPE) to `C'. If the localized help output is known to work, the option `-l' can be specified after the `_arguments --' so that the command is run in the current locale.

The `_arguments --' can be followed by the option `-i patterns' to give patterns for options which are not to be completed. The patterns can be given as the name of an array parameter or as a literal list in parentheses. For example,

```
_arguments -- -i \
  "(--(en|dis)able-FEATURE*)"
```

will cause completion to ignore the options `--enable-FEATURE' and `--disable-FEA? TURE' (this example is useful with GNU configure).

The `_arguments --' form can also be followed by the option `-s pair' to describe option aliases. The pair consists of a list of alternating patterns and corre? sponding replacements, enclosed in parens and quoted so that it forms a single ar? gument word in the _arguments call.

For example, some configure-script help output describes options only as `--en? able-foo', but the script also accepts the negated form `--disable-foo'. To allow

completion of the second form:

```
_arguments -- -s "((#s)--enable- --disable-)"
```

Miscellaneous notes

Finally, note that _arguments generally expects to be the primary function handling any completion for which it is used. It may have side effects which change the treatment of any matches added by other functions called after it. To combine _ar? guments with other functions, those functions should be called either before _argu? ments, as an action within a spec, or in handlers for `->state' actions.

Here is a more general example of the use of _arguments:

```
_arguments '-l+:left border:' \
        '-format:paper size:(letter A4)' \
        '*-copy:output file:_files::resolution:(300 600)' \
        ':postscript file:_files -g \*.\(ps\|eps\)' \
        '*:page number:'
```

This describes three options: `-l', `-format', and `-copy'. The first takes one argument described as `left border' for which no completion will be offered because of the empty action. Its argument may come directly after the `-l' or it may be given as the next word on the line.

The `-format' option takes one argument in the next word, described as `paper size' for which only the strings `letter' and `A4' will be completed.

The `-copy' option may appear more than once on the command line and takes two ar? guments. The first is mandatory and will be completed as a filename. The second is optional (because of the second colon before the description `resolution') and will be completed from the strings `300' and `600'.

The last two descriptions say what should be completed as arguments. The first de? scribes the first argument as a `postscript file' and makes files ending in `ps' or `eps' be completed. The last description gives all other arguments the description `page numbers' but does not offer completions.

_cache_invalid cache_identifier

This function returns status zero if the completions cache corresponding to the given cache identifier needs rebuilding. It determines this by looking up the cache-policy style for the current context. This should provide a function name which is run with the full path to the relevant cache file as the only argument.

Example:

```
_example_caching_policy () {
  # rebuild if cache is more than a week old
  local -a oldp
  oldp=( "$1"(Nm+7) )
  (( $#oldp ))
}
```

_call_function return name [arg ...]

If a function name exists, it is called with the arguments args. The return argument gives the name of a parameter in which the return status from the function name should be stored; if return is empty or a single hyphen it is ignored.

The return status of _call_function itself is zero if the function name exists and was called and non-zero otherwise.

_call_program [-l] [-p] tag string ...

This function provides a mechanism for the user to override the use of an external command. It looks up the command style with the supplied tag. If the style is set, its value is used as the command to execute. The strings from the call to _call_program, or from the style if set, are concatenated with spaces between them and the resulting string is evaluated. The return status is the return status of the command called.

By default, the command is run in an environment where all the locale categories (except for LC_CTYPE) are reset to `C' by calling the utility function _comp_locale (see below). If the option `-l' is given, the command is run with the current locale.

If the option `-p' is supplied it indicates that the command output is influenced by the permissions it is run with. If the gain-privileges style is set to true, _call_program will make use of commands such as sudo, if present on the command-line, to match the permissions to whatever the final command is likely to run under. When looking up the gain-privileges and command styles, the command component of the zstyle context will end with a slash (`/') followed by the command that would be used to gain privileges.

_combination [-s pattern] tag style spec ... field opts ...

This function is used to complete combinations of values, for example pairs of

hostnames and usernames. The style argument gives the style which defines the pairs; it is looked up in a context with the tag specified.

The style name consists of field names separated by hyphens, for example `users-hosts-ports'. For each field for a value is already known, a spec of the form `field=pattern' is given. For example, if the command line so far specifies a user `pws', the argument `users=pws' should appear.

The next argument with no equals sign is taken as the name of the field for which completions should be generated (presumably not one of the fields for which the value is known).

The matches generated will be taken from the value of the style. These should con? tain the possible values for the combinations in the appropriate order (users, hosts, ports in the example above). The values for the different fields are sepa? rated by colons. This can be altered with the option -s to _combination which specifies a pattern. Typically this is a character class, as for example `-s "[:@]"' in the case of the users-hosts style. Each `field=pattern' specification restricts the completions which apply to elements of the style with appropriately matching fields.

If no style with the given name is defined for the given tag, or if none of the strings in style's value match, but a function name of the required field preceded by an underscore is defined, that function will be called to generate the matches. For example, if there is no `users-hosts-ports' or no matching hostname when a host is required, the function `_hosts' will automatically be called.

If the same name is used for more than one field, in both the `field=pattern' and the argument that gives the name of the field to be completed, the number of the field (starting with one) may be given after the fieldname, separated from it by a colon.

All arguments after the required field name are passed to compadd when generating matches from the style value, or to the functions for the fields if they are called.

_command_names [-e | -]

This function completes words that are valid at command position: names of aliases, builtins, hashed commands, functions, and so on. With the -e flag, only hashed commands are completed. The - flag is ignored.

_comp_locale

This function resets all the locale categories other than LC_CTYPE to `C' so that the output from external commands can be easily analyzed by the completion system. LC_CTYPE retains the current value (taking LC_ALL and LANG into account), ensuring that non-ASCII characters in file names are still handled properly.

This function should normally be run only in a subshell, because the new locale is exported to the environment. Typical usage would be `$(_comp_locale; command ...)'.

_completers [-p]

This function completes names of completers.

-p Include the leading underscore (`_') in the matches.

_describe [-12JVx] [-oO | -t tag] descr name1 [name2] [opt ...]
[-- name1 [name2] [opt ...] ...]

This function associates completions with descriptions. Multiple groups separated by -- can be supplied, potentially with different completion options opts.

The descr is taken as a string to display above the matches if the format style for the descriptions tag is set. This is followed by one or two names of arrays fol? lowed by options to pass to compadd. The array name1 contains the possible comple? tions with their descriptions in the form `completion:description'. Any literal colons in completion must be quoted with a backslash. If a name2 is given, it should have the same number of elements as name1; in this case the corresponding elements are added as possible completions instead of the completion strings from name1. The completion list will retain the descriptions from name1. Finally, a set of completion options can appear.

If the option `-o' appears before the first argument, the matches added will be treated as names of command options (N.B. not shell options), typically following a `-', `--' or `+' on the command line. In this case _describe uses the prefix-hid? den, prefix-needed and verbose styles to find out if the strings should be added as completions and if the descriptions should be shown. Without the `-o' option, only the verbose style is used to decide how descriptions are shown. If `-O' is used instead of `-o', command options are completed as above but _describe will not han? dle the prefix-needed style.

With the -t option a tag can be specified. The default is `values' or, if the -o option is given, `options'.

The options -1, -2, -J, -V, -x are passed to _next_label.

If selected by the list-grouped style, strings with the same description will ap?

pear together in the list.

_describe uses the _all_labels function to generate the matches, so it does not

need to appear inside a loop over tag labels.

_description [-x] [-12VJ] tag name descr [spec ...]

This function is not to be confused with the previous one; it is used as a helper

function for creating options to compadd. It is buried inside many of the higher

level completion functions and so often does not need to be called directly.

The styles listed below are tested in the current context using the given tag. The

resulting options for compadd are put into the array named name (this is tradition?

ally `expl', but this convention is not enforced). The description for the corre?

sponding set of matches is passed to the function in descr.

The styles tested are: format, hidden, matcher, ignore-line, ignored-patterns,

group-name and sort. The format style is first tested for the given tag and then

for the descriptions tag if no value was found, while the remainder are only tested

for the tag given as the first argument. The function also calls _setup which

tests some more styles.

The string returned by the format style (if any) will be modified so that the se?

quence `%d' is replaced by the descr given as the third argument without any lead?

ing or trailing white space. If, after removing the white space, the descr is the

empty string, the format style will not be used and the options put into the name

array will not contain an explanation string to be displayed above the matches.

If _description is called with more than three arguments, the additional specs

should be of the form `char:str'. These supply escape sequence replacements for

the format style: every appearance of `%char' will be replaced by string.

If the -x option is given, the description will be passed to compadd using the -x

option instead of the default -X. This means that the description will be dis?

played even if there are no corresponding matches.

The options placed in the array name take account of the group-name style, so

matches are placed in a separate group where necessary. The group normally has its

elements sorted (by passing the option -J to compadd), but if an option starting

with `-V', `-J', `-1', or `-2' is passed to _description, that option will be in?

cluded in the array. Hence it is possible for the completion group to be unsorted by giving the option `-V', `-1V', or `-2V'.

In most cases, the function will be used like this:

```
local expl
_description files expl file
compadd "$expl[@]" - "$files[@]"
```

Note the use of the parameter expl, the hyphen, and the list of matches. Almost all calls to compadd within the completion system use a similar format; this en? sures that user-specified styles are correctly passed down to the builtins which implement the internals of completion.

_dir_list [-s sep] [-S]

Complete a list of directory names separated by colons (the same format as $PATH).

-s sep Use sep as separator between items. sep defaults to a colon (`:').

-S Add sep instead of slash (`/') as an autoremoveable suffix.

_dispatch context string ...

This sets the current context to context and looks for completion functions to han? dle this context by hunting through the list of command names or special contexts (as described above for compdef) given as strings. The first completion function to be defined for one of the contexts in the list is used to generate matches. Typically, the last string is -default- to cause the function for default comple? tion to be used as a fallback.

The function sets the parameter $service to the string being tried, and sets the context/command field (the fourth) of the $curcontext parameter to the context given as the first argument.

_email_addresses [-c] [-n plugin]

Complete email addresses. Addresses are provided by plugins.

-c Complete bare localhost@domain.tld addresses, without a name part or a com? ment. Without this option, RFC822 `Firstname Lastname <address>' strings are completed.

-n plugin

Complete aliases from plugin.

The following plugins are available by default: _email-ldap (see the filter style), _email-local (completes user@hostname Unix addresses), _email-mail (completes

aliases from ~/.mailrc), _email-mush, _email-mutt, and _email-pine.

Addresses from the _email-foo plugin are added under the tag `email-foo'.

Writing plugins

Plugins are written as separate functions with names starting with `_email-'. They are invoked with the -c option and compadd options. They should either do their own completion or set the $reply array to a list of `alias:address' elements and return 300. New plugins will be picked up and run automatically.

_files The function _files is a wrapper around _path_files. It supports all of the same functionality, with some enhancements -- notably, it respects the list-dirs-first style, and it allows users to override the behaviour of the -g and -/ options with the file-patterns style. _files should therefore be preferred over _path_files in most cases.

This function accepts the full set of options allowed by _path_files, described be? low.

_gnu_generic

This function is a simple wrapper around the _arguments function described above. It can be used to determine automatically the long options understood by commands that produce a list when passed the option `--help'. It is intended to be used as a top-level completion function in its own right. For example, to enable option completion for the commands foo and bar, use

compdef _gnu_generic foo bar

after the call to compinit.

The completion system as supplied is conservative in its use of this function, since it is important to be sure the command understands the option `--help'.

_guard [options] pattern descr

This function displays descr if pattern matches the string to be completed. It is intended to be used in the action for the specifications passed to _arguments and similar functions.

The return status is zero if the message was displayed and the word to complete is not empty, and non-zero otherwise.

The pattern may be preceded by any of the options understood by compadd that are passed down from _description, namely -M, -J, -V, -1, -2, -n, -F and -X. All of these options will be ignored. This fits in conveniently with the argument-passing

conventions of actions for _arguments.

As an example, consider a command taking the options -n and -none, where -n must be followed by a numeric value in the same word. By using:

```
_arguments '-n-: :_guard "[0-9]#" "numeric value"' '-none'
```

_arguments can be made to both display the message `numeric value' and complete options after `-n<TAB>'. If the `-n' is already followed by one or more digits (the pattern passed to _guard) only the message will be displayed; if the `-n' is followed by another character, only options are completed.

_message [-r12] [-VJ group] descr

_message -e [tag] descr

The descr is used in the same way as the third argument to the _description function, except that the resulting string will always be shown whether or not matches were generated. This is useful for displaying a help message in places where no completions can be generated.

The format style is examined with the messages tag to find a message; the usual tag, descriptions, is used only if the style is not set with the former.

If the -r option is given, no style is used; the descr is taken literally as the string to display. This is most useful when the descr comes from a pre-processed argument list which already contains an expanded description. Note that this option does not disable the `%'-sequence parsing done by compadd.

The -12VJ options and the group are passed to compadd and hence determine the group the message string is added to.

The second -e form gives a description for completions with the tag tag to be shown even if there are no matches for that tag. This form is called by _arguments in the event that there is no action for an option specification. The tag can be omitted and if so the tag is taken from the parameter $curtag; this is maintained by the completion system and so is usually correct. Note that if there are no matches at the time this function is called, compstate[insert] is cleared, so additional matches generated later are not inserted on the command line.

_multi_parts [-i] sep array

The argument sep is a separator character. The array may be either the name of an array parameter or a literal array in the form `(foo bar)', a parenthesised list of words separated by whitespace. The possible completions are the strings from the

array. However, each chunk delimited by sep will be completed separately. For ex?
ample, the _tar function uses `_multi_parts / patharray' to complete partial file
paths from the given array of complete file paths.
The -i option causes _multi_parts to insert a unique match even if that requires
multiple separators to be inserted. This is not usually the expected behaviour
with filenames, but certain other types of completion, for example those with a
fixed set of possibilities, may be more suited to this form.
Like other utility functions, this function accepts the `-V', `-J', `-1', `-2',
`-n', `-f', `-X', `-M', `-P', `-S', `-r', `-R', and `-q' options and passes them to
the compadd builtin.

_next_label [-x] [-12VJ] tag name descr [option ...]

This function is used to implement the loop over different tag labels for a partic?
ular tag as described above for the tag-order style. On each call it checks to see
if there are any more tag labels; if there is it returns status zero, otherwise
non-zero. As this function requires a current tag to be set, it must always follow
a call to _tags or _requested.
The -x12VJ options and the first three arguments are passed to the _description
function. Where appropriate the tag will be replaced by a tag label in this call.
Any description given in the tag-order style is preferred to the descr passed to
_next_label.
The options given after the descr are set in the parameter given by name, and hence
are to be passed to compadd or whatever function is called to add the matches.
Here is a typical use of this function for the tag foo. The call to _requested de?
termines if tag foo is required at all; the loop over _next_label handles any la?
bels defined for the tag in the tag-order style.

```
local expl ret=1
...
if _requested foo; then
  ...
  while _next_label foo expl '...'; do
    compadd "$expl[@]" ... && ret=0
  done
  ...
```

fi

return ret

_normal [-P | -p precommand]

This is the standard function called to handle completion outside any special -con? text-. It is called both to complete the command word and also the arguments for a command. In the second case, _normal looks for a special completion for that com? mand, and if there is none it uses the completion for the -default- context.

A second use is to reexamine the command line specified by the $words array and the $CURRENT parameter after those have been modified. For example, the function _pre? command, which completes after precommand specifiers such as nohup, removes the first word from the words array, decrements the CURRENT parameter, then calls `_normal -p $service'. The effect is that `nohup cmd ...' is treated in the same way as `cmd ...'.

-P Reset the list of precommands. This option should be used if completing a command line which allows internal commands (e.g. builtins and functions) regardless of prior precommands (e.g. `zsh -c').

-p precommand

Append precommand to the list of precommands. This option should be used in nearly all cases in which -P is not applicable.

If the command name matches one of the patterns given by one of the options -p or -P to compdef, the corresponding completion function is called and then the parame? ter _compskip is checked. If it is set completion is terminated at that point even if no matches have been found. This is the same effect as in the -first- context.

_options

This can be used to complete the names of shell options. It provides a matcher specification that ignores a leading `no', ignores underscores and allows up? per-case letters to match their lower-case counterparts (for example, `glob', `noglob', `NO_GLOB' are all completed). Any arguments are propagated to the com? padd builtin.

_options_set and _options_unset

These functions complete only set or unset options, with the same matching specifi? cation used in the _options function.

Note that you need to uncomment a few lines in the _main_complete function for

these functions to work properly. The lines in question are used to store the op? tion settings in effect before the completion widget locally sets the options it needs. Hence these functions are not generally used by the completion system.

_parameters

This is used to complete the names of shell parameters.

The option `-g pattern' limits the completion to parameters whose type matches the pattern. The type of a parameter is that shown by `print ${(t)param}', hence judi? cious use of `*' in pattern is probably necessary.

All other arguments are passed to the compadd builtin.

_path_files

This function is used throughout the completion system to complete filenames. It allows completion of partial paths. For example, the string `/u/i/s/sig' may be completed to `/usr/include/sys/signal.h'.

The options accepted by both _path_files and _files are:

-f Complete all filenames. This is the default.

-/ Specifies that only directories should be completed.

-g pattern

Specifies that only files matching the pattern should be completed.

-W paths

Specifies path prefixes that are to be prepended to the string from the com? mand line to generate the filenames but that should not be inserted as com? pletions nor shown in completion listings. Here, paths may be the name of an array parameter, a literal list of paths enclosed in parentheses or an absolute pathname.

-F ignored-files

This behaves as for the corresponding option to the compadd builtin. It gives direct control over which filenames should be ignored. If the option is not present, the ignored-patterns style is used.

Both _path_files and _files also accept the following options which are passed to compadd: `-J', `-V', `-1', `-2', `-n', `-X', `-M', `-P', `-S', `-q', `-r', and `-R'.

Finally, the _path_files function uses the styles expand, ambiguous, special-dirs, list-suffixes and file-sort described above.

_pick_variant [-b builtin-label] [-c command] [-r name]

label=pattern ... label [arg ...]

This function is used to resolve situations where a single command name requires more than one type of handling, either because it has more than one variant or be? cause there is a name clash between two different commands.

The command to run is taken from the first element of the array words unless this is overridden by the option -c. This command is run and its output is compared with a series of patterns. Arguments to be passed to the command can be specified at the end after all the other arguments. The patterns to try in order are given by the arguments label=pattern; if the output of `command arg ...' contains pat? tern, then label is selected as the label for the command variant. If none of the patterns match, the final command label is selected and status 1 is returned.

If the `-b builtin-label' is given, the command is tested to see if it is provided as a shell builtin, possibly autoloaded; if so, the label builtin-label is selected as the label for the variant.

If the `-r name' is given, the label picked is stored in the parameter named name.

The results are also cached in the _cmd_variant associative array indexed by the name of the command run.

_regex_arguments name spec ...

This function generates a completion function name which matches the specifications specs, a set of regular expressions as described below. After running _regex_argu? ments, the function name should be called as a normal completion function. The pattern to be matched is given by the contents of the words array up to the current cursor position joined together with null characters; no quotation is applied.

The arguments are grouped as sets of alternatives separated by `|', which are tried one after the other until one matches. Each alternative consists of a one or more specifications which are tried left to right, with each pattern matched being stripped in turn from the command line being tested, until all of the group suc? ceeds or until one fails; in the latter case, the next alternative is tried. This structure can be repeated to arbitrary depth by using parentheses; matching pro? ceeds from inside to outside.

A special procedure is applied if no test succeeds but the remaining command line string contains no null character (implying the remaining word is the one for which

completions are to be generated). The completion target is restricted to the re? maining word and any actions for the corresponding patterns are executed. In this case, nothing is stripped from the command line string. The order of evaluation of the actions can be determined by the tag-order style; the various formats supported by _alternative can be used in action. The descr is used for setting up the array parameter expl.

Specification arguments take one of following forms, in which metacharacters such as `(', `)', `#' and `|' should be quoted.

/pattern/ [%lookahead%] [-guard] [:tag:descr:action]

This is a single primitive component. The function tests whether the com? bined pattern `(#b)((#B)pattern)lookahead*' matches the command line string. If so, `guard' is evaluated and its return status is examined to determine if the test has succeeded. The pattern string `[]' is guaranteed never to match. The lookahead is not stripped from the command line before the next pattern is examined.

The argument starting with : is used in the same manner as an argument to _alternative.

A component is used as follows: pattern is tested to see if the component already exists on the command line. If it does, any following specifica? tions are examined to find something to complete. If a component is reached but no such pattern exists yet on the command line, the string containing the action is used to generate matches to insert at that point.

/pattern/+ [%lookahead%] [-guard] [:tag:descr:action]

This is similar to `/pattern/ ...' but the left part of the command line string (i.e. the part already matched by previous patterns) is also consid? ered part of the completion target.

/pattern/- [%lookahead%] [-guard] [:tag:descr:action]

This is similar to `/pattern/ ...' but the actions of the current and previ? ously matched patterns are ignored even if the following `pattern' matches the empty string.

(spec)

Parentheses may be used to groups specs; note each parenthesis is a single argument to _regex_arguments.

spec # This allows any number of repetitions of spec.

spec spec

The two specs are to be matched one after the other as described above.

spec | spec

Either of the two specs can be matched.

The function _regex_words can be used as a helper function to generate matches for a set of alternative words possibly with their own arguments as a command line ar? gument.

Examples:

```
_regex_arguments _tst /$'[^\0]#\0'/ \
  /$'[^\0]#\0'/ :'compadd aaa'
```

This generates a function _tst that completes aaa as its only argument. The tag and description for the action have been omitted for brevity (this works but is not recommended in normal use). The first component matches the command word, which is arbitrary; the second matches any argument. As the argument is also arbitrary, any following component would not depend on aaa being present.

```
_regex_arguments _tst /$'[^\0]#\0'/ \
  /$'aaa\0'/ :'compadd aaa'
```

This is a more typical use; it is similar, but any following patterns would only match if aaa was present as the first argument.

```
_regex_arguments _tst /$'[^\0]#\0'/ \( \
  /$'aaa\0'/ :'compadd aaa' \
  /$'bbb\0'/ :'compadd bbb' \) \#
```

In this example, an indefinite number of command arguments may be completed. Odd arguments are completed as aaa and even arguments as bbb. Completion fails unless the set of aaa and bbb arguments before the current one is matched correctly.

```
_regex_arguments _tst /$'[^\0]#\0'/ \
  \( /$'aaa\0'/ :'compadd aaa' \| \
  /$'bbb\0'/ :'compadd bbb' \) \#
```

This is similar, but either aaa or bbb may be completed for any argument. In this case _regex_words could be used to generate a suitable expression for the argu? ments.

This function can be used to generate arguments for the _regex_arguments command which may be inserted at any point where a set of rules is expected. The tag and description give a standard tag and description pertaining to the current context.

Each spec contains two or three arguments separated by a colon: note that there is no leading colon in this case.

Each spec gives one of a set of words that may be completed at this point, together with arguments. It is thus roughly equivalent to the _arguments function when used in normal (non-regex) completion.

The part of the spec before the first colon is the word to be completed. This may contain a *; the entire word, before and after the * is completed, but only the text before the * is required for the context to be matched, so that further argu? ments may be completed after the abbreviated form.

The second part of spec is a description for the word being completed.

The optional third part of the spec describes how words following the one being completed are themselves to be completed. It will be evaluated in order to avoid problems with quoting. This means that typically it contains a reference to an ar? ray containing previously generated regex arguments.

The option -t term specifies a terminator for the word instead of the usual space. This is handled as an auto-removable suffix in the manner of the option -s sep to _values.

The result of the processing by _regex_words is placed in the array reply, which should be made local to the calling function. If the set of words and arguments may be matched repeatedly, a # should be appended to the generated array at that point.

For example:

```
local -a reply
_regex_words mydb-commands 'mydb commands' \
  'add:add an entry to mydb:$mydb_add_cmds' \
  'show:show entries in mydb'
_regex_arguments _mydb "$reply[@]"
_mydb "$@"
```

This shows a completion function for a command mydb which takes two command argu? ments, add and show. show takes no arguments, while the arguments for add have al?

ready been prepared in an array mydb_add_cmds, quite possibly by a previous call to _regex_words.

_requested [-x] [-12VJ] tag [name descr [command [arg ...]]

This function is called to decide whether a tag already registered by a call to _tags (see below) has been requested by the user and hence completion should be performed for it. It returns status zero if the tag is requested and non-zero oth? erwise. The function is typically used as part of a loop over different tags as follows:

```
_tags foo bar baz
while _tags; do
  if _requested foo; then
    ... # perform completion for foo
  fi
  ... # test the tags bar and baz in the same way
  ... # exit loop if matches were generated
done
```

Note that the test for whether matches were generated is not performed until the end of the _tags loop. This is so that the user can set the tag-order style to specify a set of tags to be completed at the same time.

If name and descr are given, _requested calls the _description function with these arguments together with the options passed to _requested.

If command is given, the _all_labels function will be called immediately with the same arguments. In simple cases this makes it possible to perform the test for the tag and the matching in one go. For example:

```
local expl ret=1
_tags foo bar baz
while _tags; do
  _requested foo expl 'description' \
      compadd foobar foobaz && ret=0
  ...
  (( ret )) || break
done
```

If the command is not compadd, it must nevertheless be prepared to handle the same

options.

_retrieve_cache cache_identifier

This function retrieves completion information from the file given by cache_identi? fier, stored in a directory specified by the cache-path style which defaults to ~/.zcompcache. The return status is zero if retrieval was successful. It will only attempt retrieval if the use-cache style is set, so you can call this function without worrying about whether the user wanted to use the caching layer.

See _store_cache below for more details.

_sep_parts

This function is passed alternating arrays and separators as arguments. The arrays specify completions for parts of strings to be separated by the separators. The arrays may be the names of array parameters or a quoted list of words in parenthe? ses. For example, with the array `hosts=(ftp news)' the call `_sep_parts '(foo bar)' @ hosts' will complete the string `f' to `foo' and the string `b@n' to `bar@news'.

This function accepts the compadd options `-V', `-J', `-1', `-2', `-n', `-X', `-M', `-P', `-S', `-r', `-R', and `-q' and passes them on to the compadd builtin used to add the matches.

_sequence [-s sep] [-n max] [-d] function [-] ...

This function is a wrapper to other functions for completing items in a separated list. The same function is used to complete each item in the list. The separator is specified with the -s option. If -s is omitted it will use `,'. Duplicate values are not matched unless -d is specified. If there is a fixed or maximum number of items in the list, this can be specified with the -n option.

Common compadd options are passed on to the function. It is possible to use compadd directly with _sequence, though _values may be more appropriate in this situation.

_setup tag [group]

This function sets up the special parameters used by the completion system appro? priately for the tag given as the first argument. It uses the styles list-colors, list-packed, list-rows-first, last-prompt, accept-exact, menu and force-list.

The optional group supplies the name of the group in which the matches will be placed. If it is not given, the tag is used as the group name.

This function is called automatically from _description and hence is not normally

called explicitly.

_store_cache cache_identifier param ...

This function, together with _retrieve_cache and _cache_invalid, implements a caching layer which can be used in any completion function. Data obtained by costly operations are stored in parameters; this function then dumps the values of those parameters to a file. The data can then be retrieved quickly from that file via _retrieve_cache, even in different instances of the shell.

The cache_identifier specifies the file which the data should be dumped to. The file is stored in a directory specified by the cache-path style which defaults to ~/.zcompcache. The remaining params arguments are the parameters to dump to the file.

The return status is zero if storage was successful. The function will only at? tempt storage if the use-cache style is set, so you can call this function without worrying about whether the user wanted to use the caching layer.

The completion function may avoid calling _retrieve_cache when it already has the completion data available as parameters. However, in that case it should call _cache_invalid to check whether the data in the parameters and in the cache are still valid.

See the _perl_modules completion function for a simple example of the usage of the caching layer.

_tags [[-C name] tag ...]

If called with arguments, these are taken to be the names of tags valid for comple? tions in the current context. These tags are stored internally and sorted by using the tag-order style.

Next, _tags is called repeatedly without arguments from the same completion func? tion. This successively selects the first, second, etc. set of tags requested by the user. The return status is zero if at least one of the tags is requested and non-zero otherwise. To test if a particular tag is to be tried, the _requested function should be called (see above).

If `-C name' is given, name is temporarily stored in the argument field (the fifth) of the context in the curcontext parameter during the call to _tags; the field is restored on exit. This allows _tags to use a more specific context without having to change and reset the curcontext parameter (which has the same effect).

_tilde_files

Like _files, but resolve leading tildes according to the rules of filename expan? sion, so the suggested completions don't start with a `~' even if the filename on the command-line does.

_values [-O name] [-s sep] [-S sep] [-wC] desc spec ...

This is used to complete arbitrary keywords (values) and their arguments, or lists of such combinations.

If the first argument is the option `-O name', it will be used in the same way as by the _arguments function. In other words, the elements of the name array will be passed to compadd when executing an action.

If the first argument (or the first argument after `-O name') is `-s', the next ar? gument is used as the character that separates multiple values. This character is automatically added after each value in an auto-removable fashion (see below); all values completed by `_values -s' appear in the same word on the command line, un? like completion using _arguments. If this option is not present, only a single value will be completed per word.

Normally, _values will only use the current word to determine which values are al? ready present on the command line and hence are not to be completed again. If the -w option is given, other arguments are examined as well.

The first non-option argument, desc, is used as a string to print as a description before listing the values.

All other arguments describe the possible values and their arguments in the same format used for the description of options by the _arguments function (see above). The only differences are that no minus or plus sign is required at the beginning, values can have only one argument, and the forms of action beginning with an equal sign are not supported.

The character separating a value from its argument can be set using the option -S (like -s, followed by the character to use as the separator in the next argument).

By default the equals sign will be used as the separator between values and argu? ments.

Example:

```
_values -s , 'description' \
    '*foo[bar]' \
```

'(two)*one[number]:first count:' \

'two[another number]::second count:(1 2 3)'

This describes three possible values: `foo', `one', and `two'. The first is de? scribed as `bar', takes no argument and may appear more than once. The second is described as `number', may appear more than once, and takes one mandatory argument described as `first count'; no action is specified, so it will not be completed.

The `(two)' at the beginning says that if the value `one' is on the line, the value `two' will no longer be considered a possible completion. Finally, the last value (`two') is described as `another number' and takes an optional argument described as `second count' for which the completions (to appear after an `=') are `1', `2', and `3'. The _values function will complete lists of these values separated by commas.

Like _arguments, this function temporarily adds another context name component to the arguments element (the fifth) of the current context while executing the ac? tion. Here this name is just the name of the value for which the argument is com? pleted.

The style verbose is used to decide if the descriptions for the values (but not those for the arguments) should be printed.

The associative array val_args is used to report values and their arguments; this works similarly to the opt_args associative array used by _arguments. Hence the function calling _values should declare the local parameters state, state_descr, line, context and val_args:

local context state state_descr line

typeset -A val_args

when using an action of the form `->string'. With this function the context param? eter will be set to the name of the value whose argument is to be completed. Note that for _values, the state and state_descr are scalars rather than arrays. Only a single matching state is returned.

Note also that _values normally adds the character used as the separator between values as an auto-removable suffix (similar to a `/' after a directory). However, this is not possible for a `->string' action as the matches for the argument are generated by the calling function. To get the usual behaviour, the calling func? tion can add the separator x as a suffix by passing the options `-qS x' either di?

rectly or indirectly to compadd.

The option -C is treated in the same way as it is by _arguments. In that case the parameter curcontext should be made local instead of context (as described above).

_wanted [-x] [-C name] [-12VJ] tag name descr command [arg ...]

In many contexts, completion can only generate one particular set of matches, usually corresponding to a single tag. However, it is still necessary to decide whether the user requires matches of this type. This function is useful in such a case.

The arguments to _wanted are the same as those to _requested, i.e. arguments to be passed to _description. However, in this case the command is not optional; all the processing of tags, including the loop over both tags and tag labels and the generation of matches, is carried out automatically by _wanted.

Hence to offer only one tag and immediately add the corresponding matches with the given description:

```
local expl
_wanted tag expl 'description' \
    compadd matches...
```

Note that, as for _requested, the command must be able to accept options to be passed down to compadd.

Like _tags this function supports the -C option to give a different name for the argument context field. The -x option has the same meaning as for _description.

_widgets [-g pattern]

This function completes names of zle widgets (see the section `Widgets' in zshzle(1)). The pattern, if present, is matched against values of the $widgets special parameter, documented in the section `The zsh/zleparameter Module' in zshmodules(1).

COMPLETION SYSTEM VARIABLES

There are some standard variables, initialised by the _main_complete function and then used from other functions.

The standard variables are:

_comp_caller_options

The completion system uses setopt to set a number of options. This allows functions to be written without concern for compatibility with every possible combination of

user options. However, sometimes completion needs to know what the user's option preferences are. These are saved in the _comp_caller_options associative array. Op? tion names, spelled in lowercase without underscores, are mapped to one or other of the strings `on' and `off'.

_comp_priv_prefix

Completion functions such as _sudo can set the _comp_priv_prefix array to a command prefix that may then be used by _call_program to match the privi? leges when calling programs to generate matches.

Two more features are offered by the _main_complete function. The arrays comppre? funcs and comppostfuncs may contain names of functions that are to be called imme? diately before or after completion has been tried. A function will only be called once unless it explicitly reinserts itself into the array.

COMPLETION DIRECTORIES

In the source distribution, the files are contained in various subdirectories of the Com? pletion directory. They may have been installed in the same structure, or into one single function directory. The following is a description of the files found in the original di? rectory structure. If you wish to alter an installed file, you will need to copy it to some directory which appears earlier in your fpath than the standard directory where it appears.

Base The core functions and special completion widgets automatically bound to keys. You will certainly need most of these, though will probably not need to alter them. Many of these are documented above.

Zsh Functions for completing arguments of shell builtin commands and utility functions for this. Some of these are also used by functions from the Unix directory.

Unix Functions for completing arguments of external commands and suites of commands. They may need modifying for your system, although in many cases some attempt is made to decide which version of a command is present. For example, completion for the mount command tries to determine the system it is running on, while completion for many other utilities try to decide whether the GNU version of the command is in use, and hence whether the --help option is supported.

X, AIX, BSD, ...

Completion and utility function for commands available only on some systems. These are not arranged hierarchically, so, for example, both the Linux and Debian direc?

tories, as well as the X directory, may be useful on your system.