NAME

zshzle – zsh command line editor

DESCRIPTION

If the **ZLE** option is set (which it is by default in interactive shells) and the shell input is attached to the terminal, the user is able to edit command lines.

There are two display modes. The first, multiline mode, is the default. It only works if the **TERM** parameter is set to a valid terminal type that can move the cursor up. The second, single line mode, is used if **TERM** is invalid or incapable of moving the cursor up, or if the **SINGLE_LINE_ZLE** option is set. This mode is similar to **ksh**, and uses no termcap sequences. If **TERM** is "emacs", the **ZLE** option will be unset by default.

The parameters **BAUD**, **COLUMNS**, and **LINES** are also used by the line editor. See *Parameters Used By The Shell* in *zshparam*(1).

The parameter **zle_highlight** is also used by the line editor; see *Character Highlighting* below. Highlighting of special characters and the region between the cursor and the mark (as set with **set−mark−command** in Emacs mode, or by **visual−mode** in Vi mode) is enabled by default; consult this reference for more information. Irascible conservatives will wish to know that all highlighting may be disabled by the following setting:

zle_highlight=(none)

In many places, references are made to the **numeric argument**. This can by default be entered in emacs mode by holding the alt key and typing a number, or pressing escape before each digit, and in vi command mode by typing the number before entering a command. Generally the numeric argument causes the next command entered to be repeated the specified number of times, unless otherwise noted below; this is implemented by the **digit−argument** widget. See also the *Arguments* subsection of the *Widgets* section for some other ways the numeric argument can be modified.

KEYMAPS

A keymap in ZLE contains a set of bindings between key sequences and ZLE commands. The empty key sequence cannot be bound.

There can be any number of keymaps at any time, and each keymap has one or more names. If all of a keymap's names are deleted, it disappears. **bindkey** can be used to manipulate keymap names.

Initially, there are eight keymaps:

emacs   EMACS emulation
viins   vi emulation – insert mode
vicmd   vi emulation – command mode
viopp   vi emulation – operator pending
visual vi emulation – selection active
isearch incremental search mode
command
        read a command name
.safe   fallback keymap

The '**.safe**' keymap is special. It can never be altered, and the name can never be removed. However, it can be linked to other names, which can be removed. In the future other special keymaps may be added; users should avoid using names beginning with '**.**' for their own keymaps.

In addition to these names, either '**emacs**' or '**viins**' is also linked to the name '**main**'. If one of the **VISUAL** or **EDITOR** environment variables contain the string '**vi**' when the shell starts up then it will be '**viins**', otherwise it will be '**emacs**'. **bindkey**'s **−e** and **−v** options provide a convenient way to override this default choice.

When the editor starts up, it will select the '**main**' keymap. If that keymap doesn't exist, it will use '**.safe**' instead.

In the '**.safe**' keymap, each single key is bound to **self−insert**, except for ˆJ (line feed) and ˆM (return)

which are bound to **accept−line**. This is deliberately not pleasant to use; if you are using it, it means you deleted the main keymap, and you should put it back.

Reading Commands

When ZLE is reading a command from the terminal, it may read a sequence that is bound to some command and is also a prefix of a longer bound string. In this case ZLE will wait a certain time to see if more characters are typed, and if not (or they don't match any longer string) it will execute the binding. This timeout is defined by the **KEYTIMEOUT** parameter; its default is 0.4 sec. There is no timeout if the prefix string is not itself bound to a command.

The key timeout is also applied when ZLE is reading the bytes from a multibyte character string when it is in the appropriate mode. (This requires that the shell was compiled with multibyte mode enabled; typically also the locale has characters with the UTF−8 encoding, although any multibyte encoding known to the operating system is supported.) If the second or a subsequent byte is not read within the timeout period, the shell acts as if **?** were typed and resets the input state.

As well as ZLE commands, key sequences can be bound to other strings, by using '**bindkey −s**'. When such a sequence is read, the replacement string is pushed back as input, and the command reading process starts again using these fake keystrokes. This input can itself invoke further replacement strings, but in order to detect loops the process will be stopped if there are twenty such replacements without a real command being read.

A key sequence typed by the user can be turned into a command name for use in user−defined widgets with the **read−command** widget, described in the subsection 'Miscellaneous' of the section 'Standard Widgets' below.

Local Keymaps

While for normal editing a single keymap is used exclusively, in many modes a local keymap allows for some keys to be customised. For example, in an incremental search mode, a binding in the **isearch** keymap will override a binding in the **main** keymap but all keys that are not overridden can still be used.

If a key sequence is defined in a local keymap, it will hide a key sequence in the global keymap that is a prefix of that sequence. An example of this occurs with the binding of **iw** in **viopp** as this hides the binding of **i** in **vicmd**. However, a longer sequence in the global keymap that shares the same prefix can still apply so for example the binding of **ˆXa** in the global keymap will be unaffected by the binding of **ˆXb** in the local keymap.

ZLE BUILTINS

The ZLE module contains three related builtin commands. The **bindkey** command manipulates keymaps and key bindings; the **vared** command invokes ZLE on the value of a shell parameter; and the **zle** command manipulates editing widgets and allows command line access to ZLE commands from within shell functions.

bindkey [*options*] **−l** [**−L**] [*keymap ...*]
bindkey [*options*] **−d**
bindkey [*options*] **−D** *keymap ...*
bindkey [*options*] **−A** *old−keymap new−keymap*
bindkey [*options*] **−N** *new−keymap* [*old−keymap*]
bindkey [*options*] **−m**
bindkey [*options*] **−r** *in−string ...*
bindkey [*options*] **−s** *in−string out−string ...*
bindkey [*options*] *in−string command ...*
bindkey [*options*] [*in−string*]

> **bindkey**'s options can be divided into three categories: keymap selection for the current command, operation selection, and others. The keymap selection options are:

> **−e**   Selects keymap '**emacs**' for any operations by the current command, and also links '**emacs**' to '**main**' so that it is selected by default the next time the editor starts.

−v   Selects keymap '**viins**' for any operations by the current command, and also links '**viins**' to '**main**' so that it is selected by default the next time the editor starts.

−a   Selects keymap '**vicmd**' for any operations by the current command.

−M *keymap*
         The *keymap* specifies a keymap name that is selected for any operations by the current command.

If a keymap selection is required and none of the options above are used, the '**main**' keymap is used. Some operations do not permit a keymap to be selected, namely:

−l   List all existing keymap names; if any arguments are given, list just those keymaps.

         If the **−L** option is also used, list in the form of **bindkey** commands to create or link the keymaps. '**bindkey −lL main**' shows which keymap is linked to '**main**', if any, and hence if the standard emacs or vi emulation is in effect. This option does not show the **.safe** keymap because it cannot be created in that fashion; however, neither is '**bindkey −lL .safe**' reported as an error, it simply outputs nothing.

−d   Delete all existing keymaps and reset to the default state.

−D *keymap* ...
         Delete the named *keymap*s.

−A *old−keymap new−keymap*
         Make the *new−keymap* name an alias for *old−keymap*, so that both names refer to the same keymap. The names have equal standing; if either is deleted, the other remains. If there is already a keymap with the *new−keymap* name, it is deleted.

−N *new−keymap* [*old−keymap*]
         Create a new keymap, named *new−keymap*. If a keymap already has that name, it is deleted. If an *old−keymap* name is given, the new keymap is initialized to be a duplicate of it, otherwise the new keymap will be empty.

To use a newly created keymap, it should be linked to **main**. Hence the sequence of commands to create and use a new keymap '**mymap**' initialized from the **emacs** keymap (which remains unchanged) is:

>   **bindkey −N mymap emacs**
>   **bindkey −A mymap main**

Note that while '**bindkey −A** *newmap* **main**' will work when *newmap* is **emacs** or **viins**, it will not work for **vicmd**, as switching from vi insert to command mode becomes impossible.

The following operations act on the '**main**' keymap if no keymap selection option was given:

−m   Add the built−in set of meta−key bindings to the selected keymap. Only keys that are unbound or bound to **self−insert** are affected.

−r *in−string* ...
         Unbind the specified *in−string*s in the selected keymap. This is exactly equivalent to binding the strings to **undefined−key**.

         When **−R** is also used, interpret the *in−string*s as ranges.

         When **−p** is also used, the *in−string*s specify prefixes. Any binding that has the given *in−string* as a prefix, not including the binding for the *in−string* itself, if any, will be removed. For example,

>   **bindkey −rpM viins '^['**

         will remove all bindings in the vi−insert keymap beginning with an escape character (probably cursor keys), but leave the binding for the escape character itself (probably **vi−cmd−mode**). This is incompatible with the option **−R**.

−**s** *in−string out−string* ...

Bind each *in−string* to each *out−string*. When *in−string* is typed, *out−string* will be pushed back and treated as input to the line editor. When −**R** is also used, interpret the *in−string*s as ranges.

Note that both *in−string* and *out−string* are subject to the same form of interpretation, as described below.

in−string command ...

Bind each *in−string* to each *command*. When −**R** is used, interpret the *in−string*s as ranges.

[*in−string*]

List key bindings. If an *in−string* is specified, the binding of that string in the selected keymap is displayed. Otherwise, all key bindings in the selected keymap are displayed. (As a special case, if the −**e** or −**v** option is used alone, the keymap is *not* displayed − the implicit linking of keymaps is the only thing that happens.)

When the option −**p** is used, the *in−string* must be present. The listing shows all bindings which have the given key sequence as a prefix, not including any bindings for the key sequence itself.

When the −**L** option is used, the list is in the form of **bindkey** commands to create the key bindings.

When the −**R** option is used as noted above, a valid range consists of two characters, with an optional '−' between them. All characters between the two specified, inclusive, are bound as specified.

For either *in−string* or *out−string*, the following escape sequences are recognised:

\a	bell character
\b	backspace
\e, **\E**	escape
\f	form feed
\n	linefeed (newline)
\r	carriage return
\t	horizontal tab
\v	vertical tab
NN	character code in octal
\x*NN*	character code in hexadecimal
\u*NNNN*	
	unicode character code in hexadecimal
\U*NNNNNNNN*	
	unicode character code in hexadecimal
\M[−]*X*	character with meta bit set
\C[−]*X*	control character
ˆ*X*	control character

In all other cases, '\' escapes the following character. Delete is written as 'ˆ**?**'. Note that '**\M**ˆ**?**' and 'ˆ**\M?**' are not the same, and that (unlike emacs), the bindings '**\M**−*X*' and '**\e***X*' are entirely distinct, although they are initialized to the same bindings by '**bindkey −m**'.

vared [−**Aacghe**] [−**p** *prompt*] [−**r** *rprompt*]
   [−**M** *main−keymap*] [−**m** *vicmd−keymap*]
   [−**i** *init−widget*] [−**f** *finish−widget*]
   [−**t** *tty*] *name*

The value of the parameter *name* is loaded into the edit buffer, and the line editor is invoked. When the editor exits, *name* is set to the string value returned by the editor. When the −**c** flag is

given, the parameter is created if it doesn't already exist. The **−a** flag may be given with **−c** to cre-
ate an array parameter, or the **−A** flag to create an associative array. If the type of an existing pa-
rameter does not match the type to be created, the parameter is unset and recreated. The **−g** flag
may be given to suppress warnings from the **WARN_CREATE_GLOBAL** and
WARN_NESTED_VAR options.

If an array or array slice is being edited, separator characters as defined in **$IFS** will be shown
quoted with a backslash, as will backslashes themselves. Conversely, when the edited text is split
into an array, a backslash quotes an immediately following separator character or backslash; no
other special handling of backslashes, or any handling of quotes, is performed.

Individual elements of existing array or associative array parameters may be edited by using sub-
script syntax on *name*. New elements are created automatically, even without **−c**.

If the **−p** flag is given, the following string will be taken as the prompt to display at the left. If the
−r flag is given, the following string gives the prompt to display at the right. If the **−h** flag is spec-
ified, the history can be accessed from ZLE. If the **−e** flag is given, typing **ˆD** (Control−D) on an
empty line causes **vared** to exit immediately with a non−zero return value.

The **−M** option gives a keymap to link to the **main** keymap during editing, and the **−m** option
gives a keymap to link to the **vicmd** keymap during editing. For vi−style editing, this allows a pair
of keymaps to override **viins** and **vicmd**. For emacs−style editing, only **−M** is normally needed
but the **−m** option may still be used. On exit, the previous keymaps will be restored.

Vared calls the usual '**zle−line−init**' and '**zle−line−finish**' hooks before and after it takes control.
Using the **−i** and **−f** options, it is possible to replace these with other custom widgets.

If '**−t** *tty*' is given, *tty* is the name of a terminal device to be used instead of the default **/dev/tty**. If
tty does not refer to a terminal an error is reported.

zle
zle −l [**−L** | **−a**] [*string ...*]
zle −D *widget ...*
zle −A *old−widget new−widget*
zle −N *widget* [*function*]
zle −f *flag* [*flag...*]
zle −C *widget completion−widget function*
zle −R [**−c**] [*display−string*] [*string ...*]
zle −M *string*
zle −U *string*
zle −K *keymap*
zle −F [**−L** | **−w**] [*fd* [*handler*]]
zle −I
zle −T [**tc** *function* | **−r tc** | **−L**]
zle *widget* [**−n** *num*] [**−Nw**] [**−K** *keymap*] *args ...*
>   The **zle** builtin performs a number of different actions concerning ZLE.

>   With no options and no arguments, only the return status will be set. It is zero if ZLE is currently
>   active and widgets could be invoked using this builtin command and non−zero otherwise. Note
>   that even if non−zero status is returned, zle may still be active as part of the completion system;
>   this does not allow direct calls to ZLE widgets.

>   Otherwise, which operation it performs depends on its options:

> **−l** [**−L** | **−a**] [*string*]
>>   List all existing user−defined widgets. If the **−L** option is used, list in the form of **zle**
>>   commands to create the widgets.

>>   When combined with the **−a** option, all widget names are listed, including the builtin
>>   ones. In this case the **−L** option is ignored.

>>   If at least one *string* is given, and **−a** is present or **−L** is not used, nothing will be printed.

The return status will be zero if all *string*s are names of existing widgets and non−zero if at least one *string* is not a name of a defined widget. If **−a** is also present, all widget names are used for the comparison including builtin widgets, else only user−defined widgets are used.

If at least one *string* is present and the **−L** option is used, user−defined widgets matching any *string* are listed in the form of **zle** commands to create the widgets.

−D *widget ...*
> Delete the named *widget*s.

−A *old−widget new−widget*
> Make the *new−widget* name an alias for *old−widget*, so that both names refer to the same widget. The names have equal standing; if either is deleted, the other remains. If there is already a widget with the *new−widget* name, it is deleted.

−N *widget* [*function*]
> Create a user−defined widget. If there is already a widget with the specified name, it is overwritten. When the new widget is invoked from within the editor, the specified shell *function* is called. If no function name is specified, it defaults to the same name as the widget. For further information, see the section 'Widgets' below.

−f *flag* [*flag...]*
> Set various flags on the running widget. Possible values for *flag* are:
>
> **yank** for indicating that the widget has yanked text into the buffer. If the widget is wrapping an existing internal widget, no further action is necessary, but if it has inserted the text manually, then it should also take care to set **YANK_START** and **YANK_END** correctly. **yankbefore** does the same but is used when the yanked text appears after the cursor.
>
> **kill** for indicating that text has been killed into the cutbuffer. When repeatedly invoking a kill widget, text is appended to the cutbuffer instead of replacing it, but when wrapping such widgets, it is necessary to call '**zle −f kill**' to retain this effect.
>
> **vichange** for indicating that the widget represents a vi change that can be repeated as a whole with '**vi−repeat−change**'. The flag should be set early in the function before inspecting the value of **NUMERIC** or invoking other widgets. This has no effect for a widget invoked from insert mode. If insert mode is active when the widget finishes, the change extends until next returning to command mode.

−C *widget completion−widget function*
> Create a user−defined completion widget named *widget*. The completion widget will behave like the built−in completion−widget whose name is given as *completion−widget*. To generate the completions, the shell function *function* will be called. For further information, see *zshcompwid*(1).

−R [**−c**] [*display−string*] [*string ...]*
> Redisplay the command line; this is to be called from within a user−defined widget to allow changes to become visible. If a *display−string* is given and not empty, this is shown in the status line (immediately below the line being edited).
>
> If the optional *string*s are given they are listed below the prompt in the same way as completion lists are printed. If no *string*s are given but the **−c** option is used such a list is cleared.
>
> Note that this option is only useful for widgets that do not exit immediately after using it because the strings displayed will be erased immediately after return from the widget.
>
> This command can safely be called outside user defined widgets; if zle is active, the display will be refreshed, while if zle is not active, the command has no effect. In this case there will usually be no other arguments.

The status is zero if zle was active, else one.

−M *string*

As with the **−R** option, the *string* will be displayed below the command line; unlike the **−R** option, the string will not be put into the status line but will instead be printed normally below the prompt. This means that the *string* will still be displayed after the widget returns (until it is overwritten by subsequent commands).

−U *string*

This pushes the characters in the *string* onto the input stack of ZLE. After the widget currently executed finishes ZLE will behave as if the characters in the *string* were typed by the user.

As ZLE uses a stack, if this option is used repeatedly the last string pushed onto the stack will be processed first. However, the characters in each *string* will be processed in the order in which they appear in the string.

−K *keymap*

Selects the keymap named *keymap*. An error message will be displayed if there is no such keymap.

This keymap selection affects the interpretation of following keystrokes within this invocation of ZLE. Any following invocation (e.g., the next command line) will start as usual with the '**main**' keymap selected.

−F [**−L** | **−w**] [*fd* [*handler*]]

Only available if your system supports one of the 'poll' or 'select' system calls; most modern systems do.

Installs *handler* (the name of a shell function) to handle input from file descriptor *fd*. Installing a handler for an *fd* which is already handled causes the existing handler to be replaced. Any number of handlers for any number of readable file descriptors may be installed. Note that zle makes no attempt to check whether this *fd* is actually readable when installing the handler. The user must make their own arrangements for handling the file descriptor when zle is not active.

When zle is attempting to read data, it will examine both the terminal and the list of handled *fd*'s. If data becomes available on a handled *fd*, zle calls *handler* with the fd which is ready for reading as the first argument. Under normal circumstances this is the only argument, but if an error was detected, a second argument provides details: '**hup**' for a disconnect, '**nval**' for a closed or otherwise invalid descriptor, or '**err**' for any other condition. Systems that support only the 'select' system call always use '**err**'.

If the option **−w** is also given, the *handler* is instead a line editor widget, typically a shell function made into a widget using '**zle −N**'. In that case *handler* can use all the facilities of zle to update the current editing line. Note, however, that as handling *fd* takes place at a low level changes to the display will not automatically appear; the widget should call '**zle −R**' to force redisplay. As of this writing, widget handlers only support a single argument and thus are never passed a string for error state, so widgets must be prepared to test the descriptor themselves.

If either type of handler produces output to the terminal, it should call '**zle −I**' before doing so (see below). Handlers should not attempt to read from the terminal.

If no *handler* is given, but an *fd* is present, any handler for that *fd* is removed. If there is none, an error message is printed and status 1 is returned.

If no arguments are given, or the **−L** option is supplied, a list of handlers is printed in a form which can be stored for later execution.

An *fd* (but not a *handler*) may optionally be given with the **−L** option; in this case, the function will list the handler if any, else silently return status 1.

Note that this feature should be used with care. Activity on one of the *fd*'s which is not properly handled can cause the terminal to become unusable. Removing an *fd* handler from within a signal trap may cause unpredictable behavior.

Here is a simple example of using this feature. A connection to a remote TCP port is created using the ztcp command; see the description of the **zsh/net/tcp** module in *zshmodules*(1). Then a handler is installed which simply prints out any data which arrives on this connection. Note that 'select' will indicate that the file descriptor needs handling if the remote side has closed the connection; we handle that by testing for a failed read.

```
if ztcp pwspc 2811; then
  tcpfd=$REPLY
  handler() {
    zle −I
    local line
    if ! read −r line <&$1; then
      # select marks this fd if we reach EOF,
      # so handle this specially.
      print "[Read on fd $1 failed, removing.]" >&2
      zle −F $1
      return 1
    fi
    print −r − $line
  }
  zle −F $tcpfd handler
fi
```

−I   Unusually, this option is most useful outside ordinary widget functions, though it may be used within if normal output to the terminal is required. It invalidates the current zle display in preparation for output; typically this will be from a trap function. It has no effect if zle is not active. When a trap exits, the shell checks to see if the display needs restoring, hence the following will print output in such a way as not to disturb the line being edited:

```
TRAPUSR1() {
  # Invalidate zle display
  [[ −o zle ]] && zle −I
  # Show output
  print Hello
}
```

In general, the trap function may need to test whether zle is active before using this method (as shown in the example), since the **zsh/zle** module may not even be loaded; if it is not, the command can be skipped.

It is possible to call '**zle −I**' several times before control is returned to the editor; the display will only be invalidated the first time to minimise disruption.

Note that there are normally better ways of manipulating the display from within zle widgets; see, for example, '**zle −R**' above.

The returned status is zero if zle was invalidated, even though this may have been by a previous call to '**zle −I**' or by a system notification. To test if a zle widget may be called at this point, execute **zle** with no arguments and examine the return status.

−T   This is used to add, list or remove internal transformations on the processing performed by the line editor. It is typically used only for debugging or testing and is therefore of little interest to the general user.

'**zle −T** *transformation func*' specifies that the given *transformation* (see below) is

effected by shell function *func*.

'**zle −Tr** *transformation*' removes the given *transformation* if it was present (it is not an error if none was).

'**zle −TL**' can be used to list all transformations currently in operation.

Currently the only transformation is **tc**. This is used instead of outputting termcap codes to the terminal. When the transformation is in operation the shell function is passed the termcap code that would be output as its first argument; if the operation required a numeric argument, that is passed as a second argument. The function should set the shell variable **REPLY** to the transformed termcap code. Typically this is used to produce some simply formatted version of the code and optional argument for debugging or testing. Note that this transformation is not applied to other non−printing characters such as carriage returns and newlines.

widget [**−n** *num*] [**−Nw**] [**−K** *keymap*] *args* ...

Invoke the specified *widget*. This can only be done when ZLE is active; normally this will be within a user−defined widget.

With the options **−n** and **−N**, the current numeric argument will be saved and then restored after the call to *widget*; '**−n** *num*' sets the numeric argument temporarily to *num*, while '**−N**' sets it to the default, i.e. as if there were none.

With the option **−K**, *keymap* will be used as the current keymap during the execution of the widget. The previous keymap will be restored when the widget exits.

Normally, calling a widget in this way does not set the special parameter **WIDGET** and related parameters, so that the environment appears as if the top−level widget called by the user were still active. With the option **−w**, **WIDGET** and related parameters are set to reflect the widget being executed by the **zle** call.

Any further arguments will be passed to the widget; note that as standard argument handling is performed, any general argument list should be preceded by **−−**. If it is a shell function, these are passed down as positional parameters; for builtin widgets it is up to the widget in question what it does with them. Currently arguments are only handled by the incremental−search commands, the **history−search−forward** and **−backward** and the corresponding functions prefixed by **vi−**, and by **universal−argument**. No error is flagged if the command does not use the arguments, or only uses some of them.

The return status reflects the success or failure of the operation carried out by the widget, or if it is a user−defined widget the return status of the shell function.

A non−zero return status causes the shell to beep when the widget exits, unless the **BEEP** options was unset or the widget was called via the **zle** command. Thus if a user defined widget requires an immediate beep, it should call the **beep** widget directly.

WIDGETS

All actions in the editor are performed by 'widgets'. A widget's job is simply to perform some small action. The ZLE commands that key sequences in keymaps are bound to are in fact widgets. Widgets can be user−defined or built in.

The standard widgets built into ZLE are listed in Standard Widgets below. Other built−in widgets can be defined by other modules (see *zshmodules*(1)). Each built−in widget has two names: its normal canonical name, and the same name preceded by a '**.**'. The '**.**' name is special: it can't be rebound to a different widget. This makes the widget available even when its usual name has been redefined.

User−defined widgets are defined using '**zle −N**', and implemented as shell functions. When the widget is executed, the corresponding shell function is executed, and can perform editing (or other) actions. It is recommended that user−defined widgets should not have names starting with '**.**'.

USER−DEFINED WIDGETS

User−defined widgets, being implemented as shell functions, can execute any normal shell command. They can also run other widgets (whether built−in or user−defined) using the **zle** builtin command. The standard input of the function is redirected from /dev/null to prevent external commands from unintentionally block-ing ZLE by reading from the terminal, but **read −k** or **read −q** can be used to read characters. Finally, they can examine and edit the ZLE buffer being edited by reading and setting the special parameters described below.

These special parameters are always available in widget functions, but are not in any way special outside ZLE. If they have some normal value outside ZLE, that value is temporarily inaccessible, but will return when the widget function exits. These special parameters in fact have local scope, like parameters created in a function using **local**.

Inside completion widgets and traps called while ZLE is active, these parameters are available read−only.

Note that the parameters appear as local to any ZLE widget in which they appear. Hence if it is desired to override them this needs to be done within a nested function:

> **widget−function() {**
> **# $WIDGET here refers to the special variable**
> **# that is local inside widget−function**
> **() {**
> **# This anonymous nested function allows WIDGET**
> **# to be used as a local variable. The −h**
> **# removes the special status of the variable.**
> **local −h WIDGET**
> **}**
> **}**

BUFFER (scalar)

> The entire contents of the edit buffer. If it is written to, the cursor remains at the same offset, un-less that would put it outside the buffer.

BUFFERLINES (integer)

> The number of screen lines needed for the edit buffer currently displayed on screen (i.e. without any changes to the preceding parameters done after the last redisplay); read−only.

CONTEXT (scalar)

> The context in which zle was called to read a line; read−only. One of the values:
>
> **start**   The start of a command line (at prompt **PS1**).
>
> **cont**   A continuation to a command line (at prompt **PS2**).
>
> **select** In a **select** loop (at prompt **PS3**).
>
> **vared**   Editing a variable in **vared**.

CURSOR (integer)

> The offset of the cursor, within the edit buffer. This is in the range 0 to **$#BUFFER**, and is by definition equal to **$#LBUFFER**. Attempts to move the cursor outside the buffer will result in the cursor being moved to the appropriate end of the buffer.

CUTBUFFER (scalar)

> The last item cut using one of the '**kill−**' commands; the string which the next yank would insert in the line. Later entries in the kill ring are in the array **killring**. Note that the command '**zle copy−region−as−kill** *string*' can be used to set the text of the cut buffer from a shell function and cycle the kill ring in the same way as interactively killing text.

HISTNO (integer)

> The current history number. Setting this has the same effect as moving up or down in the history to the corresponding history line. An attempt to set it is ignored if the line is not stored in the his-tory. Note this is not the same as the parameter **HISTCMD**, which always gives the number of

the history line being added to the main shell's history. **HISTNO** refers to the line being retrieved within zle.

ISEARCHMATCH_ACTIVE (integer)
ISEARCHMATCH_START (integer)
ISEARCHMATCH_END (integer)

> **ISEARCHMATCH_ACTIVE** indicates whether a part of the **BUFFER** is currently matched by an incremental search pattern. **ISEARCHMATCH_START** and **ISEARCHMATCH_END** give the location of the matched part and are in the same units as **CURSOR**. They are only valid for reading when **ISEARCHMATCH_ACTIVE** is non−zero.

> All parameters are read−only.

KEYMAP (scalar)

> The name of the currently selected keymap; read−only.

KEYS (scalar)

> The keys typed to invoke this widget, as a literal string; read−only.

KEYS_QUEUED_COUNT (integer)

> The number of bytes pushed back to the input queue and therefore available for reading immediately before any I/O is done; read−only. See also **PENDING**; the two values are distinct.

killring (array)

> The array of previously killed items, with the most recently killed first. This gives the items that would be retrieved by a **yank−pop** in the same order. Note, however, that the most recently killed item is in **$CUTBUFFER**; **$killring** shows the array of previous entries.

> The default size for the kill ring is eight, however the length may be changed by normal array operations. Any empty string in the kill ring is ignored by the **yank−pop** command, hence the size of the array effectively sets the maximum length of the kill ring, while the number of non−zero strings gives the current length, both as seen by the user at the command line.

LASTABORTEDSEARCH (scalar)

> The last search string used by an interactive search that was aborted by the user (status 3 returned by the search widget).

LASTSEARCH (scalar)

> The last search string used by an interactive search; read−only. This is set even if the search failed (status 0, 1 or 2 returned by the search widget), but not if it was aborted by the user.

LASTWIDGET (scalar)

> The name of the last widget that was executed; read−only.

LBUFFER (scalar)

> The part of the buffer that lies to the left of the cursor position. If it is assigned to, only that part of the buffer is replaced, and the cursor remains between the new **$LBUFFER** and the old **$RBUFFER**.

MARK (integer)

> Like **CURSOR**, but for the mark. With vi−mode operators that wait for a movement command to select a region of text, setting **MARK** allows the selection to extend in both directions from the initial cursor position.

NUMERIC (integer)

> The numeric argument. If no numeric argument was given, this parameter is unset. When this is set inside a widget function, builtin widgets called with the **zle** builtin command will use the value assigned. If it is unset inside a widget function, builtin widgets called behave as if no numeric argument was given.

PENDING (integer)

> The number of bytes pending for input, i.e. the number of bytes which have already been typed and can immediately be read. On systems where the shell is not able to get this information, this

parameter will always have a value of zero. Read−only. See also **KEYS_QUEUED_COUNT**;
the two values are distinct.

PREBUFFER (scalar)
>   In a multi−line input at the secondary prompt, this read−only parameter contains the contents of
>   the lines before the one the cursor is currently in.

PREDISPLAY (scalar)
>   Text to be displayed before the start of the editable text buffer. This does not have to be a com-
>   plete line; to display a complete line, a newline must be appended explicitly. The text is reset on
>   each new invocation (but not recursive invocation) of zle.

POSTDISPLAY (scalar)
>   Text to be displayed after the end of the editable text buffer. This does not have to be a complete
>   line; to display a complete line, a newline must be prepended explicitly. The text is reset on each
>   new invocation (but not recursive invocation) of zle.

RBUFFER (scalar)
>   The part of the buffer that lies to the right of the cursor position. If it is assigned to, only that part
>   of the buffer is replaced, and the cursor remains between the old **$LBUFFER** and the new
>   **$RBUFFER**.

REGION_ACTIVE (integer)
>   Indicates if the region is currently active. It can be assigned 0 or 1 to deactivate and activate the
>   region respectively. A value of 2 activates the region in line−wise mode with the highlighted text
>   extending for whole lines only; see *Character Highlighting* below.

region_highlight (array)
>   Each element of this array may be set to a string that describes highlighting for an arbitrary region
>   of the command line that will take effect the next time the command line is redisplayed. High-
>   lighting of the non−editable parts of the command line in **PREDISPLAY** and **POSTDISPLAY**
>   are possible, but note that the **P** flag is needed for character indexing to include **PREDISPLAY**.
>
>   Each string consists of the following parts:
>
>   • Optionally, a '**P**' to signify that the start and end offset that follow include any string set
>   by the **PREDISPLAY** special parameter; this is needed if the predisplay string itself is to
>   be highlighted. Whitespace may follow the '**P**'.
>
>   • A start offset in the same units as **CURSOR**, terminated by whitespace.
>
>   • An end offset in the same units as **CURSOR**, terminated by whitespace.
>
>   • A highlight specification in the same format as used for contexts in the parameter
>   **zle_highlight**, see the section 'Character Highlighting' below; for example, **standout** or
>   **fg=red,bold**
>
>   For example,
>
>   >   **region_highlight=("P0 20 bold")**
>
>   specifies that the first twenty characters of the text including any predisplay string should be high-
>   lighted in bold.
>
>   Note that the effect of **region_highlight** is not saved and disappears as soon as the line is accepted.
>
>   The final highlighting on the command line depends on both **region_highlight** and **zle_highlight**;
>   see the section CHARACTER HIGHLIGHTING below for details.

registers (associative array)
>   The contents of each of the vi register buffers. These are typically set using **vi−set−buffer** fol-
>   lowed by a delete, change or yank command.

SUFFIX_ACTIVE (integer)
SUFFIX_START (integer)
SUFFIX_END (integer)

> **SUFFIX_ACTIVE** indicates whether an auto−removable completion suffix is currently active. **SUFFIX_START** and **SUFFIX_END** give the location of the suffix and are in the same units as **CURSOR**. They are only valid for reading when **SUFFIX_ACTIVE** is non−zero.

> All parameters are read−only.

UNDO_CHANGE_NO (integer)

> A number representing the state of the undo history. The only use of this is passing as an argument to the **undo** widget in order to undo back to the recorded point. Read−only.

UNDO_LIMIT_NO (integer)

> A number corresponding to an existing change in the undo history; compare **UNDO_CHANGE_NO**. If this is set to a value greater than zero, the **undo** command will not allow the line to be undone beyond the given change number. It is still possible to use '**zle undo** *change*' in a widget to undo beyond that point; in that case, it will not be possible to undo at all until **UNDO_LIMIT_NO** is reduced. Set to 0 to disable the limit.

> A typical use of this variable in a widget function is as follows (note the additional function scope is required):

> > **() {**
> > ** local UNDO_LIMIT_NO=$UNDO_CHANGE_NO**
> > ** # Perform some form of recursive edit.**
> > **}**

WIDGET (scalar)

> The name of the widget currently being executed; read−only.

WIDGETFUNC (scalar)

> The name of the shell function that implements a widget defined with either **zle −N** or **zle −C**. In the former case, this is the second argument to the **zle −N** command that defined the widget, or the first argument if there was no second argument. In the latter case this is the third argument to the **zle −C** command that defined the widget. Read−only.

WIDGETSTYLE (scalar)

> Describes the implementation behind the completion widget currently being executed; the second argument that followed **zle −C** when the widget was defined. This is the name of a builtin completion widget. For widgets defined with **zle −N** this is set to the empty string. Read−only.

YANK_ACTIVE (integer)
YANK_START (integer)
YANK_END (integer)

> **YANK_ACTIVE** indicates whether text has just been yanked (pasted) into the buffer. **YANK_START** and **YANK_END** give the location of the pasted text and are in the same units as **CURSOR**. They are only valid for reading when **YANK_ACTIVE** is non−zero. They can also be assigned by widgets that insert text in a yank−like fashion, for example wrappers of **bracketed−paste**. See also **zle −f**.

> **YANK_ACTIVE** is read−only.

ZLE_RECURSIVE (integer)

> Usually zero, but incremented inside any instance of **recursive−edit**. Hence indicates the current recursion level.

> **ZLE_RECURSIVE** is read−only.

ZLE_STATE (scalar)

> Contains a set of space−separated words that describe the current **zle** state.

> Currently, the states shown are the insert mode as set by the **overwrite−mode** or **vi−replace**

widgets and whether history commands will visit imported entries as controlled by the set−local−history widget. The string contains '**insert**' if characters to be inserted on the command line move existing characters to the right or '**overwrite**' if characters to be inserted overwrite existing characters. It contains '**localhistory**' if only local history commands will be visited or '**globalhistory**' if imported history commands will also be visited.

The substrings are sorted in alphabetical order so that if you want to test for two specific substrings in a future−proof way, you can do match by doing:

> **if [[$ZLE_STATE == *globalhistory*insert*]]; then ...; fi**

Special Widgets

There are a few user−defined widgets which are special to the shell. If they do not exist, no special action is taken. The environment provided is identical to that for any other editing widget.

zle−isearch−exit

Executed at the end of incremental search at the point where the isearch prompt is removed from the display. See **zle−isearch−update** for an example.

zle−isearch−update

Executed within incremental search when the display is about to be redrawn. Additional output below the incremental search prompt can be generated by using '**zle −M**' within the widget. For example,

> **zle−isearch−update() { zle −M "Line $HISTNO"; }**
> **zle −N zle−isearch−update**

Note the line output by '**zle −M**' is not deleted on exit from incremental search. This can be done from a **zle−isearch−exit** widget:

> **zle−isearch−exit() { zle −M ""; }**
> **zle −N zle−isearch−exit**

zle−line−pre−redraw

Executed whenever the input line is about to be redrawn, providing an opportunity to update the region_highlight array.

zle−line−init

Executed every time the line editor is started to read a new line of input. The following example puts the line editor into vi command mode when it starts up.

> **zle−line−init() { zle −K vicmd; }**
> **zle −N zle−line−init**

(The command inside the function sets the keymap directly; it is equivalent to **zle vi−cmd−mode**.)

zle−line−finish

This is similar to **zle−line−init** but is executed every time the line editor has finished reading a line of input.

zle−history−line−set

Executed when the history line changes.

zle−keymap−select

Executed every time the keymap changes, i.e. the special parameter **KEYMAP** is set to a different value, while the line editor is active. Initialising the keymap when the line editor starts does not cause the widget to be called.

The value **$KEYMAP** within the function reflects the new keymap. The old keymap is passed as the sole argument.

This can be used for detecting switches between the vi command (**vicmd**) and insert (usually **main**) keymaps.

STANDARD WIDGETS

The following is a list of all the standard widgets, and their default bindings in emacs mode, vi command mode and vi insert mode (the '**emacs**', '**vicmd**' and '**viins**' keymaps, respectively).

Note that cursor keys are bound to movement keys in all three keymaps; the shell assumes that the cursor keys send the key sequences reported by the terminal−handling library (termcap or terminfo). The key sequences shown in the list are those based on the VT100, common on many modern terminals, but in fact these are not necessarily bound. In the case of the **viins** keymap, the initial escape character of the sequences serves also to return to the **vicmd** keymap: whether this happens is determined by the **KEYTIME-OUT** parameter, see *zshparam*(1).

Movement

vi−backward−blank−word (unbound) (**B**) (unbound)
>   Move backward one word, where a word is defined as a series of non−blank characters.

vi−backward−blank−word−end (unbound) (**gE**) (unbound)
>   Move to the end of the previous word, where a word is defined as a series of non−blank characters.

backward−char (**ˆB ESC−[D**) (unbound) (unbound)
>   Move backward one character.

vi−backward−char (unbound) (**ˆH h ˆ?**) (**ESC−[D**)
>   Move backward one character, without changing lines.

backward−word (**ESC−B ESC−b**) (unbound) (unbound)
>   Move to the beginning of the previous word.

emacs−backward−word
>   Move to the beginning of the previous word.

vi−backward−word (unbound) (**b**) (unbound)
>   Move to the beginning of the previous word, vi−style.

vi−backward−word−end (unbound) (**ge**) (unbound)
>   Move to the end of the previous word, vi−style.

beginning−of−line (**ˆA**) (unbound) (unbound)
>   Move to the beginning of the line. If already at the beginning of the line, move to the beginning of the previous line, if any.

vi−beginning−of−line
>   Move to the beginning of the line, without changing lines.

down−line (unbound) (unbound) (unbound)
>   Move down a line in the buffer.

end−of−line (**ˆE**) (unbound) (unbound)
>   Move to the end of the line. If already at the end of the line, move to the end of the next line, if any.

vi−end−of−line (unbound) (**$**) (unbound)
>   Move to the end of the line. If an argument is given to this command, the cursor will be moved to the end of the line (argument − 1) lines down.

vi−forward−blank−word (unbound) (**W**) (unbound)
>   Move forward one word, where a word is defined as a series of non−blank characters.

vi−forward−blank−word−end (unbound) (**E**) (unbound)
>   Move to the end of the current word, or, if at the end of the current word, to the end of the next word, where a word is defined as a series of non−blank characters.

forward−char (**ˆF ESC−[C**) (unbound) (unbound)
>   Move forward one character.

vi−forward−char (unbound) (**space l**) (**ESC−[C**)
>   Move forward one character.

vi−find−next−char (**ˆXˆF**) (**f**) (unbound)
>   Read a character from the keyboard, and move to the next occurrence of it in the line.

vi−find−next−char−skip (unbound) (**t**) (unbound)
>   Read a character from the keyboard, and move to the position just before the next occurrence of it
>   in the line.

vi−find−prev−char (unbound) (**F**) (unbound)
>   Read a character from the keyboard, and move to the previous occurrence of it in the line.

vi−find−prev−char−skip (unbound) (**T**) (unbound)
>   Read a character from the keyboard, and move to the position just after the previous occurrence of
>   it in the line.

vi−first−non−blank (unbound) (**ˆ**) (unbound)
>   Move to the first non−blank character in the line.

vi−forward−word (unbound) (**w**) (unbound)
>   Move forward one word, vi−style.

forward−word (**ESC−F ESC−f**) (unbound) (unbound)
>   Move to the beginning of the next word. The editor's idea of a word is specified with the **WORD-
>   CHARS** parameter.

emacs−forward−word
>   Move to the end of the next word.

vi−forward−word−end (unbound) (**e**) (unbound)
>   Move to the end of the next word.

vi−goto−column (**ESC−|**) (**|**) (unbound)
>   Move to the column specified by the numeric argument.

vi−goto−mark (unbound) (**'**) (unbound)
>   Move to the specified mark.

vi−goto−mark−line (unbound) (**'**) (unbound)
>   Move to beginning of the line containing the specified mark.

vi−repeat−find (unbound) (**;**) (unbound)
>   Repeat the last **vi−find** command.

vi−rev−repeat−find (unbound) (**,**) (unbound)
>   Repeat the last **vi−find** command in the opposite direction.

up−line (unbound) (unbound) (unbound)
>   Move up a line in the buffer.

History Control

beginning−of−buffer−or−history (**ESC−<**) (**gg**) (unbound)
>   Move to the beginning of the buffer, or if already there, move to the first event in the history list.

beginning−of−line−hist
>   Move to the beginning of the line. If already at the beginning of the buffer, move to the previous
>   history line.

beginning−of−history
>   Move to the first event in the history list.

down−line−or−history (**ˆN ESC−[B**) (**j**) (**ESC−[B**)
>   Move down a line in the buffer, or if already at the bottom line, move to the next event in the his-
>   tory list.

vi−down−line−or−history (unbound) (**+**) (unbound)
>   Move down a line in the buffer, or if already at the bottom line, move to the next event in the his−
>   tory list. Then move to the first non−blank character on the line.

down−line−or−search
>   Move down a line in the buffer, or if already at the bottom line, search forward in the history for a
>   line beginning with the first word in the buffer.
>
>   If called from a function by the **zle** command with arguments, the first argument is taken as the
>   string for which to search, rather than the first word in the buffer.

down−history (unbound) (**ˆN**) (unbound)
>   Move to the next event in the history list.

history−beginning−search−backward
>   Search backward in the history for a line beginning with the current line up to the cursor. This
>   leaves the cursor in its original position.

end−of−buffer−or−history (**ESC−>**) (unbound) (unbound)
>   Move to the end of the buffer, or if already there, move to the last event in the history list.

end−of−line−hist
>   Move to the end of the line. If already at the end of the buffer, move to the next history line.

end−of−history
>   Move to the last event in the history list.

vi−fetch−history (unbound) (**G**) (unbound)
>   Fetch the history line specified by the numeric argument. This defaults to the current history line
>   (i.e. the one that isn't history yet).

history−incremental−search−backward (**ˆR ˆXr**) (unbound) (unbound)
>   Search backward incrementally for a specified string. The search is case−insensitive if the search
>   string does not have uppercase letters and no numeric argument was given. The string may begin
>   with 'ˆ' to anchor the search to the beginning of the line. When called from a user−defined func−
>   tion returns the following statuses: 0, if the search succeeded; 1, if the search failed; 2, if the
>   search term was a bad pattern; 3, if the search was aborted by the **send−break** command.
>
>   A restricted set of editing functions is available in the mini−buffer. Keys are looked up in the spe−
>   cial **isearch** keymap, and if not found there in the main keymap (note that by default the **isearch**
>   keymap is empty). An interrupt signal, as defined by the stty setting, will stop the search and go
>   back to the original line. An undefined key will have the same effect. Note that the following al−
>   ways perform the same task within incremental searches and cannot be replaced by user defined
>   widgets, nor can the set of functions be extended. The supported functions are:
>
>   **accept−and−hold**
>   **accept−and−infer−next−history**
>   **accept−line**
>   **accept−line−and−down−history**
>>   Perform the usual function after exiting incremental search. The command line displayed
>>   is executed.
>
>   **backward−delete−char**
>   **vi−backward−delete−char**
>>   Back up one place in the search history. If the search has been repeated this does not im−
>>   mediately erase a character in the minibuffer.
>
>   **accept−search**
>>   Exit incremental search, retaining the command line but performing no further action.
>>   Note that this function is not bound by default and has no effect outside incremental
>>   search.

backward−delete−word
backward−kill−word
vi−backward−kill−word

> Back up one character in the minibuffer; if multiple searches have been performed since the character was inserted the search history is rewound to the point just before the character was entered. Hence this has the effect of repeating **backward−delete−char**.

clear−screen

> Clear the screen, remaining in incremental search mode.

history−incremental−search−backward

> Find the next occurrence of the contents of the mini−buffer. If the mini−buffer is empty, the most recent previously used search string is reinstated.

history−incremental−search−forward

> Invert the sense of the search.

magic−space

> Inserts a non−magical space.

quoted−insert
vi−quoted−insert

> Quote the character to insert into the minibuffer.

redisplay

> Redisplay the command line, remaining in incremental search mode.

vi−cmd−mode

> Select the '**vicmd**' keymap; the '**main**' keymap (insert mode) will be selected initially.
>
> In addition, the modifications that were made while in vi insert mode are merged to form a single undo event.

vi−repeat−search
vi−rev−repeat−search

> Repeat the search. The direction of the search is indicated in the mini−buffer.

Any character that is not bound to one of the above functions, or **self−insert** or **self−insert−un-meta**, will cause the mode to be exited. The character is then looked up and executed in the keymap in effect at that point.

When called from a widget function by the **zle** command, the incremental search commands can take a string argument. This will be treated as a string of keys, as for arguments to the **bindkey** command, and used as initial input for the command. Any characters in the string which are unused by the incremental search will be silently ignored. For example,

> **zle history−incremental−search−backward forceps**

will search backwards for **forceps**, leaving the minibuffer containing the string '**forceps**'.

history−incremental−search−forward (ˆS ˆXs) (unbound) (unbound)

> Search forward incrementally for a specified string. The search is case−insensitive if the search string does not have uppercase letters and no numeric argument was given. The string may begin with 'ˆ' to anchor the search to the beginning of the line. The functions available in the mini−buffer are the same as for **history−incremental−search−backward**.

history−incremental−pattern−search−backward
history−incremental−pattern−search−forward

> These widgets behave similarly to the corresponding widgets with no **−pattern**, but the search string typed by the user is treated as a pattern, respecting the current settings of the various options affecting pattern matching. See FILENAME GENERATION in *zshexpn*(1) for a description of patterns. If no numeric argument was given lowercase letters in the search string may match uppercase letters in the history. The string may begin with 'ˆ' to anchor the search to the beginning

of the line.

The prompt changes to indicate an invalid pattern; this may simply indicate the pattern is not yet complete.

Note that only non−overlapping matches are reported, so an expression with wildcards may return fewer matches on a line than are visible by inspection.

history−search−backward (**ESC−P ESC−p**) (unbound) (unbound)
Search backward in the history for a line beginning with the first word in the buffer.

If called from a function by the **zle** command with arguments, the first argument is taken as the string for which to search, rather than the first word in the buffer.

vi−history−search−backward (unbound) (**/**) (unbound)
Search backward in the history for a specified string. The string may begin with '^' to anchor the search to the beginning of the line.

A restricted set of editing functions is available in the mini−buffer. An interrupt signal, as defined by the stty setting, will stop the search. The functions available in the mini−buffer are: **accept−line**, **backward−delete−char**, **vi−backward−delete−char**, **backward−kill−word**, **vi−backward−kill−word**, **clear−screen**, **redisplay**, **quoted−insert** and **vi−quoted−insert**.

vi−cmd−mode is treated the same as accept−line, and **magic−space** is treated as a space. Any other character that is not bound to self−insert or self−insert−unmeta will beep and be ignored. If the function is called from vi command mode, the bindings of the current insert mode will be used.

If called from a function by the **zle** command with arguments, the first argument is taken as the string for which to search, rather than the first word in the buffer.

history−search−forward (**ESC−N ESC−n**) (unbound) (unbound)
Search forward in the history for a line beginning with the first word in the buffer.

If called from a function by the **zle** command with arguments, the first argument is taken as the string for which to search, rather than the first word in the buffer.

vi−history−search−forward (unbound) (**?**) (unbound)
Search forward in the history for a specified string. The string may begin with '^' to anchor the search to the beginning of the line. The functions available in the mini−buffer are the same as for **vi−history−search−backward**. Argument handling is also the same as for that command.

infer−next−history (**^X^N**) (unbound) (unbound)
Search in the history list for a line matching the current one and fetch the event following it.

insert−last−word (**ESC−_ ESC−.**) (unbound) (unbound)
Insert the last word from the previous history event at the cursor position. If a positive numeric argument is given, insert that word from the end of the previous history event. If the argument is zero or negative insert that word from the left (zero inserts the previous command word). Repeating this command replaces the word just inserted with the last word from the history event prior to the one just used; numeric arguments can be used in the same way to pick a word from that event.

When called from a shell function invoked from a user−defined widget, the command can take one to three arguments. The first argument specifies a history offset which applies to successive calls to this widget: if it is −1, the default behaviour is used, while if it is 1, successive calls will move forwards through the history. The value 0 can be used to indicate that the history line examined by the previous execution of the command will be reexamined. Note that negative numbers should be preceded by a '−−' argument to avoid confusing them with options.

If two arguments are given, the second specifies the word on the command line in normal array index notation (as a more natural alternative to the numeric argument). Hence 1 is the first word, and −1 (the default) is the last word.

If a third argument is given, its value is ignored, but it is used to signify that the history offset is relative to the current history line, rather than the one remembered after the previous invocations

of **insert−last−word**.

For example, the default behaviour of the command corresponds to

       zle insert−last−word −− −1 −1

while the command

       zle insert−last−word −− −1 1 −

always copies the first word of the line in the history immediately before the line being edited. This has the side effect that later invocations of the widget will be relative to that line.

vi−repeat−search (unbound) (**n**) (unbound)
> Repeat the last vi history search.

vi−rev−repeat−search (unbound) (**N**) (unbound)
> Repeat the last vi history search, but in reverse.

up−line−or−history (**^P ESC−[A**) (**k**) (**ESC−[A**)
> Move up a line in the buffer, or if already at the top line, move to the previous event in the history list.

vi−up−line−or−history (unbound) (**−**) (unbound)
> Move up a line in the buffer, or if already at the top line, move to the previous event in the history list. Then move to the first non−blank character on the line.

up−line−or−search
> Move up a line in the buffer, or if already at the top line, search backward in the history for a line beginning with the first word in the buffer.
>
> If called from a function by the **zle** command with arguments, the first argument is taken as the string for which to search, rather than the first word in the buffer.

up−history (unbound) (**^P**) (unbound)
> Move to the previous event in the history list.

history−beginning−search−forward
> Search forward in the history for a line beginning with the current line up to the cursor. This leaves the cursor in its original position.

set−local−history
> By default, history movement commands visit the imported lines as well as the local lines. This widget lets you toggle this on and off, or set it with the numeric argument. Zero for both local and imported lines and nonzero for only local lines.

Modifying Text

vi−add−eol (unbound) (**A**) (unbound)
> Move to the end of the line and enter insert mode.

vi−add−next (unbound) (**a**) (unbound)
> Enter insert mode after the current cursor position, without changing lines.

backward−delete−char (**^H ^?**) (unbound) (unbound)
> Delete the character behind the cursor.

vi−backward−delete−char (unbound) (**X**) (**^H**)
> Delete the character behind the cursor, without changing lines. If in insert mode, this won't delete past the point where insert mode was last entered.

backward−delete−word
> Delete the word behind the cursor.

backward−kill−line
> Kill from the beginning of the line to the cursor position.

backward−kill−word (**ˆW ESC−ˆH ESC−ˆ?**) (unbound) (unbound)
>   Kill the word behind the cursor.

vi−backward−kill−word (unbound) (unbound) (**ˆW**)
>   Kill the word behind the cursor, without going past the point where insert mode was last entered.

capitalize−word (**ESC−C ESC−c**) (unbound) (unbound)
>   Capitalize the current word and move past it.

vi−change (unbound) (**c**) (unbound)
>   Read a movement command from the keyboard, and kill from the cursor position to the endpoint
>   of the movement. Then enter insert mode. If the command is **vi−change**, change the current line.
>
>   For compatibility with vi, if the command is **vi−forward−word** or **vi−forward−blank−word**, the
>   whitespace after the word is not included. If you prefer the more consistent behaviour with the
>   whitespace included use the following key binding:
>
>>   **bindkey −a −s cw dwi**

vi−change−eol (unbound) (**C**) (unbound)
>   Kill to the end of the line and enter insert mode.

vi−change−whole−line (unbound) (**S**) (unbound)
>   Kill the current line and enter insert mode.

copy−region−as−kill (**ESC−W ESC−w**) (unbound) (unbound)
>   Copy the area from the cursor to the mark to the kill buffer.
>
>   If called from a ZLE widget function in the form '**zle copy−region−as−kill** *string*' then *string* will
>   be taken as the text to copy to the kill buffer. The cursor, the mark and the text on the command
>   line are not used in this case.

copy−prev−word (**ESC−ˆ_**) (unbound) (unbound)
>   Duplicate the word to the left of the cursor.

copy−prev−shell−word
>   Like **copy−prev−word**, but the word is found by using shell parsing, whereas **copy−prev−word**
>   looks for blanks. This makes a difference when the word is quoted and contains spaces.

vi−delete (unbound) (**d**) (unbound)
>   Read a movement command from the keyboard, and kill from the cursor position to the endpoint
>   of the movement. If the command is **vi−delete**, kill the current line.

delete−char
>   Delete the character under the cursor.

vi−delete−char (unbound) (**x**) (unbound)
>   Delete the character under the cursor, without going past the end of the line.

delete−word
>   Delete the current word.

down−case−word (**ESC−L ESC−l**) (unbound) (unbound)
>   Convert the current word to all lowercase and move past it.

vi−down−case (unbound) (**gu**) (unbound)
>   Read a movement command from the keyboard, and convert all characters from the cursor position
>   to the endpoint of the movement to lowercase. If the movement command is **vi−down−case**, swap
>   the case of all characters on the current line.

kill−word (**ESC−D ESC−d**) (unbound) (unbound)
>   Kill the current word.

gosmacs−transpose−chars
>   Exchange the two characters behind the cursor.

vi−indent (unbound) (**>**) (unbound)
>   Indent a number of lines.

vi−insert (unbound) (**i**) (unbound)
>   Enter insert mode.

vi−insert−bol (unbound) (**I**) (unbound)
>   Move to the first non−blank character on the line and enter insert mode.

vi−join (**^X^J**) (**J**) (unbound)
>   Join the current line with the next one.

kill−line (**^K**) (unbound) (unbound)
>   Kill from the cursor to the end of the line. If already on the end of the line, kill the newline character.

vi−kill−line (unbound) (unbound) (**^U**)
>   Kill from the cursor back to wherever insert mode was last entered.

vi−kill−eol (unbound) (**D**) (unbound)
>   Kill from the cursor to the end of the line.

kill−region
>   Kill from the cursor to the mark.

kill−buffer (**^X^K**) (unbound) (unbound)
>   Kill the entire buffer.

kill−whole−line (**^U**) (unbound) (unbound)
>   Kill the current line.

vi−match−bracket (**^X^B**) (**%**) (unbound)
>   Move to the bracket character (one of **{}, ()** or **[]**) that matches the one under the cursor. If the cursor is not on a bracket character, move forward without going past the end of the line to find one, and then go to the matching bracket.

vi−open−line−above (unbound) (**O**) (unbound)
>   Open a line above the cursor and enter insert mode.

vi−open−line−below (unbound) (**o**) (unbound)
>   Open a line below the cursor and enter insert mode.

vi−oper−swap−case (unbound) (**g˜**) (unbound)
>   Read a movement command from the keyboard, and swap the case of all characters from the cursor position to the endpoint of the movement. If the movement command is **vi−oper−swap−case**, swap the case of all characters on the current line.

overwrite−mode (**^X^O**) (unbound) (unbound)
>   Toggle between overwrite mode and insert mode.

vi−put−before (unbound) (**P**) (unbound)
>   Insert the contents of the kill buffer before the cursor. If the kill buffer contains a sequence of lines (as opposed to characters), paste it above the current line.

vi−put−after (unbound) (**p**) (unbound)
>   Insert the contents of the kill buffer after the cursor. If the kill buffer contains a sequence of lines (as opposed to characters), paste it below the current line.

put−replace−selection (unbound) (unbound) (unbound)
>   Replace the contents of the current region or selection with the contents of the kill buffer. If the kill buffer contains a sequence of lines (as opposed to characters), the current line will be split by the pasted lines.

quoted−insert (**ˆV**) (unbound) (unbound)
>   Insert the next character typed into the buffer literally. An interrupt character will not be inserted.

vi−quoted−insert (unbound) (unbound) (**ˆQ ˆV**)
>   Display a 'ˆ' at the cursor position, and insert the next character typed into the buffer literally. An interrupt character will not be inserted.

quote−line (**ESC−'**) (unbound) (unbound)
>   Quote the current line; that is, put a '' character at the beginning and the end, and convert all '' characters to '**\'**'.

quote−region (**ESC−"**) (unbound) (unbound)
>   Quote the region from the cursor to the mark.

vi−replace (unbound) (**R**) (unbound)
>   Enter overwrite mode.

vi−repeat−change (unbound) (**.**) (unbound)
>   Repeat the last vi mode text modification. If a count was used with the modification, it is remembered. If a count is given to this command, it overrides the remembered count, and is remembered for future uses of this command. The cut buffer specification is similarly remembered.

vi−replace−chars (unbound) (**r**) (unbound)
>   Replace the character under the cursor with a character read from the keyboard.

self−insert (printable characters) (unbound) (printable characters and some control characters)
>   Insert a character into the buffer at the cursor position.

self−insert−unmeta (**ESC−ˆI ESC−ˆJ ESC−ˆM**) (unbound) (unbound)
>   Insert a character into the buffer after stripping the meta bit and converting ˆM to ˆJ.

vi−substitute (unbound) (**s**) (unbound)
>   Substitute the next character(s).

vi−swap−case (unbound) (**˜**) (unbound)
>   Swap the case of the character under the cursor and move past it.

transpose−chars (**ˆT**) (unbound) (unbound)
>   Exchange the two characters to the left of the cursor if at end of line, else exchange the character under the cursor with the character to the left.

transpose−words (**ESC−T ESC−t**) (unbound) (unbound)
>   Exchange the current word with the one before it.
>
>   With a positive numeric argument N, the word around the cursor, or following it if the cursor is between words, is transposed with the preceding N words. The cursor is put at the end of the resulting group of words.
>
>   With a negative numeric argument $−N$, the effect is the same as using a positive argument N except that the original cursor position is retained, regardless of how the words are rearranged.

vi−unindent (unbound) (**<**) (unbound)
>   Unindent a number of lines.

vi−up−case (unbound) (**gU**) (unbound)
>   Read a movement command from the keyboard, and convert all characters from the cursor position to the endpoint of the movement to lowercase. If the movement command is **vi−up−case**, swap the case of all characters on the current line.

up−case−word (**ESC−U ESC−u**) (unbound) (unbound)
>   Convert the current word to all caps and move past it.

yank (**ˆY**) (unbound) (unbound)
>   Insert the contents of the kill buffer at the cursor position.

yank−pop (**ESC−y**) (unbound) (unbound)
>   Remove the text just yanked, rotate the kill−ring (the history of previously killed text) and yank
>   the new top. Only works following **yank**, **vi−put−before**, **vi−put−after** or **yank−pop**.

vi−yank (unbound) (**y**) (unbound)
>   Read a movement command from the keyboard, and copy the region from the cursor position to
>   the endpoint of the movement into the kill buffer. If the command is **vi−yank**, copy the current
>   line.

vi−yank−whole−line (unbound) (**Y**) (unbound)
>   Copy the current line into the kill buffer.

vi−yank−eol
>   Copy the region from the cursor position to the end of the line into the kill buffer. Arguably, this
>   is what Y should do in vi, but it isn't what it actually does.

Arguments
digit−argument (**ESC−0**..**ESC−9**) (**1−9**) (unbound)
>   Start a new numeric argument, or add to the current one. See also **vi−digit−or−begin-**
>   **ning−of−line**. This only works if bound to a key sequence ending in a decimal digit.

>   Inside a widget function, a call to this function treats the last key of the key sequence which called
>   the widget as the digit.

neg−argument (**ESC−−**) (unbound) (unbound)
>   Changes the sign of the following argument.

universal−argument
>   Multiply the argument of the next command by 4. Alternatively, if this command is followed by
>   an integer (positive or negative), use that as the argument for the next command. Thus digits can-
>   not be repeated using this command. For example, if this command occurs twice, followed imme-
>   diately by **forward−char**, move forward sixteen spaces; if instead it is followed by **−2**, then **for-**
>   **ward−char**, move backward two spaces.

>   Inside a widget function, if passed an argument, i.e. '**zle universal−argument** *num*', the numeric
>   argument will be set to *num*; this is equivalent to '**NUMERIC=***num*'.

argument−base
>   Use the existing numeric argument as a numeric base, which must be in the range 2 to 36 inclu-
>   sive. Subsequent use of **digit−argument** and **universal−argument** will input a new numeric ar-
>   gument in the given base. The usual hexadecimal convention is used: the letter **a** or **A** corresponds
>   to 10, and so on. Arguments in bases requiring digits from 10 upwards are more conveniently in-
>   put with **universal−argument**, since **ESC−a** etc. are not usually bound to **digit−argument**.

>   The function can be used with a command argument inside a user−defined widget. The following
>   code sets the base to 16 and lets the user input a hexadecimal argument until a key out of the digit
>   range is typed:

>>   **zle argument−base 16**
>>   **zle universal−argument**

Completion
accept−and−menu−complete
>   In a menu completion, insert the current completion into the buffer, and advance to the next possi-
>   ble completion.

complete−word
>   Attempt completion on the current word.

delete−char−or−list (**^D**) (unbound) (unbound)
>   Delete the character under the cursor. If the cursor is at the end of the line, list possible comple-
>   tions for the current word.

expand−cmd−path
> Expand the current command to its full pathname.

expand−or−complete (**TAB**) (unbound) (**TAB**)
> Attempt shell expansion on the current word. If that fails, attempt completion.

expand−or−complete−prefix
> Attempt shell expansion on the current word up to cursor.

expand−history (**ESC−space ESC−!**) (unbound) (unbound)
> Perform history expansion on the edit buffer.

expand−word (**^X***) (unbound) (unbound)
> Attempt shell expansion on the current word.

list−choices (**ESC−^D**) (**^D =**) (**^D**)
> List possible completions for the current word.

list−expand (**^Xg ^XG**) (**^G**) (**^G**)
> List the expansion of the current word.

magic−space
> Perform history expansion and insert a space into the buffer. This is intended to be bound to space.

menu−complete
> Like **complete−word**, except that menu completion is used. See the **MENU_COMPLETE** option.

menu−expand−or−complete
> Like **expand−or−complete**, except that menu completion is used.

reverse−menu−complete
> Perform menu completion, like **menu−complete**, except that if a menu completion is already in progress, move to the *previous* completion rather than the next.

end−of−list
> When a previous completion displayed a list below the prompt, this widget can be used to move the prompt below the list.

Miscellaneous

accept−and−hold (**ESC−A ESC−a**) (unbound) (unbound)
> Push the contents of the buffer on the buffer stack and execute it.

accept−and−infer−next−history
> Execute the contents of the buffer. Then search the history list for a line matching the current one and push the event following onto the buffer stack.

accept−line (**^J ^M**) (**^J ^M**) (**^J ^M**)
> Finish editing the buffer. Normally this causes the buffer to be executed as a shell command.

accept−line−and−down−history (**^O**) (unbound) (unbound)
> Execute the current line, and push the next history event on the buffer stack.

auto−suffix−remove
> If the previous action added a suffix (space, slash, etc.) to the word on the command line, remove it. Otherwise do nothing. Removing the suffix ends any active menu completion or menu selection.
>
> This widget is intended to be called from user−defined widgets to enforce a desired suffix−removal behavior.

auto−suffix−retain
> If the previous action added a suffix (space, slash, etc.) to the word on the command line, force it to be preserved. Otherwise do nothing. Retaining the suffix ends any active menu completion or

menu selection.

This widget is intended to be called from user−defined widgets to enforce a desired suffix−preservation behavior.

beep   Beep, unless the **BEEP** option is unset.

bracketed−paste

This widget is invoked when text is pasted to the terminal emulator. It is not intended to be bound to actual keys but instead to the special sequence generated by the terminal emulator when text is pasted.

When invoked interactively, the pasted text is inserted to the buffer and placed in the cutbuffer. If a numeric argument is given, shell quoting will be applied to the pasted text before it is inserted.

When a named buffer is specified with **vi−set−buffer** (**"x**), the pasted text is stored in that named buffer but not inserted.

When called from a widget function as '**bracketed−paste** *name*', the pasted text is assigned to the variable *name* and no other processing is done.

See also the **zle_bracketed_paste** parameter.

vi−cmd−mode (**ˆXˆV**) (unbound) (**ˆ[**)

Enter command mode; that is, select the '**vicmd**' keymap. Yes, this is bound by default in emacs mode.

vi−caps−lock−panic

Hang until any lowercase key is pressed. This is for vi users without the mental capacity to keep track of their caps lock key (like the author).

clear−screen (**ˆL ESC−ˆL**) (**ˆL**) (**ˆL**)

Clear the screen and redraw the prompt.

deactivate−region

Make the current region inactive. This disables vim−style visual selection mode if it is active.

describe−key−briefly

Reads a key sequence, then prints the function bound to that sequence.

exchange−point−and−mark (**ˆXˆX**) (unbound) (unbound)

Exchange the cursor position (point) with the position of the mark. Unless a negative numeric argument is given, the region between point and mark is activated so that it can be highlighted. If a zero numeric argument is given, the region is activated but point and mark are not swapped.

execute−named−cmd (**ESC−x**) (**:**) (unbound)

Read the name of an editor command and execute it. Aliasing this widget with '**zle −A**' or replacing it with '**zle −N**' has no effect when interpreting key bindings, but '**zle execute−named−cmd**' will invoke such an alias or replacement.

A restricted set of editing functions is available in the mini−buffer. Keys are looked up in the special **command** keymap, and if not found there in the main keymap. An interrupt signal, as defined by the stty setting, will abort the function. Note that the following always perform the same task within the **executed−named−cmd** environment and cannot be replaced by user defined widgets, nor can the set of functions be extended. The allowed functions are: **backward−delete−char**, **vi−backward−delete−char**, **clear−screen**, **redisplay**, **quoted−insert**, **vi−quoted−insert**, **backward−kill−word**, **vi−backward−kill−word**, **kill−whole−line**, **vi−kill−line**, **backward−kill−line**, **list−choices**, **delete−char−or−list**, **complete−word**, **accept−line**, **expand−or−complete** and **expand−or−complete−prefix**.

kill−region kills the last word, and vi−cmd−mode is treated the same as accept−line. The space and tab characters, if not bound to one of these functions, will complete the name and then list the possibilities if the **AUTO_LIST** option is set. Any other character that is not bound to **self−insert** or **self−insert−unmeta** will beep and be ignored. The bindings of the current insert mode will be

used.

Currently this command may not be redefined or called by name.

execute−last−named−cmd (**ESC−z**) (unbound) (unbound)
> Redo the last function executed with **execute−named−cmd**.

> Like **execute−named−cmd**, this command may not be redefined, but it may be called by name.

get−line (**ESC−G ESC−g**) (unbound) (unbound)
> Pop the top line off the buffer stack and insert it at the cursor position.

pound−insert (unbound) (**#**) (unbound)
> If there is no # character at the beginning of the buffer, add one to the beginning of each line. If there is one, remove a # from each line that has one. In either case, accept the current line. The **INTERACTIVE_COMMENTS** option must be set for this to have any usefulness.

vi−pound−insert
> If there is no # character at the beginning of the current line, add one. If there is one, remove it. The **INTERACTIVE_COMMENTS** option must be set for this to have any usefulness.

push−input
> Push the entire current multiline construct onto the buffer stack and return to the top−level (**PS1**) prompt. If the current parser construct is only a single line, this is exactly like **push−line**. Next time the editor starts up or is popped with **get−line**, the construct will be popped off the top of the buffer stack and loaded into the editing buffer.

push−line (**^Q ESC−Q ESC−q**) (unbound) (unbound)
> Push the current buffer onto the buffer stack and clear the buffer. Next time the editor starts up, the buffer will be popped off the top of the buffer stack and loaded into the editing buffer.

push−line−or−edit
> At the top−level (**PS1**) prompt, equivalent to **push−line**. At a secondary (**PS2**) prompt, move the entire current multiline construct into the editor buffer. The latter is equivalent to **push−input** followed by **get−line**.

read−command
> Only useful from a user−defined widget. A keystroke is read just as in normal operation, but instead of the command being executed the name of the command that would be executed is stored in the shell parameter **REPLY**. This can be used as the argument of a future **zle** command. If the key sequence is not bound, status 1 is returned; typically, however, **REPLY** is set to **undefined−key** to indicate a useless key sequence.

recursive−edit
> Only useful from a user−defined widget. At this point in the function, the editor regains control until one of the standard widgets which would normally cause zle to exit (typically an **accept−line** caused by hitting the return key) is executed. Instead, control returns to the user−defined widget. The status returned is non−zero if the return was caused by an error, but the function still continues executing and hence may tidy up. This makes it safe for the user−defined widget to alter the command line or key bindings temporarily.

> The following widget, **caps−lock**, serves as an example.

> > **self−insert−ucase() {**
> > **LBUFFER+=${(U)KEYS[−1]}**
> > **}**

> > **integer stat**

> > **zle −N self−insert self−insert−ucase**
> > **zle −A caps−lock save−caps−lock**
> > **zle −A accept−line caps−lock**

        zle recursive−edit
        stat=$?

        zle −A .self−insert self−insert
        zle −A save−caps−lock caps−lock
        zle −D save−caps−lock

        ((stat)) && zle send−break

        return $stat

This causes typed letters to be inserted capitalised until either **accept−line** (i.e. typically the return key) is typed or the **caps−lock** widget is invoked again; the later is handled by saving the old definition of **caps−lock** as **save−caps−lock** and then rebinding it to invoke **accept−line**. Note that an error from the recursive edit is detected as a non−zero return status and propagated by using the **send−break** widget.

redisplay (unbound) (**^R**) (**^R**)
Redisplays the edit buffer.

reset−prompt (unbound) (unbound) (unbound)
Force the prompts on both the left and right of the screen to be re−expanded, then redisplay the edit buffer. This reflects changes both to the prompt variables themselves and changes in the expansion of the values (for example, changes in time or directory, or changes to the value of variables referred to by the prompt).

Otherwise, the prompt is only expanded each time zle starts, and when the display has been interrupted by output from another part of the shell (such as a job notification) which causes the command line to be reprinted.

reset−prompt doesn't alter the special parameter **LASTWIDGET**.

send−break (**^G ESC−^G**) (unbound) (unbound)
Abort the current editor function, e.g. **execute−named−command**, or the editor itself, e.g. if you are in **vared**. Otherwise abort the parsing of the current line; in this case the aborted line is available in the shell variable **ZLE_LINE_ABORTED**. If the editor is aborted from within **vared**, the variable **ZLE_VARED_ABORTED** is set.

run−help (**ESC−H ESC−h**) (unbound) (unbound)
Push the buffer onto the buffer stack, and execute the command '**run−help** *cmd*', where *cmd* is the current command. **run−help** is normally aliased to **man**.

vi−set−buffer (unbound) (**"**) (unbound)
Specify a buffer to be used in the following command. There are 37 buffers that can be specified: the 26 'named' buffers **"a** to **"z**, the 'yank' buffer **"0**, the nine 'queued' buffers **"1** to **"9** and the 'black hole' buffer **"_**. The named buffers can also be specified as **"A** to **"Z**.

When a buffer is specified for a cut, change or yank command, the text concerned replaces the previous contents of the specified buffer. If a named buffer is specified using a capital, the newly cut text is appended to the buffer instead of overwriting it. When using the **"_** buffer, nothing happens. This can be useful for deleting text without affecting any buffers.

If no buffer is specified for a cut or change command, **"1** is used, and the contents of **"1** to **"8** are each shifted along one buffer; the contents of **"9** is lost. If no buffer is specified for a yank command, **"0** is used. Finally, a paste command without a specified buffer will paste the text from the most recent command regardless of any buffer that might have been used with that command.

When called from a widget function by the **zle** command, the buffer can optionally be specified with an argument. For example,

        zle vi−set−buffer A

vi−set−mark (unbound) (**m**) (unbound)
>   Set the specified mark at the cursor position.

set−mark−command (**^@**) (unbound) (unbound)
>   Set the mark at the cursor position. If called with a negative numeric argument, do not set the
>   mark but deactivate the region so that it is no longer highlighted (it is still usable for other pur-
>   poses). Otherwise the region is marked as active.

spell−word (**ESC−$ ESC−S ESC−s**) (unbound) (unbound)
>   Attempt spelling correction on the current word.

split−undo
>   Breaks the undo sequence at the current change. This is useful in vi mode as changes made in in-
>   sert mode are coalesced on entering command mode. Similarly, **undo** will normally revert as one
>   all the changes made by a user−defined widget.

undefined−key
>   This command is executed when a key sequence that is not bound to any command is typed. By
>   default it beeps.

undo (**^_ ^Xu ^X^U**) (**u**) (unbound)
>   Incrementally undo the last text modification. When called from a user−defined widget, takes an
>   optional argument indicating a previous state of the undo history as returned by the
>   **UNDO_CHANGE_NO** variable; modifications are undone until that state is reached, subject to
>   any limit imposed by the **UNDO_LIMIT_NO** variable.
>
>   Note that when invoked from vi command mode, the full prior change made in insert mode is re-
>   verted, the changes having been merged when command mode was selected.

redo (unbound) (**^R**) (unbound)
>   Incrementally redo undone text modifications.

vi−undo−change (unbound) (unbound) (unbound)
>   Undo the last text modification. If repeated, redo the modification.

visual−mode (unbound) (**v**) (unbound)
>   Toggle vim−style visual selection mode. If line−wise visual mode is currently enabled then it is
>   changed to being character−wise. If used following an operator, it forces the subsequent movement
>   command to be treated as a character−wise movement.

visual−line−mode (unbound) (**V**) (unbound)
>   Toggle vim−style line−wise visual selection mode. If character−wise visual mode is currently en-
>   abled then it is changed to being line−wise. If used following an operator, it forces the subsequent
>   movement command to be treated as a line−wise movement.

what−cursor−position (**^X=**) (**ga**) (unbound)
>   Print the character under the cursor, its code as an octal, decimal and hexadecimal number, the cur-
>   rent cursor position within the buffer and the column of the cursor in the current line.

where−is
>   Read the name of an editor command and print the listing of key sequences that invoke the speci-
>   fied command. A restricted set of editing functions is available in the mini−buffer. Keys are
>   looked up in the special **command** keymap, and if not found there in the main keymap.

which−command (**ESC−?**) (unbound) (unbound)
>   Push the buffer onto the buffer stack, and execute the command '**which−command** *cmd*'. where
>   *cmd* is the current command. **which−command** is normally aliased to **whence**.

vi−digit−or−beginning−of−line (unbound) (**0**) (unbound)
>   If the last command executed was a digit as part of an argument, continue the argument. Other-
>   wise, execute vi−beginning−of−line.

Text Objects

Text objects are commands that can be used to select a block of text according to some criteria. They are a feature of the vim text editor and so are primarily intended for use with vi operators or from visual selection mode. However, they can also be used from vi−insert or emacs mode. Key bindings listed below apply to the **viopp** and **visual** keymaps.

select−a−blank−word (**aW**)

Select a word including adjacent blanks, where a word is defined as a series of non−blank characters. With a numeric argument, multiple words will be selected.

select−a−shell−word (**aa**)

Select the current command argument applying the normal rules for quoting.

select−a−word (**aw**)

Select a word including adjacent blanks, using the normal vi−style word definition. With a numeric argument, multiple words will be selected.

select−in−blank−word (**iW**)

Select a word, where a word is defined as a series of non−blank characters. With a numeric argument, multiple words will be selected.

select−in−shell−word (**ia**)

Select the current command argument applying the normal rules for quoting. If the argument begins and ends with matching quote characters, these are not included in the selection.

select−in−word (**iw**)

Select a word, using the normal vi−style word definition. With a numeric argument, multiple words will be selected.

CHARACTER HIGHLIGHTING

The line editor has the ability to highlight characters or regions of the line that have a particular significance. This is controlled by the array parameter **zle_highlight**, if it has been set by the user.

If the parameter contains the single entry **none** all highlighting is turned off. Note the parameter is still expected to be an array.

Otherwise each entry of the array should consist of a word indicating a context for highlighting, then a colon, then a comma−separated list of the types of highlighting to apply in that context.

The contexts available for highlighting are the following:

default Any text within the command line not affected by any other highlighting. Text outside the editable area of the command line is not affected.

isearch When one of the incremental history search widgets is active, the area of the command line matched by the search string or pattern.

region The currently selected text. In emacs terminology, this is referred to as the region and is bounded by the cursor (point) and the mark. The region is only highlighted if it is active, which is the case after the mark is modified with **set−mark−command** or **exchange−point−and−mark**. Note that whether or not the region is active has no effect on its use within emacs style widgets, it simply determines whether it is highlighted. In vi mode, the region corresponds to selected text in visual mode.

special Individual characters that have no direct printable representation but are shown in a special manner by the line editor. These characters are described below.

suffix This context is used in completion for characters that are marked as suffixes that will be removed if the completion ends at that point, the most obvious example being a slash (*/*) after a directory name. Note that suffix removal is configurable; the circumstances under which the suffix will be removed may differ for different completions.

paste Following a command to paste text, the characters that were inserted.

When **region_highlight** is set, the contexts that describe a region −− **isearch**, **region**, **suffix**, and **paste** −−

are applied first, then **region_highlight** is applied, then the remaining **zle_highlight** contexts are applied. If a particular character is affected by multiple specifications, the last specification wins.

zle_highlight may contain additional fields for controlling how terminal sequences to change colours are output. Each of the following is followed by a colon and a string in the same form as for key bindings. This will not be necessary for the vast majority of terminals as the defaults shown in parentheses are widely used.

fg_start_code (\e[3)
> The start of the escape sequence for the foreground colour. This is followed by one to three ASCII digits representing the colour. Only used for palette colors, i.e. not 24−bit colors specified via a color triplet.

fg_default_code (9)
> The number to use instead of the colour to reset the default foreground colour.

fg_end_code (m)
> The end of the escape sequence for the foreground colour.

bg_start_code (\e[4)
> The start of the escape sequence for the background colour. See **fg_start_code** above.

bg_default_code (9)
> The number to use instead of the colour to reset the default background colour.

bg_end_code (m)
> The end of the escape sequence for the background colour.

The available types of highlighting are the following. Note that not all types of highlighting are available on all terminals:

none   No highlighting is applied to the given context. It is not useful for this to appear with other types of highlighting; it is used to override a default.

fg=_colour_
> The foreground colour should be set to _colour_, a decimal integer, the name of one of the eight most widely−supported colours or as a '**#**' followed by an RGB triplet in hexadecimal format.
>
> Not all terminals support this and, of those that do, not all provide facilities to test the support, hence the user should decide based on the terminal type. Most terminals support the colours **black**, **red**, **green**, **yellow**, **blue**, **magenta**, **cyan** and **white**, which can be set by name. In addition. **default** may be used to set the terminal's default foreground colour. Abbreviations are allowed; **b** or **bl** selects black. Some terminals may generate additional colours if the **bold** attribute is also present.
>
> On recent terminals and on systems with an up−to−date terminal database the number of colours supported may be tested by the command '**echotc Co**'; if this succeeds, it indicates a limit on the number of colours which will be enforced by the line editor. The number of colours is in any case limited to 256 (i.e. the range 0 to 255).
>
> Some modern terminal emulators have support for 24−bit true colour (16 million colours). In this case, the hex triplet format can be used. This consists of a '**#**' followed by either a three or six digit hexadecimal number describing the red, green and blue components of the colour. Hex triplets can also be used with 88 and 256 colour terminals via the **zsh/nearcolor** module (see _zshmodules_(1)).
>
> Colour is also known as color.

bg=_colour_
> The background colour should be set to _colour_. This works similarly to the foreground colour, except the background is not usually affected by the bold attribute.

bold   The characters in the given context are shown in a bold font. Not all terminals distinguish bold fonts.

standout

The characters in the given context are shown in the terminal's standout mode. The actual effect is specific to the terminal; on many terminals it is inverse video. On some such terminals, where the cursor does not blink it appears with standout mode negated, making it less than clear where the cursor actually is. On such terminals one of the other effects may be preferable for highlighting the region and matched search string.

underline

The characters in the given context are shown underlined. Some terminals show the foreground in a different colour instead; in this case whitespace will not be highlighted.

The characters described above as 'special' are as follows. The formatting described here is used irrespective of whether the characters are highlighted:

ASCII control characters

Control characters in the ASCII range are shown as '^' followed by the base character.

Unprintable multibyte characters

This item applies to control characters not in the ASCII range, plus other characters as follows. If the **MULTIBYTE** option is in effect, multibyte characters not in the ASCII character set that are reported as having zero width are treated as combining characters when the option **COMBIN-ING_CHARS** is on. If the option is off, or if a character appears where a combining character is not valid, the character is treated as unprintable.

Unprintable multibyte characters are shown as a hexadecimal number between angle brackets. The number is the code point of the character in the wide character set; this may or may not be Unicode, depending on the operating system.

Invalid multibyte characters

If the **MULTIBYTE** option is in effect, any sequence of one or more bytes that does not form a valid character in the current character set is treated as a series of bytes each shown as a special character. This case can be distinguished from other unprintable characters as the bytes are represented as two hexadecimal digits between angle brackets, as distinct from the four or eight digits that are used for unprintable characters that are nonetheless valid in the current character set.

Not all systems support this: for it to work, the system's representation of wide characters must be code values from the Universal Character Set, as defined by IS0 10646 (also known as Unicode).

Wrapped double−width characters

When a double−width character appears in the final column of a line, it is instead shown on the next line. The empty space left in the original position is highlighted as a special character.

If **zle_highlight** is not set or no value applies to a particular context, the defaults applied are equivalent to

> **zle_highlight=(region:standout special:standout**
> **suffix:bold isearch:underline paste:standout)**

i.e. both the region and special characters are shown in standout mode.

Within widgets, arbitrary regions may be highlighted by setting the special array parameter **region_high-light**; see above.