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

$ man zshcompwid.1

ZSHCOMPWID(1) General Commands Manual ZSHCOMPWID(1)

NAME

zshcompwid - zsh completion widgets

DESCRIPTION

The shell's programmable completion mechanism can be manipulated in two ways; here the low-level features supporting the newer, function-based mechanism are defined. A complete set of shell functions based on these features is described in zshcompsys(1), and users with no inter? est in adding to that system (or, potentially, writing their own -- see dictionary entry for `hubris') should skip the current section. The older system based on the compctl builtin command is described in zsh? compctl(1).

Completion widgets are defined by the -C option to the zle builtin com? mand provided by the zsh/zle module (see zshzle(1)). For example,

zle -C complete expand-or-complete completer

defines a widget named `complete'. The second argument is the name of any of the builtin widgets that handle completions: complete-word, ex? pand-or-complete, expand-or-complete-prefix, menu-complete, menu-ex?

pand-or-complete, reverse-menu-complete, list-choices, or delete-char-or-list. Note that this will still work even if the widget in question has been re-bound.

When this newly defined widget is bound to a key using the bindkey builtin command defined in the zsh/zle module (see zshzle(1)), typing that key will call the shell function `completer'. This function is re? sponsible for generating the possible matches using the builtins de? scribed below. As with other ZLE widgets, the function is called with its standard input closed.

Once the function returns, the completion code takes over control again and treats the matches in the same manner as the specified builtin wid? get, in this case expand-or-complete.

COMPLETION SPECIAL PARAMETERS

The parameters ZLE_REMOVE_SUFFIX_CHARS and ZLE_SPACE_SUFFIX_CHARS are used by the completion mechanism, but are not special. See Parameters Used By The Shell in zshparam(1).

Inside completion widgets, and any functions called from them, some pa? rameters have special meaning; outside these functions they are not special to the shell in any way. These parameters are used to pass in? formation between the completion code and the completion widget. Some of the builtin commands and the condition codes use or change the cur? rent values of these parameters. Any existing values will be hidden during execution of completion widgets; except for compstate, the pa? rameters are reset on each function exit (including nested function calls from within the completion widget) to the values they had when the function was entered.

CURRENT

This is the number of the current word, i.e. the word the cursor is currently on in the words array. Note that this value is only correct if the ksharrays option is not set.

IPREFIX

Initially this will be set to the empty string. This parameter functions like PREFIX; it contains a string which precedes the

one in PREFIX and is not considered part of the list of matches.

Typically, a string is transferred from the beginning of PREFIX to the end of IPREFIX, for example:

```
IPREFIX=${PREFIX%%\=*}=
PREFIX=${PREFIX#*=}
```

causes the part of the prefix up to and including the first equal sign not to be treated as part of a matched string. This can be done automatically by the compset builtin, see below.

ISUFFIX

As IPREFIX, but for a suffix that should not be considered part of the matches; note that the ISUFFIX string follows the SUFFIX string.

PREFIX Initially this will be set to the part of the current word from the beginning of the word up to the position of the cursor; it may be altered to give a common prefix for all matches.

QIPREFIX

This parameter is read-only and contains the quoted string up to the word being completed. E.g. when completing `"foo', this parameter contains the double quote. If the -q option of compset is used (see below), and the original string was `"foo bar' with the cursor on the `bar', this parameter contains `"foo '.

QISUFFIX

Like QIPREFIX, but containing the suffix.

SUFFIX Initially this will be set to the part of the current word from the cursor position to the end; it may be altered to give a common suffix for all matches. It is most useful when the option COMPLETE_IN_WORD is set, as otherwise the whole word on the command line is treated as a prefix.

compstate

This is an associative array with various keys and values that the completion code uses to exchange information with the completion widget. The keys are:

all_quotes

The -q option of the compset builtin command (see below) allows a quoted string to be broken into separate words; if the cursor is on one of those words, that word will be completed, possibly invoking `compset -q' recursively. With this key it is possible to test the types of quoted strings which are currently broken into parts in this fashion. Its value contains one character for each quot? ing level. The characters are a single quote or a double quote for strings quoted with these characters, a dollars sign for strings quoted with $'...' and a backslash for strings not starting with a quote character. The first character in the value always corresponds to the inner? most quoting level.

context

This will be set by the completion code to the overall context in which completion is attempted. Possible values are:

array_value

when completing inside the value of an array pa? rameter assignment; in this case the words array contains the words inside the parentheses.

brace_parameter

when completing the name of a parameter in a pa? rameter expansion beginning with ${. This context will also be set when completing parameter flags following ${(; the full command line argument is presented and the handler must test the value to be completed to ascertain that this is the case.

assign_parameter

when completing the name of a parameter in a pa? rameter assignment.

command

when completing for a normal command (either in

command position or for an argument of the com?
mand).

condition
when completing inside a `[[...]]' conditional ex?
pression; in this case the words array contains
only the words inside the conditional expression.

math when completing in a mathematical environment such
as a `((...))' construct.

parameter
when completing the name of a parameter in a pa?
rameter expansion beginning with $ but not ${.

redirect
when completing after a redirection operator.

subscript
when completing inside a parameter subscript.

value when completing the value of a parameter assign?
ment.

exact Controls the behaviour when the REC_EXACT option is set.
It will be set to accept if an exact match would be ac?
cepted, and will be unset otherwise.
If it was set when at least one match equal to the string
on the line was generated, the match is accepted.

exact_string
The string of an exact match if one was found, otherwise
unset.

ignored
The number of words that were ignored because they
matched one of the patterns given with the -F option to
the compadd builtin command.

insert This controls the manner in which a match is inserted
into the command line. On entry to the widget function,
if it is unset the command line is not to be changed; if
set to unambiguous, any prefix common to all matches is

to be inserted; if set to automenu-unambiguous, the com?
mon prefix is to be inserted and the next invocation of
the completion code may start menu completion (due to the
AUTO_MENU option being set); if set to menu or automenu
menu completion will be started for the matches currently
generated (in the latter case this will happen because
the AUTO_MENU is set). The value may also contain the
string `tab' when the completion code would normally not
really do completion, but only insert the TAB character.
On exit it may be set to any of the values above (where
setting it to the empty string is the same as unsetting
it), or to a number, in which case the match whose number
is given will be inserted into the command line. Nega?
tive numbers count backward from the last match (with
`-1' selecting the last match) and out-of-range values
are wrapped around, so that a value of zero selects the
last match and a value one more than the maximum selects
the first. Unless the value of this key ends in a space,
the match is inserted as in a menu completion, i.e. with?
out automatically appending a space.
Both menu and automenu may also specify the number of the
match to insert, given after a colon. For example,
`menu:2' says to start menu completion, beginning with
the second match.
Note that a value containing the substring `tab' makes
the matches generated be ignored and only the TAB be in?
serted.
Finally, it may also be set to all, which makes all
matches generated be inserted into the line.

insert_positions

When the completion system inserts an unambiguous string
into the line, there may be multiple places where charac?
ters are missing or where the character inserted differs

from at least one match. The value of this key contains a colon separated list of all these positions, as indexes into the command line.

last_prompt

If this is set to a non-empty string for every match added, the completion code will move the cursor back to the previous prompt after the list of completions has been displayed. Initially this is set or unset according to the ALWAYS_LAST_PROMPT option.

list This controls whether or how the list of matches will be displayed. If it is unset or empty they will never be listed; if its value begins with list, they will always be listed; if it begins with autolist or ambiguous, they will be listed when the AUTO_LIST or LIST_AMBIGUOUS op? tions respectively would normally cause them to be.

If the substring force appears in the value, this makes the list be shown even if there is only one match. Nor? mally, the list would be shown only if there are at least two matches.

The value contains the substring packed if the LIST_PACKED option is set. If this substring is given for all matches added to a group, this group will show the LIST_PACKED behavior. The same is done for the LIST_ROWS_FIRST option with the substring rows.

Finally, if the value contains the string explanations, only the explanation strings, if any, will be listed and if it contains messages, only the messages (added with the -x option of compadd) will be listed. If it contains both explanations and messages both kinds of explanation strings will be listed. It will be set appropriately on entry to a completion widget and may be changed there.

list_lines

This gives the number of lines that are needed to display

the full list of completions. Note that to calculate the total number of lines to display you need to add the number of lines needed for the command line to this value, this is available as the value of the BUFFERLINES special parameter.

list_max

Initially this is set to the value of the LISTMAX parameter. It may be set to any other value; when the widget exits this value will be used in the same way as the value of LISTMAX.

nmatches

The number of matches generated and accepted by the completion code so far.

old_insert

On entry to the widget this will be set to the number of the match of an old list of completions that is currently inserted into the command line. If no match has been inserted, this is unset.

As with old_list, the value of this key will only be used if it is the string keep. If it was set to this value by the widget and there was an old match inserted into the command line, this match will be kept and if the value of the insert key specifies that another match should be inserted, this will be inserted after the old one.

old_list

This is set to yes if there is still a valid list of completions from a previous completion at the time the widget is invoked. This will usually be the case if and only if the previous editing operation was a completion widget or one of the builtin completion functions. If there is a valid list and it is also currently shown on the screen, the value of this key is shown.

After the widget has exited the value of this key is only

used if it was set to keep. In this case the completion code will continue to use this old list. If the widget generated new matches, they will not be used.

parameter

The name of the parameter when completing in a subscript or in the value of a parameter assignment.

pattern_insert

Normally this is set to menu, which specifies that menu completion will be used whenever a set of matches was generated using pattern matching. If it is set to any other non-empty string by the user and menu completion is not selected by other option settings, the code will in? stead insert any common prefix for the generated matches as with normal completion.

pattern_match

Locally controls the behaviour given by the GLOB_COMPLETE option. Initially it is set to `*' if and only if the option is set. The completion widget may set it to this value, to an empty string (which has the same effect as unsetting it), or to any other non-empty string. If it is non-empty, unquoted metacharacters on the command line will be treated as patterns; if it is `*', then addition? ally a wildcard `*' is assumed at the cursor position; if it is empty or unset, metacharacters will be treated lit? erally.

Note that the matcher specifications given to the compadd builtin command are not used if this is set to a non-empty string.

quote When completing inside quotes, this contains the quota? tion character (i.e. either a single quote, a double quote, or a backtick). Otherwise it is unset.

quoting

When completing inside single quotes, this is set to the

string single; inside double quotes, the string double; inside backticks, the string backtick. Otherwise it is unset.

redirect

The redirection operator when completing in a redirection position, i.e. one of <, >, etc.

restore

This is set to auto before a function is entered, which forces the special parameters mentioned above (words, CURRENT, PREFIX, IPREFIX, SUFFIX, and ISUFFIX) to be re? stored to their previous values when the function exits. If a function unsets it or sets it to any other string, they will not be restored.

to_end Specifies the occasions on which the cursor is moved to the end of a string when a match is inserted. On entry to a widget function, it may be single if this will hap? pen when a single unambiguous match was inserted or match if it will happen any time a match is inserted (for exam? ple, by menu completion; this is likely to be the effect of the ALWAYS_TO_END option). On exit, it may be set to single as above. It may also be set to always, or to the empty string or unset; in those cases the cursor will be moved to the end of the string always or never respectively. Any other string is treated as match.

unambiguous

This key is read-only and will always be set to the com? mon (unambiguous) prefix the completion code has gener? ated for all matches added so far.

unambiguous_cursor

This gives the position the cursor would be placed at if the common prefix in the unambiguous key were inserted, relative to the value of that key. The cursor would be

placed before the character whose index is given by this key.

unambiguous_positions
This contains all positions where characters in the unambiguous string are missing or where the character inserted differs from at least one of the matches. The positions are given as indexes into the string given by the value of the unambiguous key.

vared If completion is called while editing a line using the vared builtin, the value of this key is set to the name of the parameter given as an argument to vared. This key is only set while a vared command is active.

words This array contains the words present on the command line currently being edited.

COMPLETION BUILTIN COMMANDS

compadd [-akqQfenUl12C] [-F array]
[-P prefix] [-S suffix]
[-p hidden-prefix] [-s hidden-suffix]
[-i ignored-prefix] [-I ignored-suffix]
[-W file-prefix] [-d array]
[-J group-name] [-X explanation] [-x message]
[-V group-name] [-o [order]]
[-r remove-chars] [-R remove-func]
[-D array] [-O array] [-A array]
[-E number]
[-M match-spec] [--] [words ...]

This builtin command can be used to add matches directly and control all the information the completion code stores with each possible match. The return status is zero if at least one match was added and non-zero if no matches were added.

The completion code breaks the string to complete into seven fields in the order:

<ipre><apre><hpre><word><hsuf><asuf><isuf>

The first field is an ignored prefix taken from the command line, the contents of the IPREFIX parameter plus the string given with the -i option. With the -U option, only the string from the -i option is used. The field <apre> is an optional pre? fix string given with the -P option. The <hpre> field is a string that is considered part of the match but that should not be shown when listing completions, given with the -p option; for example, functions that do filename generation might specify a common path prefix this way. <word> is the part of the match that should appear in the list of completions, i.e. one of the words given at the end of the compadd command line. The suffixes <hsuf>, <asuf> and <isuf> correspond to the prefixes <hpre>, <apre> and <ipre> and are given by the options -s, -S and -I, respectively.

The supported flags are:

-P prefix

This gives a string to be inserted before the given words. The string given is not considered as part of the match and any shell metacharacters in it will not be quoted when the string is inserted.

-S suffix

Like -P, but gives a string to be inserted after the match.

-p hidden-prefix

This gives a string that should be inserted into the com? mand line before the match but that should not appear in the list of matches. Unless the -U option is given, this string must be matched as part of the string on the com? mand line.

-s hidden-suffix

Like `-p', but gives a string to insert after the match.

-i ignored-prefix

This gives a string to insert into the command line just

before any string given with the `-P' option. Without `-P' the string is inserted before the string given with `-p' or directly before the match.

-I ignored-suffix

Like -i, but gives an ignored suffix.

-a With this flag the words are taken as names of arrays and the possible matches are their values. If only some elements of the arrays are needed, the words may also contain subscripts, as in `foo[2,-1]'.

-k With this flag the words are taken as names of associative arrays and the possible matches are their keys. As for -a, the words may also contain subscripts, as in `foo[(R)*bar*]'.

-d array

This adds per-match display strings. The array should contain one element per word given. The completion code will then display the first element instead of the first word, and so on. The array may be given as the name of an array parameter or directly as a space-separated list of words in parentheses.

If there are fewer display strings than words, the leftover words will be displayed unchanged and if there are more display strings than words, the leftover display strings will be silently ignored.

-l This option only has an effect if used together with the -d option. If it is given, the display strings are listed one per line, not arrayed in columns.

-o [order]

This controls the order in which matches are sorted. order is a comma-separated list comprising the following possible values. These values can be abbreviated to their initial two or three characters. Note that the order forms part of the group name space so matches with

different orderings will not be in the same group.

match If given, the order of the output is determined by the match strings; otherwise it is determined by the display strings (i.e. the strings given by the -d option). This is the default if `-o' is speci? fied but the order argument is omitted.

nosort This specifies that the matches are pre-sorted and their order should be preserved. This value only makes sense alone and cannot be combined with any others.

numeric

If the matches include numbers, sort them numeri? cally rather than lexicographically.

reverse

Arrange the matches backwards by reversing the sort ordering.

-J group-name

Gives the name of the group of matches the words should be stored in.

-V group-name

Like -J but naming an unsorted group. This option is identical to the combination of -J and -o nosort.

-1 If given together with the -V option, makes only consecu? tive duplicates in the group be removed. If combined with the -J option, this has no visible effect. Note that groups with and without this flag are in different name spaces.

-2 If given together with the -J or -V option, makes all du? plicates be kept. Again, groups with and without this flag are in different name spaces.

-X explanation

The explanation string will be printed with the list of matches, above the group currently selected.

Within the explanation, the following sequences may be used to specify output attributes as described in the section EXPANSION OF PROMPT SEQUENCES in zshmisc(1): `%B', `%S', `%U', `%F', `%K' and their lower case coun? terparts, as well as `%{...%}'. `%F', `%K' and `%{...%}' take arguments in the same form as prompt expansion. (Note that the sequence `%G' is not available; an argu? ment to `%{' should be used instead.) The sequence `%%' produces a literal `%'.

These sequences are most often employed by users when customising the format style (see zshcompsys(1)), but they must also be taken into account when writing comple? tion functions, as passing descriptions with unescaped `%' characters to utility functions such as _arguments and _message may produce unexpected results. If arbitrary text is to be passed in a description, it can be escaped using e.g. ${my_str//\%/%%}.

-x message

Like -X, but the message will be printed even if there are no matches in the group.

-q The suffix given with -S will be automatically removed if the next character typed is a blank or does not insert anything, or if the suffix consists of only one character and the next character typed is the same character.

-r remove-chars

This is a more versatile form of the -q option. The suf? fix given with -S or the slash automatically added after completing directories will be automatically removed if the next character typed inserts one of the characters given in the remove-chars. This string is parsed as a characters class and understands the backslash sequences used by the print command. For example, `-r "a-z\t"' re? moves the suffix if the next character typed inserts a

lower case character or a TAB, and `-r "^0-9"' removes the suffix if the next character typed inserts anything but a digit. One extra backslash sequence is understood in this string: `\-' stands for all characters that in? sert nothing. Thus `-S "=" -q' is the same as `-S "=" -r "= \t\n\-"'.

This option may also be used without the -S option; then any automatically added space will be removed when one of the characters in the list is typed.

-R remove-func

This is another form of the -r option. When a suffix has been inserted and the completion accepted, the function remove-func will be called after the next character typed. It is passed the length of the suffix as an argu? ment and can use the special parameters available in or? dinary (non-completion) zle widgets (see zshzle(1)) to analyse and modify the command line.

-f If this flag is given, all of the matches built from words are marked as being the names of files. They are not required to be actual filenames, but if they are, and the option LIST_TYPES is set, the characters describing the types of the files in the completion lists will be shown. This also forces a slash to be added when the name of a directory is completed.

-e This flag can be used to tell the completion code that the matches added are parameter names for a parameter ex? pansion. This will make the AUTO_PARAM_SLASH and AUTO_PARAM_KEYS options be used for the matches.

-W file-prefix

This string is a pathname that will be prepended to each of the matches formed by the given words together with any prefix specified by the -p option to form a complete filename for testing. Hence it is only useful if com?

bined with the -f flag, as the tests will not otherwise be performed.

-F array

Specifies an array containing patterns. Words matching one of these patterns are ignored, i.e. not considered to be possible matches.

The array may be the name of an array parameter or a list of literal patterns enclosed in parentheses and quoted, as in `-F "(*?.o *?.h)"'. If the name of an array is given, the elements of the array are taken as the pat? terns.

-Q This flag instructs the completion code not to quote any metacharacters in the words when inserting them into the command line.

-M match-spec

This gives local match specifications as described below in the section `Completion Matching Control'. This option may be given more than once. In this case all match-specs given are concatenated with spaces between them to form the specification string to use. Note that they will only be used if the -U option is not given.

-n Specifies that the words added are to be used as possible matches, but are not to appear in the completion listing.

-U If this flag is given, all words given will be accepted and no matching will be done by the completion code. Nor? mally this is used in functions that do the matching themselves.

-O array

If this option is given, the words are not added to the set of possible completions. Instead, matching is done as usual and all of the words given as arguments that match the string on the command line will be stored in the array parameter whose name is given as array.

-A array

As the -O option, except that instead of those of the words which match being stored in array, the strings gen? erated internally by the completion code are stored. For example, with a matching specification of `-M "L:|no=", the string `nof' on the command line and the string `foo' as one of the words, this option stores the string `no? foo' in the array, whereas the -O option stores the `foo' originally given.

-D array

As with -O, the words are not added to the set of possi? ble completions. Instead, the completion code tests whether each word in turn matches what is on the line. If the nth word does not match, the nth element of the array is removed. Elements for which the corresponding word is matched are retained.

-C This option adds a special match which expands to all other matches when inserted into the line, even those that are added after this option is used. Together with the -d option it is possible to specify a string that should be displayed in the list for this special match. If no string is given, it will be shown as a string con? taining the strings that would be inserted for the other matches, truncated to the width of the screen.

-E number

This option adds number empty matches after the words have been added. An empty match takes up space in com? pletion listings but will never be inserted in the line and can't be selected with menu completion or menu selec? tion. This makes empty matches only useful to format completion lists and to make explanatory string be shown in completion lists (since empty matches can be given display strings with the -d option). And because all but

one empty string would otherwise be removed, this option implies the -V and -2 options (even if an explicit -J option is given). This can be important to note as it affects the name space into which matches are added.

-

-- This flag ends the list of flags and options. All arguments after it will be taken as the words to use as matches even if they begin with hyphens.

Except for the -M flag, if any of these flags is given more than once, the first one (and its argument) will be used.

compset -p number

compset -P [number] pattern

compset -s number

compset -S [number] pattern

compset -n begin [end]

compset -N beg-pat [end-pat]

compset -q

This command simplifies modification of the special parameters, while its return status allows tests on them to be carried out.

The options are:

-p number

If the value of the PREFIX parameter is at least number characters long, the first number characters are removed from it and appended to the contents of the IPREFIX parameter.

-P [number] pattern

If the value of the PREFIX parameter begins with anything that matches the pattern, the matched portion is removed from PREFIX and appended to IPREFIX.

Without the optional number, the longest match is taken, but if number is given, anything up to the numberth match is moved. If the number is negative, the numberth longest match is moved. For example, if PREFIX contains the

string `a=b=c', then compset -P '*\=' will move the string `a=b=' into the IPREFIX parameter, but compset -P 1 '*\=' will move only the string `a='.

-s number

As -p, but transfer the last number characters from the value of SUFFIX to the front of the value of ISUFFIX.

-S [number] pattern

As -P, but match the last portion of SUFFIX and transfer the matched portion to the front of the value of ISUFFIX.

-n begin [end]

If the current word position as specified by the parame? ter CURRENT is greater than or equal to begin, anything up to the beginth word is removed from the words array and the value of the parameter CURRENT is decremented by begin.

If the optional end is given, the modification is done only if the current word position is also less than or equal to end. In this case, the words from position end onwards are also removed from the words array.

Both begin and end may be negative to count backwards from the last element of the words array.

-N beg-pat [end-pat]

If one of the elements of the words array before the one at the index given by the value of the parameter CURRENT matches the pattern beg-pat, all elements up to and in? cluding the matching one are removed from the words array and the value of CURRENT is changed to point to the same word in the changed array.

If the optional pattern end-pat is also given, and there is an element in the words array matching this pattern, the parameters are modified only if the index of this word is higher than the one given by the CURRENT parame? ter (so that the matching word has to be after the cur?

sor). In this case, the words starting with the one matching end-pat are also removed from the words array. If words contains no word matching end-pat, the testing and modification is performed as if it were not given.

-q The word currently being completed is split on spaces into separate words, respecting the usual shell quoting conventions. The resulting words are stored in the words array, and CURRENT, PREFIX, SUFFIX, QIPREFIX, and QISUF? FIX are modified to reflect the word part that is com? pleted.

In all the above cases the return status is zero if the test succeeded and the parameters were modified and non-zero other? wise. This allows one to use this builtin in tests such as:

if compset -P '*\='; then ...

This forces anything up to and including the last equal sign to be ignored by the completion code.

compcall [-TD]

This allows the use of completions defined with the compctl builtin from within completion widgets. The list of matches will be generated as if one of the non-widget completion func? tions (complete-word, etc.) had been called, except that only compctls given for specific commands are used. To force the code to try completions defined with the -T option of compctl and/or the default completion (whether defined by compctl -D or the builtin default) in the appropriate places, the -T and/or -D flags can be passed to compcall.

The return status can be used to test if a matching compctl def? inition was found. It is non-zero if a compctl was found and zero otherwise.

Note that this builtin is defined by the zsh/compctl module.

COMPLETION CONDITION CODES

The following additional condition codes for use within the [[...]] construct are available in completion widgets. These work on the spe?

cial parameters. All of these tests can also be performed by the compset builtin, but in the case of the condition codes the contents of the special parameters are not modified.

-prefix [number] pattern

true if the test for the -P option of compset would succeed.

-suffix [number] pattern

true if the test for the -S option of compset would succeed.

-after beg-pat

true if the test of the -N option with only the beg-pat given would succeed.

-between beg-pat end-pat

true if the test for the -N option with both patterns would suc? ceed.

COMPLETION MATCHING CONTROL

It is possible by use of the -M option of the compadd builtin command to specify how the characters in the string to be completed (referred to here as the command line) map onto the characters in the list of matches produced by the completion code (referred to here as the trial completions). Note that this is not used if the command line contains a glob pattern and the GLOB_COMPLETE option is set or the pattern_match of the compstate special association is set to a non-empty string.

The match-spec given as the argument to the -M option (see `Completion Builtin Commands' above) consists of one or more matching descriptions separated by whitespace. Each description consists of a letter fol? lowed by a colon and then the patterns describing which character se? quences on the line match which character sequences in the trial com? pletion. Any sequence of characters not handled in this fashion must match exactly, as usual.

The forms of match-spec understood are as follows. In each case, the form with an upper case initial character retains the string already typed on the command line as the final result of completion, while with a lower case initial character the string on the command line is changed into the corresponding part of the trial completion.

m:lpat=tpat

M:lpat=tpat

Here, lpat is a pattern that matches on the command line, corre?
sponding to tpat which matches in the trial completion.

l:lanchor|lpat=tpat

L:lanchor|lpat=tpat

l:lanchor||ranchor=tpat

L:lanchor||ranchor=tpat

b:lpat=tpat

B:lpat=tpat

These letters are for patterns that are anchored by another pat?
tern on the left side. Matching for lpat and tpat is as for m
and M, but the pattern lpat matched on the command line must be
preceded by the pattern lanchor. The lanchor can be blank to
anchor the match to the start of the command line string; other?
wise the anchor can occur anywhere, but must match in both the
command line and trial completion strings.
If no lpat is given but a ranchor is, this matches the gap be?
tween substrings matched by lanchor and ranchor. Unlike lanchor,
the ranchor only needs to match the trial completion string.
The b and B forms are similar to l and L with an empty anchor,
but need to match only the beginning of the word on the command
line or trial completion, respectively.

r:lpat|ranchor=tpat

R:lpat|ranchor=tpat

r:lanchor||ranchor=tpat

R:lanchor||ranchor=tpat

e:lpat=tpat

E:lpat=tpat

As l, L, b and B, with the difference that the command line and
trial completion patterns are anchored on the right side. Here
an empty ranchor and the e and E forms force the match to the
end of the command line or trial completion string.

x: This form is used to mark the end of matching specifications: subsequent specifications are ignored. In a single standalone list of specifications this has no use but where matching speci? fications are accumulated, such as from nested function calls, it can allow one function to override another.

Each lpat, tpat or anchor is either an empty string or consists of a sequence of literal characters (which may be quoted with a backslash), question marks, character classes, and correspondence classes; ordinary shell patterns are not used. Literal characters match only themselves, question marks match any character, and character classes are formed as for globbing and match any character in the given set.

Correspondence classes are defined like character classes, but with two differences: they are delimited by a pair of braces, and negated classes are not allowed, so the characters ! and ^ have no special meaning directly after the opening brace. They indicate that a range of characters on the line match a range of characters in the trial com? pletion, but (unlike ordinary character classes) paired according to the corresponding position in the sequence. For example, to make any ASCII lower case letter on the line match the corresponding upper case letter in the trial completion, you can use `m:{a-z}={A-Z}' (however, see below for the recommended form for this). More than one pair of classes can occur, in which case the first class before the = corre? sponds to the first after it, and so on. If one side has more such classes than the other side, the superfluous classes behave like normal character classes. In anchor patterns correspondence classes also be? have like normal character classes.

The standard `[:name:]' forms described for standard shell patterns (see the section FILENAME GENERATION in zshexpn(1)) may appear in cor? respondence classes as well as normal character classes. The only spe? cial behaviour in correspondence classes is if the form on the left and the form on the right are each one of [:upper:], [:lower:]. In these cases the character in the word and the character on the line must be the same up to a difference in case. Hence to make any lower case

character on the line match the corresponding upper case character in the trial completion you can use `m:{[:lower:]}={[:upper:]}'. Although the matching system does not yet handle multibyte characters, this is likely to be a future extension, at which point this syntax will handle arbitrary alphabets; hence this form, rather than the use of explicit ranges, is the recommended form. In other cases `[:name:]' forms are allowed. If the two forms on the left and right are the same, the characters must match exactly. In remaining cases, the corresponding tests are applied to both characters, but they are not otherwise con? strained; any matching character in one set goes with any matching character in the other set: this is equivalent to the behaviour of or? dinary character classes.

The pattern tpat may also be one or two stars, `*' or `**'. This means that the pattern on the command line can match any number of characters in the trial completion. In this case the pattern must be anchored (on either side); in the case of a single star, the anchor then determines how much of the trial completion is to be included -- only the charac? ters up to the next appearance of the anchor will be matched. With two stars, substrings matched by the anchor can be matched, too.

Examples:

The keys of the options association defined by the parameter module are the option names in all-lower-case form, without underscores, and with? out the optional no at the beginning even though the builtins setopt and unsetopt understand option names with upper case letters, under? scores, and the optional no. The following alters the matching rules so that the prefix no and any underscore are ignored when trying to match the trial completions generated and upper case letters on the line match the corresponding lower case letters in the words:

```
compadd -M 'L:|[nN][oO]= M:_= M:{[:upper:]}={[:lower:]}' - \
  ${(k)options}
```

The first part says that the pattern `[nN][oO]' at the beginning (the empty anchor before the pipe symbol) of the string on the line matches the empty string in the list of words generated by completion, so it

will be ignored if present. The second part does the same for an under? score anywhere in the command line string, and the third part uses cor? respondence classes so that any upper case letter on the line matches the corresponding lower case letter in the word. The use of the upper case forms of the specification characters (L and M) guarantees that what has already been typed on the command line (in particular the pre? fix no) will not be deleted.

Note that the use of L in the first part means that it matches only when at the beginning of both the command line string and the trial completion. I.e., the string `_NO_f' would not be completed to `_NO_foo', nor would `NONO_f' be completed to `NONO_foo' because of the leading underscore or the second `NO' on the line which makes the pat? tern fail even though they are otherwise ignored. To fix this, one would use `B:[nN][oO]=' instead of the first part. As described above, this matches at the beginning of the trial completion, independent of other characters or substrings at the beginning of the command line word which are ignored by the same or other match-specs.

The second example makes completion case insensitive. This is just the same as in the option example, except here we wish to retain the char? acters in the list of completions:

```
compadd -M 'm:{[:lower:]}={[:upper:]}' ...
```

This makes lower case letters match their upper case counterparts. To make upper case letters match the lower case forms as well:

```
compadd -M 'm:{[:lower:][:upper:]}={[:upper:][:lower:]}' ...
```

A nice example for the use of * patterns is partial word completion. Sometimes you would like to make strings like `c.s.u' complete to strings like `comp.source.unix', i.e. the word on the command line con? sists of multiple parts, separated by a dot in this example, where each part should be completed separately -- note, however, that the case where each part of the word, i.e. `comp', `source' and `unix' in this example, is to be completed from separate sets of matches is a differ? ent problem to be solved by the implementation of the completion wid? get. The example can be handled by:

compadd -M 'r:|.=* r:|=*' \

- comp.sources.unix comp.sources.misc ...

The first specification says that lpat is the empty string, while an? chor is a dot; tpat is *, so this can match anything except for the `.' from the anchor in the trial completion word. So in `c.s.u', the matcher sees `c', followed by the empty string, followed by the anchor `.', and likewise for the second dot, and replaces the empty strings before the anchors, giving `c[omp].s[ources].u[nix]', where the last part of the completion is just as normal.

With the pattern shown above, the string `c.u' could not be completed to `comp.sources.unix' because the single star means that no dot (matched by the anchor) can be skipped. By using two stars as in `r:|.=**', however, `c.u' could be completed to `comp.sources.unix'. This also shows that in some cases, especially if the anchor is a real pattern, like a character class, the form with two stars may result in more matches than one would like.

The second specification is needed to make this work when the cursor is in the middle of the string on the command line and the option COM? PLETE_IN_WORD is set. In this case the completion code would normally try to match trial completions that end with the string as typed so far, i.e. it will only insert new characters at the cursor position rather than at the end. However in our example we would like the code to recognise matches which contain extra characters after the string on the line (the `nix' in the example). Hence we say that the empty string at the end of the string on the line matches any characters at the end of the trial completion.

More generally, the specification

compadd -M 'r:|[.,_-]=* r:|=*' ...

allows one to complete words with abbreviations before any of the char? acters in the square brackets. For example, to complete veryverylong? file.c rather than veryverylongheader.h with the above in effect, you can just type very.c before attempting completion.

The specifications with both a left and a right anchor are useful to

complete partial words whose parts are not separated by some special character. For example, in some places strings have to be completed that are formed `LikeThis' (i.e. the separate parts are determined by a leading upper case letter) or maybe one has to complete strings with trailing numbers. Here one could use the simple form with only one an? chor as in:

compadd -M 'r:|[[:upper:]0-9]=* r:|=*' LikeTHIS FooHoo 5foo123 5bar234

But with this, the string `H' would neither complete to `FooHoo' nor to `LikeTHIS' because in each case there is an upper case letter before the `H' and that is matched by the anchor. Likewise, a `2' would not be completed. In both cases this could be changed by using `r:|[[:up? per:]0-9]=**', but then `H' completes to both `LikeTHIS' and `FooHoo' and a `2' matches the other strings because characters can be inserted before every upper case letter and digit. To avoid this one would use:

compadd -M 'r:[^[:upper:]0-9]||[[:upper:]0-9]=** r:|=*' \
LikeTHIS FooHoo foo123 bar234

By using these two anchors, a `H' matches only upper case `H's that are immediately preceded by something matching the left anchor `[^[:up? per:]0-9]'. The effect is, of course, that `H' matches only the string `FooHoo', a `2' matches only `bar234' and so on.

When using the completion system (see zshcompsys(1)), users can define match specifications that are to be used for specific contexts by using the matcher and matcher-list styles. The values for the latter will be used everywhere.

COMPLETION WIDGET EXAMPLE

The first step is to define the widget:

zle -C complete complete-word complete-files

Then the widget can be bound to a key using the bindkey builtin com? mand:

bindkey '^X\t' complete

After that the shell function complete-files will be invoked after typ? ing control-X and TAB. The function should then generate the matches, e.g.:

complete-files () { compadd - * }

This function will complete files in the current directory matching the current word.