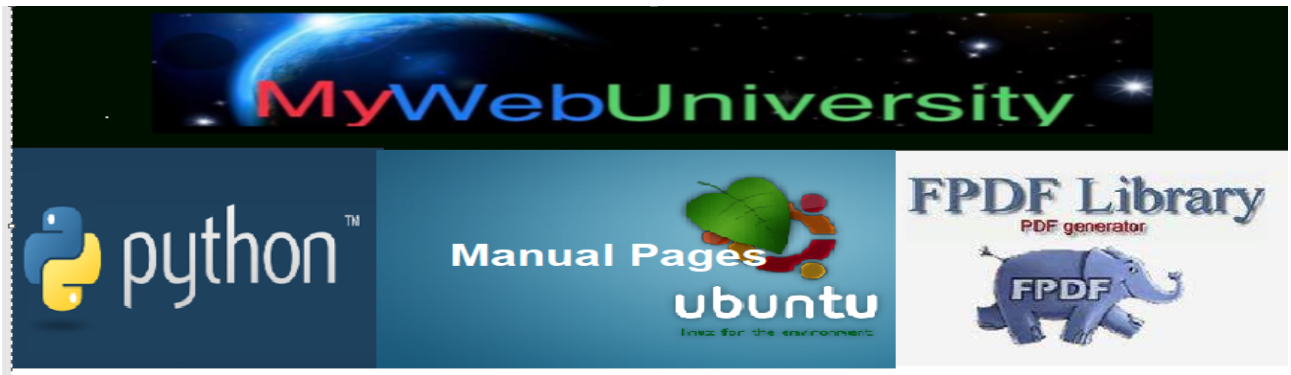

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 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 interest 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 zshcompctl(1).

Completion widgets are defined by the -C option to the zle builtin command 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, expand-or-complete, expand-or-complete-pre? fix, menu-complete, menu-expand-or-complete, reverse-menu-complete, list-choices, or delete-char-or-list. Note that this will still work even if the widget in question has been re-bound.

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

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

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 parameters have special meaning; outside these functions they are not special to the shell in any way. These para? meters are used to pass information between the completion code and the completion widget.

Some of the builtin commands and the condition codes use or change the current values of these parameters. Any existing values will be hidden during execution of completion wid? gets; except for compstate, the parameters are reset on each function exit (including nested function calls from within the completion widget) to the values they had when the function was entered.

CURRENT

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

IPREFIX

Initially this will be set to the empty string. This parameter functions like PRE? FIX; 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 com?
pleted. 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' recur?
sively. With this key it is possible to test the types of quoted strings
which are currently broken into parts in this fashion. Its value contains one
character for each quoting level. The characters are a single quote or a dou?
ble quote for strings quoted with these characters, a dollars sign for strings
quoted with $'...' and a backslash for strings not starting with a quote char?
acter. The first character in the value always corresponds to the innermost
quoting level.

context

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

array_value

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

brace_parameter

when completing the name of a parameter in a parameter expansion begin?

ning 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 parameter assignment.

command
when completing for a normal command (either in command position or for an argument of the command).

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

math when completing in a mathematical environment such as a `((...))' con? struct.

parameter
when completing the name of a parameter in a parameter expansion begin? ning with $ but not ${.

redirect
when completing after a redirection operator.

subscript
when completing inside a parameter subscript.

value when completing the value of a parameter assignment.

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

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

ignored
The number of completions 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 in? serted; if set to automenu-unambiguous, the common prefix is to be inserted and the next invocation of the completion code may start menu completion (due to the AUTO_MENU option being set); if set to menu or automenu menu completion will be started for the matches currently generated (in the latter case this will happen because the AUTO_MENU is set). The value may also contain the string `tab' when the completion code would normally not really do completion, but only insert the TAB character.

On exit it may be set to any of the values above (where setting it to the empty string is the same as unsetting it), or to a number, in which case the match whose number is given will be inserted into the command line. Negative numbers count backward from the last match (with `-1' selecting the last match) and out-of-range values are wrapped around, so that a value of zero se? lects the last match and a value one more than the maximum selects the first.

Unless the value of this key ends in a space, the match is inserted as in a menu completion, i.e. without automatically appending a space.

Both menu and automenu may also specify the number of the match to insert, given after a colon. For example, `menu:2' says to start menu completion, be? ginning with the second match.

Note that a value containing the substring `tab' makes the matches generated be ignored and only the TAB be inserted.

Finally, it may also be set to all, which makes all matches generated be in? serted into the line.

insert_positions

When the completion system inserts an unambiguous string into the line, there may be multiple places where characters are missing or where the character in? serted 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 com? pletions has been displayed. Initially this is set or unset according to the

ALWAYS_LAST_PROMPT option.

list This controls whether or how the list of matches will be displayed. If it is unset or empty they will never be listed; if its value begins with list, they will always be listed; if it begins with autolist or ambiguous, they will be listed when the AUTO_LIST or LIST_AMBIGUOUS options respectively would nor? mally cause them to be.

If the substring force appears in the value, this makes the list be shown even if there is only one match. Normally, the list would be shown only if there are at least two matches.

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

Finally, if the value contains the string explanations, only the explanation strings, if any, will be listed and if it contains messages, only the messages (added with the -x option of compadd) will be listed. If it contains both ex? planations 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 added 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 in?

serted 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 in?

serted after the old one.

old_list

This is set to yes if there is still a valid list of completions from a previ?

ous 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_match (see below).

If it is set to any other non-empty string by the user and menu completion is

not selected by other option settings, the code will instead insert any common

prefix for the generated matches as with normal completion.

pattern_match

Locally controls the behaviour given by the GLOB_COMPLETE option. Initially

it is set to `*' if and only if the option is set. The completion widget may

set it to this value, to an empty string (which has the same effect as unset?

ting it), or to any other non-empty string. If it is non-empty, unquoted

metacharacters on the command line will be treated as patterns; if it is `*',

then additionally a wildcard `*' is assumed at the cursor position; if it is

empty or unset, metacharacters will be treated literally.

Note that the match specifications given to the compadd builtin command are

not used if this is set to a non-empty string.

quote When completing inside quotes, this contains the quotation character (i.e. ei?
ther 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. Oth?
erwise 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 pa?
rameters mentioned above (words, CURRENT, PREFIX, IPREFIX, SUFFIX, and ISUF?
FIX) to be restored to their previous values when the function exits. If a
function unsets it or sets it to any other string, they will not be restored.

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

unambiguous
This key is read-only and will always be set to the common (unambiguous) pre?
fix the completion code has generated for all matches added so far.

unambiguous_cursor
This gives the position the cursor would be placed at if the common prefix in
the unambiguous key were inserted, relative to the value of that key. The cur?
sor 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] [--] [completions ...]

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

The completion code breaks each match into seven fields in the order:

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

The first field is an ignored prefix taken from the command line, the contents of the IPREFIX parameter plus the string given with the -i option. With the -U option, only the string from the -i option is used. The field <apre> is an optional prefix string given with the -P option. The <hpre> field is a string that is considered part of the match but that should not be shown when listing completions, given with the -p option; for example, functions that do filename generation might specify a common path prefix this way. <body> is the part of the match that should appear in the list of matches shown to the user. 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 each match. 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 each match.

-p hidden-prefix

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

-s hidden-suffix

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

-i ignored-prefix

This gives a string to insert just before any string given with the `-P' op? tion. Without `-P' the string is inserted before the string given with `-p' or directly before each match.

-I ignored-suffix

Like -i, but gives an ignored suffix.

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

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

-d array

This adds per-completion display strings. The array should contain one element per completion given. The completion code will then display the first element instead of the first completion, 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 paren? theses.

If there are fewer display strings than completions, the leftover completions

will be displayed unchanged and if there are more display strings than comple?
tions, 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-sepa? rated list comprising the following possible values. These values can be ab? breviated 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 specified but the order argument is omitted.

nosort This specifies that the completions 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 numerically rather than lexi? cographically.

reverse

Arrange the matches backwards by reversing the sort ordering.

-J group-name

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

-V group-name

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

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

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

-X explanation

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

Within the explanation, the following sequences may be used to specify output attributes as described in the section EXPANSION OF PROMPT SEQUENCES in zsh? misc(1): `%B', `%S', `%U', `%F', `%K' and their lower case counterparts, 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 writ? ing completion functions, as passing descriptions with unescaped `%' charac? ters to utility functions such as _arguments and _message may produce unex? pected 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 suffix given with -S or the slash automatically added after completing directories will be automati? cally 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 under? stands the backslash sequences used by the print command. For example, `-r "a-z\t"' removes 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 insert 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 match has been accepted and a suffix has been inserted, the function remove-func will be called after the next character typed. It is passed the length of the suffix as an argument and can use the special parameters available in ordinary (non-completion) zle widgets (see zshzle(1)) to analyse and modify the command line.

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

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

-W file-prefix

This string is a pathname that will be prepended to each match together with any prefix specified by the -p option to form a complete filename for testing. Hence it is only useful if combined with the -f flag, as the tests will not otherwise be performed.

-F array

Specifies an array containing patterns. completions that match one of these patterns are ignored, that is, not considered to be matches.

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

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

-M match-spec

This gives local match specifications as described below in the section `Com? pletion 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 matching completions are to be added to the set of matches, but are not to be listed to the user.

-U If this flag is given, all completions are added to the set of matches and no matching will be done by the completion code. Normally this is used in functions that do the matching themselves.

-O array

If this option is given, the completions are not added to the set of matches. Instead, matching is done as usual and all of the completions that match 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 completions which match being stored in array, the strings generated internally by the completion code are stored. For example, with a match specification of `-M "L:|no=", a current word of `nof' and completions of `foo', this option stores the string `nofoo' in the array, whereas the -O option stores the `foo' originally given.

-D array

As with -O, the completions are not added to the set of matches. Instead, whenever the nth completion does not match, the nth element of the array is removed. Elements for which the corresponding completion matches are retained. This option can be used more than once to remove elements from multiple arrays.

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

-E number

This option adds number empty matches after matching completions have been added. An empty match takes up space in completion listings but will never be inserted in the line and can't be selected with menu completion or menu selection. 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 completions 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 sta? tus 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 parameter 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 including the matching one are removed from the words array and the value of CURRENT is changed to point to the same word in the changed array.

If the optional pattern end-pat is also given, and there is an element in the words array matching this pattern, the parameters are modified only if the in? dex of this word is higher than the one given by the CURRENT parameter (so that the matching word has to be after the cursor). In this case, the words starting with the one matching end-pat are also removed from the words array.

If words contains no word matching end-pat, the testing and modification is performed as if it were not given.

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

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

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

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

compcall [-TD]

This allows the use of completions defined with the compctl builtin from within completion widgets. The list of matches will be generated as if one of the non-widget completion functions (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 definition was found. It is non-zero if a compctl was found and zero otherwise.

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

COMPLETION CONDITION CODES

The following additional condition codes for use within the [[...]] construct are available in completion widgets. These work on the special parameters. All of these tests can also be performed by the compset builtin, but in the case of the condition codes the contents of the special parameters are not modified.

-prefix [number] pattern

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

-suffix [number] pattern

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

-after beg-pat

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

-between beg-pat end-pat

true if the test for the -N option with both patterns would succeed.

COMPLETION MATCHING CONTROL

When the user invokes completion, the current word on the command line (that is, the word the cursor is currently on) is used to generate a match pattern. Only those completions that match the pattern are offered to the user as matches.

The default match pattern is generated from the current word by either

? appending a `*' (matching any number of characters in a completion) or,

? if the shell option COMPLETE_IN_WORD is set, inserting a `*' at the cursor position.

This narrow pattern can be broadened selectively by passing a match specification to the compadd builtin command through its -M option (see `Completion Builtin Commands' above). A match specification consists of one or more matchers separated by whitespace. Matchers in a match specification are applied one at a time, from left to right. Once all matchers have been applied, completions are compared to the final match pattern and non-matching ones are discarded.

? Note that the -M option is ignored if the current word contains a glob pattern and the shell option GLOB_COMPLETE is set or if the pattern_match key of the special as? sociative array compstate is set to a non-empty value (see `Completion Special Para? meters' above).

? Users of the completion system (see zshcompsys(1)) should generally not use the -M option directly, but rather use the matcher-list and matcher styles (see the subsec? tion Standard Styles in the documentation for COMPLETION SYSTEM CONFIGURATION in zsh? compsys(1)).

Each matcher consists of

? a case-sensitive letter

? a `:',

? one or more patterns separated by pipes (`|'),

? an equals sign (`='), and

? another pattern.

The patterns before the `=' are used to match substrings of the current word. For each matched substring, the corresponding part of the match pattern is broadened with the pattern after the `=', by means of a logical OR.

Each pattern in a matcher cosists of either

? the empty string or

? a sequence of

  ? literal characters (which may be quoted with a `\'),

  ? question marks (`?'),

  ? bracket expressions (`[...]'; see the subsection Glob Operators in the docu? mentation for GLOB OPERATORS in zshexpn(1)), and/or

  ? brace expressions (see below).

Other shell patterns are not allowed.

A brace expression, like a bracket expression, consists of a list of

? literal characters,

? ranges (`0-9'), and/or

? character classes (`[:name:]').

However, they differ from each other as follows:

? A brace expression is delimited by a pair of braces (`{...}').

? Brace expressions do not support negations. That is, an initial `!' or `^' has no special meaning and will be interpreted as a literal character.

? When a character in the current word matches the nth pattern in a brace expression, the corresponding part of the match pattern is broadened only with the nth pattern of the brace expression on the other side of the `=', if there is one; if there is no brace expression on the other side, then this pattern is the empty string. However, if either brace expression has more elements than the other, then the excess entries are simply ignored. When comparing indexes, each literal character or character class counts as one element, but each range is instead expanded to the full list of literal characters it represents. Additionally, if on both sides of the `=', the nth pattern is `[:upper:]' or `[:lower:]', then these are expanded as ranges, too.

Note that, although the matching system does not yet handle multibyte characters, this is likely to be a future extension. Hence, using `[:upper:]' and `[:lower:]' is recommended over `A-Z' and `a-z'.

Below are the different forms of matchers supported. Each uppercase form behaves exactly like its lowercase counterpart, but adds an additional step after the match pattern has fil? tered out non-matching completions: Each of a match's substrings that was matched by a sub? pattern from an uppercase matcher is replaced with the corresponding substring of the cur? rent word. However, patterns from lowercase matchers have higher weight: If a substring of the current word was matched by patterns from both a lowercase and an uppercase matcher, then the lowercase matcher's pattern wins and the corresponding part of the match is not modified.

Unless indicated otherwise, each example listed assumes COMPLETE_IN_WORD to be unset (as it is by default).

m:word-pat=match-pat

M:word-pat=match-pat

For each substring of the current word that matches word-pat, broaden the correspond?

ing part of the match pattern to additionally match match-pat.

Examples:

m:{[:lower:]}={[:upper:]} lets any lower case character in the current word be completed to itself or its uppercase counterpart. So, the completions `foo', `FOO' and `Foo' will are be considered matches for the word `fo'.

M:_= inserts every underscore from the current word into each match, in the same relative position, determined by matching the substrings around it. So, given a completion `foo', the word `f_o' will be completed to the match `f_oo', even though the latter was not present as a completion.

b:word-pat=match-pat

B:word-pat=match-pat

e:word-pat=match-pat

E:word-pat=match-pat

For each consecutive substring at the b:eginning or e:nd of the current word that matches word-pat, broaden the corresponding part of the match pattern to additionally match match-pat.

Examples:

`b:-=+' lets any number of minuses at the start of the current word be com? pleted to a minus or a plus.

`B:0=' adds all zeroes at the beginning of the current word to the beginning of each match.

l:|word-pat=match-pat

L:|word-pat=match-pat

R:word-pat|=match-pat

r:word-pat|=match-pat

If there is a substring at the l:eft or r:ight edge of the current word that matches word-pat, then broaden the corresponding part of the match pattern to additionally match match-pat.

For each l:, L:, r: and R: matcher (including the ones below), the pattern match-pat may also be a `*'. This matches any number of characters in a completion.

Examples:

`r:|=*' appends a `*' to the match pattern, even when COMPLETE_IN_WORD is set and the cursor is not at the end of the current word.

If the current word starts with a minus, then `L:|-=' will prepend it to each match.

l:anchor|word-pat=match-pat

L:anchor|word-pat=match-pat

r:word-pat|anchor=match-pat

R:word-pat|anchor=match-pat

For each substring of the current word that matches word-pat and has on its l:eft or r:ight another substring matching anchor, broaden the corresponding part of the match pattern to additionally match match-pat.

Note that these matchers (and the ones below) modify only what is matched by word-pat; they do not change the matching behavior of what is matched by anchor (or coanchor; see the matchers below). Thus, unless its corresponding part of the match pattern has been modified, the anchor in the current word has to match literally in each completion, just like any other substring of the current word.

If a matcher includes at least one anchor (which includes the matchers with two anchors, below), then match-pat may also be `*' or `**'. `*' can match any part of a completion that does not contain any substrings matching anchor, whereas a `**' can match any part of a completion, period. (Note that this is different from the behavior of `*' in the anchorless forms of `l:' and `r:' and and also different from `*' and `**' in glob expressions.)

Examples:

`r:|.=*' makes the completion `comp.sources.unix' a match for the word `..u' -- but not for the word `.u'.

Given a completion `--foo', the matcher `L:--|no-=' will complete the word `--no-' to the match `--no-foo'.

l:anchor||coanchor=match-pat

L:anchor||coanchor=match-pat

r:coanchor||anchor=match-pat

R:coanchor||anchor=match-pat

For any two consecutive substrings of the current word that match anchor and coanchor, in the order given, insert the pattern match-pat between their corresponding parts in the match pattern.

Note that, unlike anchor, the pattern coanchor does not change what `*' can match.

Examples:

`r:?||[[:upper:]]=*' will complete the current word `fB' to `fooBar', but it will not complete it to `fooHooBar' (because `*' here cannot match anything that includes a match for `[[:upper:]]), nor will it complete `B' to `fooBar' (because there is no character in the current word to match coanchor).

Given the current word `pass.n' and a completion `pass.byname', the matcher `L:.||[[:alpha:]]=by' will produce the match `pass.name'.

x:

Ignore this matcher and all matchers to its right.

This matcher is used to mark the end of a match specification. In a single stand? alone list of matchers, this has no use, but where match specifications are concate? nated, as is often the case when using the completion system (see zshcompsys(1)), it can allow one match specification to override another.

COMPLETION WIDGET EXAMPLE

The first step is to define the widget:

```
zle -C complete complete-word complete-files
```

Then the widget can be bound to a key using the bindkey builtin command:

```
bindkey '^X\t' complete
```

After that the shell function complete-files will be invoked after typing control-X and TAB.

The function should then generate the matches, e.g.:

```
complete-files () { compadd - * }
```

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