NAME

zshcontrib – user contributions to zsh

DESCRIPTION

The Zsh source distribution includes a number of items contributed by the user community. These are not inherently a part of the shell, and some may not be available in every zsh installation. The most significant of these are documented here. For documentation on other contributed items such as shell functions, look for comments in the function source files.

UTILITIES

Accessing On–Line Help

The key sequence **ESC h** is normally bound by ZLE to execute the **run–help** widget (see *zshzle*(1)). This invokes the **run–help** command with the command word from the current input line as its argument. By default, **run–help** is an alias for the **man** command, so this often fails when the command word is a shell builtin or a user–defined function. By redefining the **run–help** alias, one can improve the on–line help provided by the shell.

The **helpfiles** utility, found in the **Util** directory of the distribution, is a Perl program that can be used to process the zsh manual to produce a separate help file for each shell builtin and for many other shell features as well. The autoloadable **run–help** function, found in **Functions/Misc**, searches for these helpfiles and performs several other tests to produce the most complete help possible for the command.

Help files are installed by default to a subdirectory of **/usr/share/zsh** or **/usr/local/share/zsh**.

To create your own help files with **helpfiles**, choose or create a directory where the individual command help files will reside. For example, you might choose **~/zsh_help**. If you unpacked the zsh distribution in your home directory, you would use the commands:

```
mkdir ~/zsh_help
perl ~/zsh-5.8/Util/helpfiles ~/zsh_help
```

The **HELPDIR** parameter tells **run–help** where to look for the help files. When unset, it uses the default installation path. To use your own set of help files, set this to the appropriate path in one of your startup files:

```
HELPDIR=~/zsh_help
```

To use the **run–help** function, you need to add lines something like the following to your **.zshrc** or equivalent startup file:

```
unalias run-help
autoload run-help
```

Note that in order for '**autoload run–help**' to work, the **run–help** file must be in one of the directories named in your **fpath** array (see *zshparam*(1)). This should already be the case if you have a standard zsh installation; if it is not, copy **Functions/Misc/run–help** to an appropriate directory.

Recompiling Functions

If you frequently edit your zsh functions, or periodically update your zsh installation to track the latest developments, you may find that function digests compiled with the **zcompile** builtin are frequently out of date with respect to the function source files. This is not usually a problem, because zsh always looks for the newest file when loading a function, but it may cause slower shell startup and function loading. Also, if a digest file is explicitly used as an element of **fpath**, zsh won't check whether any of its source files has changed.

The **zrecompile** autoloadable function, found in **Functions/Misc**, can be used to keep function digests up to date.

zrecompile [**–qt**] [*name* ...]
zrecompile [**–qt**] **–p** *arg* ... [**––** *arg* ...]

This tries to find ***.zwc** files and automatically re–compile them if at least one of the original files is newer than the compiled file. This works only if the names stored in the compiled files are full paths or are relative to the directory that contains the **.zwc** file.

In the first form, each *name* is the name of a compiled file or a directory containing ***.zwc** files that should be checked. If no arguments are given, the directories and ***.zwc** files in **fpath** are used.

When **−t** is given, no compilation is performed, but a return status of zero (true) is set if there are files that need to be re−compiled and non−zero (false) otherwise. The **−q** option quiets the chatty output that describes what **zrecompile** is doing.

Without the **−t** option, the return status is zero if all files that needed re−compilation could be compiled and non−zero if compilation for at least one of the files failed.

If the **−p** option is given, the *arg*s are interpreted as one or more sets of arguments for **zcompile**, separated by '**−−**'. For example:

```
zrecompile −p \
     −R ~/.zshrc −− \
     −M ~/.zcompdump −− \
     ~/zsh/comp.zwc ~/zsh/Completion/*/_*
```

This compiles **~/.zshrc** into **~/.zshrc.zwc** if that doesn't exist or if it is older than **~/.zshrc**. The compiled file will be marked for reading instead of mapping. The same is done for **~/.zcompdump** and **~/.zcompdump.zwc**, but this compiled file is marked for mapping. The last line re−creates the file **~/zsh/comp.zwc** if any of the files matching the given pattern is newer than it.

Without the **−p** option, **zrecompile** does not create function digests that do not already exist, nor does it add new functions to the digest.

The following shell loop is an example of a method for creating function digests for all functions in your **fpath**, assuming that you have write permission to the directories:

```
for ((i=1; i <= $#fpath; ++i)); do
  dir=$fpath[i]
  zwc=${dir:t}.zwc
  if [[ $dir == (.|..) || $dir == (.|..)/* ]]; then
    continue
  fi
  files=($dir/*(N−.))
  if [[ −w $dir:h && −n $files ]]; then
    files=(${${(M)files%/*/*}#/})
    if ( cd $dir:h &&
         zrecompile −p −U −z $zwc $files ); then
      fpath[i]=$fpath[i].zwc
    fi
  fi
done
```

The **−U** and **−z** options are appropriate for functions in the default zsh installation **fpath**; you may need to use different options for your personal function directories.

Once the digests have been created and your **fpath** modified to refer to them, you can keep them up to date by running **zrecompile** with no arguments.

Keyboard Definition

The large number of possible combinations of keyboards, workstations, terminals, emulators, and window systems makes it impossible for zsh to have built−in key bindings for every situation. The **zkbd** utility, found in **Functions/Misc**, can help you quickly create key bindings for your configuration.

Run **zkbd** either as an autoloaded function, or as a shell script:

```
zsh −f ~/zsh−5.8/Functions/Misc/zkbd
```

When you run **zkbd**, it first asks you to enter your terminal type; if the default it offers is correct, just press return. It then asks you to press a number of different keys to determine characteristics of your keyboard and terminal; **zkbd** warns you if it finds anything out of the ordinary, such as a Delete key that sends neither

^H nor **^?**.

The keystrokes read by **zkbd** are recorded as a definition for an associative array named **key**, written to a file in the subdirectory **.zkbd** within either your **HOME** or **ZDOTDIR** directory. The name of the file is composed from the **TERM**, **VENDOR** and **OSTYPE** parameters, joined by hyphens.

You may read this file into your **.zshrc** or another startup file with the '**source**' or '**.**' commands, then reference the **key** parameter in bindkey commands, like this:

```
source ${ZDOTDIR:−$HOME}/.zkbd/$TERM−$VENDOR−$OSTYPE
[[ −n ${key[Left]} ]] && bindkey "${key[Left]}" backward−char
[[ −n ${key[Right]} ]] && bindkey "${key[Right]}" forward−char
# etc.
```

Note that in order for '**autoload zkbd**' to work, the **zkdb** file must be in one of the directories named in your **fpath** array (see *zshparam*(1)). This should already be the case if you have a standard zsh installation; if it is not, copy **Functions/Misc/zkbd** to an appropriate directory.

Dumping Shell State

Occasionally you may encounter what appears to be a bug in the shell, particularly if you are using a beta version of zsh or a development release. Usually it is sufficient to send a description of the problem to one of the zsh mailing lists (see *zsh*(1)), but sometimes one of the zsh developers will need to recreate your environment in order to track the problem down.

The script named **reporter**, found in the **Util** directory of the distribution, is provided for this purpose. (It is also possible to **autoload reporter**, but **reporter** is not installed in **fpath** by default.) This script outputs a detailed dump of the shell state, in the form of another script that can be read with '**zsh −f**' to recreate that state.

To use **reporter**, read the script into your shell with the '**.**' command and redirect the output into a file:

```
. ~/zsh−5.8/Util/reporter > zsh.report
```

You should check the **zsh.report** file for any sensitive information such as passwords and delete them by hand before sending the script to the developers. Also, as the output can be voluminous, it's best to wait for the developers to ask for this information before sending it.

You can also use **reporter** to dump only a subset of the shell state. This is sometimes useful for creating startup files for the first time. Most of the output from reporter is far more detailed than usually is necessary for a startup file, but the **aliases**, **options**, and **zstyles** states may be useful because they include only changes from the defaults. The **bindings** state may be useful if you have created any of your own keymaps, because **reporter** arranges to dump the keymap creation commands as well as the bindings for every keymap.

As is usual with automated tools, if you create a startup file with **reporter**, you should edit the results to remove unnecessary commands. Note that if you're using the new completion system, you should *not* dump the **functions** state to your startup files with **reporter**; use the **compdump** function instead (see *zshcompsys*(1)).

reporter [*state* ...]
Print to standard output the indicated subset of the current shell state. The *state* arguments may be one or more of:

all Output everything listed below.
aliases Output alias definitions.
bindings
Output ZLE key maps and bindings.
completion
Output old−style **compctl** commands. New completion is covered by **functions** and **zstyles**.

functions
Output autoloads and function definitions.

limits Output **limit** commands.

options
Output **setopt** commands.

styles Same as **zstyles**.

variables
Output shell parameter assignments, plus **export** commands for any environment variables.

zstyles Output **zstyle** commands.

If the *state* is omitted, **all** is assumed.

With the exception of '**all**', every *state* can be abbreviated by any prefix, even a single letter; thus **a** is the same as **aliases**, **z** is the same as **zstyles**, etc.

Manipulating Hook Functions

add−zsh−hook [**−L** | **−dD**] [**−Uzk**] *hook function*

Several functions are special to the shell, as described in the section SPECIAL FUNCTIONS, see *zshmisc*(1), in that they are automatically called at specific points during shell execution. Each has an associated array consisting of names of functions to be called at the same point; these are so−called 'hook functions'. The shell function **add−zsh−hook** provides a simple way of adding or removing functions from the array.

hook is one of **chpwd**, **periodic**, **precmd**, **preexec**, **zshaddhistory**, **zshexit**, or **zsh_directory_name**, the special functions in question. Note that **zsh_directory_name** is called in a different way from the other functions, but may still be manipulated as a hook.

function is name of an ordinary shell function. If no options are given this will be added to the array of functions to be executed in the given context. Functions are invoked in the order they were added.

If the option **−L** is given, the current values for the hook arrays are listed with **typeset**.

If the option **−d** is given, the *function* is removed from the array of functions to be executed.

If the option **−D** is given, the *function* is treated as a pattern and any matching names of functions are removed from the array of functions to be executed.

The options **−U**, **−z** and **−k** are passed as arguments to **autoload** for *function*. For functions contributed with zsh, the options **−Uz** are appropriate.

add−zle−hook−widget [**−L** | **−dD**] [**−Uzk**] *hook widgetname*

Several widget names are special to the line editor, as described in the section Special Widgets, see *zshzle*(1), in that they are automatically called at specific points during editing. Unlike function hooks, these do not use a predefined array of other names to call at the same point; the shell function **add−zle−hook−widget** maintains a similar array and arranges for the special widget to invoke those additional widgets.

hook is one of **isearch−exit**, **isearch−update**, **line−pre−redraw**, **line−init**, **line−finish**, **history−line−set**, or **keymap−select**, corresponding to each of the special widgets **zle−isearch−exit**, etc. The special widget names are also accepted as the *hook* argument.

widgetname is the name of a ZLE widget. If no options are given this is added to the array of widgets to be invoked in the given hook context. Widgets are invoked in the order they were added, with

```
zle widgetname −Nw −− "$@"
```

Note that this means that the '**WIDGET**' special parameter tracks the *widgetname* when the widget function is called, rather than tracking the name of the corresponding special hook widget.

If the option **−d** is given, the *widgetname* is removed from the array of widgets to be executed.

If the option **−D** is given, the *widgetname* is treated as a pattern and any matching names of widgets are removed from the array.

If *widgetname* does not name an existing widget when added to the array, it is assumed that a shell function also named *widgetname* is meant to provide the implementation of the widget. This name is therefore marked for autoloading, and the options **−U**, **−z** and **−k** are passed as arguments to **autoload** as with **add−zsh−hook**. The widget is also created with '**zle −N** *widgetname*' to cause the corresponding function to be loaded the first time the hook is called.

The arrays of *widgetname* are currently maintained in **zstyle** contexts, one for each *hook* context, with a style of '**widgets**'. If the **−L** option is given, this set of styles is listed with '**zstyle −L**'. This implementation may change, and the special widgets that refer to the styles are created only if **add−zle−hook−widget** is called to add at least one widget, so if this function is used for any hooks, then all hooks should be managed only via this function.

REMEMBERING RECENT DIRECTORIES

The function **cdr** allows you to change the working directory to a previous working directory from a list maintained automatically. It is similar in concept to the directory stack controlled by the **pushd**, **popd** and **dirs** builtins, but is more configurable, and as it stores all entries in files it is maintained across sessions and (by default) between terminal emulators in the current session. Duplicates are automatically removed, so that the list reflects the single most recent use of each directory.

Note that the **pushd** directory stack is not actually modified or used by **cdr** unless you configure it to do so as described in the configuration section below.

Installation

The system works by means of a hook function that is called every time the directory changes. To install the system, autoload the required functions and use the **add−zsh−hook** function described above:

```
autoload −Uz chpwd_recent_dirs cdr add−zsh−hook
add−zsh−hook chpwd chpwd_recent_dirs
```

Now every time you change directly interactively, no matter which command you use, the directory to which you change will be remembered in most−recent−first order.

Use

All direct user interaction is via the **cdr** function.

The argument to cdr is a number *N* corresponding to the *N*th most recently changed−to directory. 1 is the immediately preceding directory; the current directory is remembered but is not offered as a destination. Note that if you have multiple windows open 1 may refer to a directory changed to in another window; you can avoid this by having per−terminal files for storing directory as described for the **recent−dirs−file** style below.

If you set the **recent−dirs−default** style described below **cdr** will behave the same as **cd** if given a non−numeric argument, or more than one argument. The recent directory list is updated just the same however you change directory.

If the argument is omitted, 1 is assumed. This is similar to **pushd**'s behaviour of swapping the two most recent directories on the stack.

Completion for the argument to **cdr** is available if compinit has been run; menu selection is recommended, using:

```
zstyle ':completion:*:*:cdr:*:*' menu selection
```

to allow you to cycle through recent directories; the order is preserved, so the first choice is the most recent directory before the current one. The verbose style is also recommended to ensure the directory is shown; this style is on by default so no action is required unless you have changed it.

Options

The behaviour of **cdr** may be modified by the following options.

−l lists the numbers and the corresponding directories in abbreviated form (i.e. with ˜ substitution reapplied), one per line. The directories here are not quoted (this would only be an issue if a directory name contained a newline). This is used by the completion system.

−r sets the variable **reply** to the current set of directories. Nothing is printed and the directory is not changed.

−e allows you to edit the list of directories, one per line. The list can be edited to any extent you like; no sanity checking is performed. Completion is available. No quoting is necessary (except for newlines, where I have in any case no sympathy); directories are in unabbreviated from and contain an absolute path, i.e. they start with **/**. Usually the first entry should be left as the current directory.

−p **’***pattern***’**

Prunes any items in the directory list that match the given extended glob pattern; the pattern needs to be quoted from immediate expansion on the command line. The pattern is matched against each completely expanded file name in the list; the full string must match, so wildcards at the end (e.g. **’*removeme*’**) are needed to remove entries with a given substring.

If output is to a terminal, then the function will print the new list after pruning and prompt for confirmation by the user. This output and confirmation step can be skipped by using **−P** instead of **−p**.

Configuration

Configuration is by means of the styles mechanism that should be familiar from completion; if not, see the description of the **zstyle** command in see *zshmodules*(1). The context for setting styles should be **’:chpwd:*’** in case the meaning of the context is extended in future, for example:

```
zstyle ':chpwd:*' recent−dirs−max 0
```

sets the value of the **recent−dirs−max** style to 0. In practice the style name is specific enough that a context of ’*’ should be fine.

An exception is **recent−dirs−insert**, which is used exclusively by the completion system and so has the usual completion system context (**’:completion:*’** if nothing more specific is needed), though again **’*’** should be fine in practice.

recent−dirs−default

If true, and the command is expecting a recent directory index, and either there is more than one argument or the argument is not an integer, then fall through to "cd". This allows the lazy to use only one command for directory changing. Completion recognises this, too; see recent−dirs−insert for how to control completion when this option is in use.

recent−dirs−file

The file where the list of directories is saved. The default is **${ZDOTDIR:−$HOME}/.chpwd−recent−dirs**, i.e. this is in your home directory unless you have set the variable **ZDOTDIR** to point somewhere else. Directory names are saved in **$’...’** quoted form, so each line in the file can be supplied directly to the shell as an argument.

The value of this style may be an array. In this case, the first file in the list will always be used for saving directories while any other files are left untouched. When reading the recent directory list, if there are fewer than the maximum number of entries in the first file, the contents of later files in the array will be appended with duplicates removed from the list shown. The contents of the two files are not sorted together, i.e. all the entries in the first file are shown first. The special value **+** can appear in the list to indicate the default file should be read at that point. This allows effects like the following:

```
zstyle ':chpwd:*' recent−dirs−file \
˜/.chpwd−recent−dirs−${TTY##*/} +
```

Recent directories are read from a file numbered according to the terminal. If there are insufficient entries the list is supplemented from the default file.

It is possible to use **zstyle −e** to make the directory configurable at run time:

```
zstyle −e ':chpwd:*' recent−dirs−file pick−recent−dirs−file
pick−recent−dirs−file() {
 if [[ $PWD = ~/text/writing(|/*) ]]; then
   reply=(~/.chpwd−recent−dirs−writing)
 else
   reply=(+)
 fi
}
```

In this example, if the current directory is **~/text/writing** or a directory under it, then use a special file for saving recent directories, else use the default.

recent−dirs−insert

Used by completion. If **recent−dirs−default** is true, then setting this to **true** causes the actual directory, rather than its index, to be inserted on the command line; this has the same effect as using the corresponding index, but makes the history clearer and the line easier to edit. With this setting, if part of an argument was already typed, normal directory completion rather than recent directory completion is done; this is because recent directory completion is expected to be done by cycling through entries menu fashion.

If the value of the style is **always**, then only recent directories will be completed; in that case, use the **cd** command when you want to complete other directories.

If the value is **fallback**, recent directories will be tried first, then normal directory completion is performed if recent directory completion failed to find a match.

Finally, if the value is **both** then both sets of completions are presented; the usual tag mechanism can be used to distinguish results, with recent directories tagged as **recent−dirs**. Note that the recent directories inserted are abbreviated with directory names where appropriate.

recent−dirs−max

The maximum number of directories to save to the file. If this is zero or negative there is no maximum. The default is 20. Note this includes the current directory, which isn't offered, so the highest number of directories you will be offered is one less than the maximum.

recent−dirs−prune

This style is an array determining what directories should (or should not) be added to the recent list. Elements of the array can include:

parent Prune parents (more accurately, ancestors) from the recent list. If present, changing directly down by any number of directories causes the current directory to be overwritten. For example, changing from ~pws to ~pws/some/other/dir causes ~pws not to be left on the recent directory stack. This only applies to direct changes to descendant directories; earlier directories on the list are not pruned. For example, changing from ~pws/yet/another to ~pws/some/other/dir does not cause ~pws to be pruned.

pattern:*pattern*

Gives a zsh pattern for directories that should not be added to the recent list (if not already there). This element can be repeated to add different patterns. For example, **'pattern:/tmp(|/*)'** stops **/tmp** or its descendants from being added. The **EXTENDED_GLOB** option is always turned on for these patterns.

recent−dirs−pushd

If set to true, **cdr** will use **pushd** instead of **cd** to change the directory, so the directory is saved on the directory stack. As the directory stack is completely separate from the list of files saved by the mechanism used in this file there is no obvious reason to do this.

Use with dynamic directory naming

It is possible to refer to recent directories using the dynamic directory name syntax by using the supplied function **zsh_directory_name_cdr** a hook:

```
autoload −Uz add−zsh−hook
add−zsh−hook −Uz zsh_directory_name zsh_directory_name_cdr
```

When this is done, **~[1]** will refer to the most recent directory other than $PWD, and so on. Completion after **~[**... also works.

Details of directory handling

This section is for the curious or confused; most users will not need to know this information.

Recent directories are saved to a file immediately and hence are preserved across sessions. Note currently no file locking is applied: the list is updated immediately on interactive commands and nowhere else (unlike history), and it is assumed you are only going to change directory in one window at once. This is not safe on shared accounts, but in any case the system has limited utility when someone else is changing to a different set of directories behind your back.

To make this a little safer, only directory changes instituted from the command line, either directly or indirectly through shell function calls (but not through subshells, evals, traps, completion functions and the like) are saved. Shell functions should use **cd −q** or **pushd −q** to avoid side effects if the change to the directory is to be invisible at the command line. See the contents of the function **chpwd_recent_dirs** for more details.

ABBREVIATED DYNAMIC REFERENCES TO DIRECTORIES

The dynamic directory naming system is described in the subsection *Dynamic named directories* of the section *Filename Expansion* in *expn*(1). In this, a reference to **~[...]** is expanded by a function found by the hooks mechanism.

The contributed function **zsh_directory_name_generic** provides a system allowing the user to refer to directories with only a limited amount of new code. It supports all three of the standard interfaces for directory naming: converting from a name to a directory, converting in the reverse direction to find a short name, and completion of names.

The main feature of this function is a path−like syntax, combining abbreviations at multiple levels separated by ":". As an example, ~[g:p:s] might specify:

g The top level directory for your git area. This first component has to match, or the function will return indicating another directory name hook function should be tried.

p The name of a project within your git area.

s The source area within that project. This allows you to collapse references to long hierarchies to a very compact form, particularly if the hierarchies are similar across different areas of the disk.

Name components may be completed: if a description is shown at the top of the list of completions, it includes the path to which previous components expand, while the description for an individual completion shows the path segment it would add. No additional configuration is needed for this as the completion system is aware of the dynamic directory name mechanism.

Usage

To use the function, first define a wrapper function for your specific case. We'll assume it's to be autoloaded. This can have any name but we'll refer to it as zdn_mywrapper. This wrapper function will define various variables and then call this function with the same arguments that the wrapper function gets. This configuration is described below.

Then arrange for the wrapper to be run as a zsh_directory_name hook:

```
autoload −Uz add−zsh−hook zsh_diretory_name_generic zdn_mywrapper
add−zsh−hook −U zsh_directory_name zdn_mywrapper
```

Configuration

The wrapper function should define a local associative array zdn_top. Alternatively, this can be set with a style called **mapping**. The context for the style is **:zdn:***wrapper−name* where *wrapper−name* is the function calling zsh_directory_name_generic; for example:

```
zstyle :zdn:zdn_mywrapper: mapping zdn_mywrapper_top
```

The keys in this associative array correspond to the first component of the name. The values are matching directories. They may have an optional suffix with a slash followed by a colon and the name of a variable in the same format to give the next component. (The slash before the colon is to disambiguate the case where a colon is needed in the path for a drive. There is otherwise no syntax for escaping this, so path components whose names start with a colon are not supported.) A special component **:default:** specifies a variable in the form **/:***var* (the path section is ignored and so is usually empty) that will be used for the next component if no variable is given for the path. Variables referred to within **zdn_top** have the same format as **zdn_top** itself, but contain relative paths.

For example,

```
local −A zdn_top=(
  g  ~/git
  ga ~/alternate/git
  gs /scratch/$USER/git/:second2
  :default: /:second1
)
```

This specifies the behaviour of a directory referred to as **~[g:...]** or **~[ga:...]** or **~[gs:...]**. Later path components are optional; in that case **~[g]** expands to **~/git**, and so on. **gs** expands to **/scratch/$USER/git** and uses the associative array **second2** to match the second component; **g** and **ga** use the associative array **second1** to match the second component.

When expanding a name to a directory, if the first component is not **g** or **ga** or **gs**, it is not an error; the function simply returns 1 so that a later hook function can be tried. However, matching the first component commits the function, so if a later component does not match, an error is printed (though this still does not stop later hooks from being executed).

For components after the first, a relative path is expected, but note that multiple levels may still appear. Here is an example of **second1**:

```
local −A second1=(
  p  myproject
  s  somproject
  os otherproject/subproject/:third
)
```

The path as found from **zdn_top** is extended with the matching directory, so **~[g:p]** becomes **~/git/myproject**. The slash between is added automatically (it's not possible to have a later component modify the name of a directory already matched). Only **os** specifies a variable for a third component, and there's no **:default:**, so it's an error to use a name like **~[g:p:x]** or **~[ga:s:y]** because there's nowhere to look up the **x** or **y**.

The associative arrays need to be visible within this function; the generic function therefore uses internal variable names beginning **_zdn_** in order to avoid clashes. Note that the variable **reply** needs to be passed back to the shell, so should not be local in the calling function.

The function does not test whether directories assembled by component actually exist; this allows the system to work across automounted file systems. The error from the command trying to use a non−existent directory should be sufficient to indicate the problem.

Complete example

Here is a full fictitious but usable autoloadable definition of the example function defined by the code above. So **~[gs:p:s]** expands to **/scratch/$USER/git/myscratchproject/top/srcdir** (with **$USER** also expanded).

```
local −A zdn_top=(
  g  ~/git
  ga ~/alternate/git
  gs /scratch/$USER/git/:second2
  :default: /:second1
)

local −A second1=(
  p   myproject
  s   somproject
  os  otherproject/subproject/:third
)

local −A second2=(
  p   myscratchproject
  s   somescratchproject
)

local −A third=(
  s   top/srcdir
  d   top/documentation
)

# autoload not needed if you did this at initialisation...
autoload −Uz zsh_directory_name_generic
zsh_directory_name_generic "$@
```

It is also possible to use global associative arrays, suitably named, and set the style for the context of your wrapper function to refer to this. Then your set up code would contain the following:

```
typeset −A zdn_mywrapper_top=(...)
# ... and so on for other associative arrays ...
zstyle ':zdn:zdn_mywrapper:' mapping zdn_mywrapper_top
autoload −Uz add−zsh−hook zsh_directory_name_generic zdn_mywrapper
add−zsh−hook −U zsh_directory_name zdn_mywrapper
```

and the function **zdn_mywrapper** would contain only the following:

```
zsh_directory_name_generic "$@"
```

GATHERING INFORMATION FROM VERSION CONTROL SYSTEMS

In a lot of cases, it is nice to automatically retrieve information from version control systems (VCSs), such as subversion, CVS or git, to be able to provide it to the user; possibly in the user's prompt. So that you can instantly tell which branch you are currently on, for example.

In order to do that, you may use the **vcs_info** function.

The following VCSs are supported, showing the abbreviated name by which they are referred to within the system:

Bazaar (**bzr**)
  https://bazaar.canonical.com/

Codeville (**cdv**)
  http://freecode.com/projects/codeville/

Concurrent Versioning System (**cvs**)
  https://www.nongnu.org/cvs/

Darcs (**darcs**)
  http://darcs.net/

Fossil (**fossil**)
  https://fossil−scm.org/

Git (**git**)
  https://git−scm.com/

GNU arch (**tla**)
  https://www.gnu.org/software/gnu−arch/

Mercurial (**hg**)
: **https://www.mercurial−scm.org/**

Monotone (**mtn**)
: **https://monotone.ca/**

Perforce (**p4**)
: **https://www.perforce.com/**

Subversion (**svn**)
: **https://subversion.apache.org/**

SVK (**svk**)
: **https://svk.bestpractical.com/**

There is also support for the patch management system **quilt** (**https://savannah.nongnu.org/projects/quilt**). See **Quilt Support** below for details.

To load **vcs_info**:

```
autoload −Uz vcs_info
```

It can be used in any existing prompt, because it does not require any specific **$psvar** entries to be available.

Quickstart

To get this feature working quickly (including colors), you can do the following (assuming, you loaded **vcs_info** properly − see above):

```
zstyle ':vcs_info:*' actionformats \
   '%F{5}(%f%s%F{5})%F{3}−%F{5}[%F{2}%b%F{3}|%F{1}%a%F{5}]%f '
zstyle ':vcs_info:*' formats       \
   '%F{5}(%f%s%F{5})%F{3}−%F{5}[%F{2}%b%F{5}]%f '
zstyle ':vcs_info:(sv[nk]|bzr):*' branchformat '%b%F{1}:%F{3}%r'
precmd () { vcs_info }
PS1='%F{5}[%F{2}%n%F{5}] %F{3}%3~ ${vcs_info_msg_0_}%f%# '
```

Obviously, the last two lines are there for demonstration. You need to call **vcs_info** from your **precmd** function. Once that is done you need a *single quoted* **'${vcs_info_msg_0_}'** in your prompt.

To be able to use **'${vcs_info_msg_0_}'** directly in your prompt like this, you will need to have the **PROMPT_SUBST** option enabled.

Now call the **vcs_info_printsys** utility from the command line:

```
% vcs_info_printsys
## list of supported version control backends:
## disabled systems are prefixed by a hash sign (#)
bzr
cdv
cvs
darcs
fossil
git
hg
mtn
p4
svk
svn
tla
## flavours (cannot be used in the enable or disable styles; they
## are enabled and disabled with their master [git−svn −> git])
## they *can* be used in contexts: ':vcs_info:git−svn:*'.
git−p4
```

git−svn
hg−git
hg−hgsubversion
hg−hgsvn

You may not want all of these because there is no point in running the code to detect systems you do not use. So there is a way to disable some backends altogether:

```
zstyle ':vcs_info:*' disable bzr cdv darcs mtn svk tla
```

You may also pick a few from that list and enable only those:

```
zstyle ':vcs_info:*' enable git cvs svn
```

If you rerun **vcs_info_printsys** after one of these commands, you will see the backends listed in the **disable** style (or backends not in the **enable** style − if you used that) marked as disabled by a hash sign. That means the detection of these systems is skipped *completely*. No wasted time there.

Configuration

The **vcs_info** feature can be configured via **zstyle**.

First, the context in which we are working:
:vcs_info:*vcs−string***:***user−context***:***repo−root−name*

vcs−string
: is one of: **git**, **git−svn**, **git−p4**, **hg**, **hg−git**, **hg−hgsubversion**, **hg−hgsvn**, **darcs**, **bzr**, **cdv**, **mtn**, **svn**, **cvs**, **svk**, **tla**, **p4** or **fossil**. This is followed by '**.quilt−***quilt−mode*' in Quilt mode (see **Quilt Support** for details) and by '**+***hook−name*' while hooks are active (see **Hooks in vcs_info** for details).

  Currently, hooks in quilt mode don't add the '**.quilt−***quilt−mode*' information. This may change in the future.

user−context
: is a freely configurable string, assignable by the user as the first argument to **vcs_info** (see its description below).

repo−root−name
: is the name of a repository in which you want a style to match. So, if you want a setting specific to **/usr/src/zsh**, with that being a CVS checkout, you can set *repo−root−name* to **zsh** to make it so.

There are three special values for *vcs−string*: The first is named **−init−**, that is in effect as long as there was no decision what VCS backend to use. The second is **−preinit−**; it is used *before* **vcs_info** is run, when initializing the data exporting variables. The third special value is **formats** and is used by the **vcs_info_lastmsg** for looking up its styles.

The initial value of *repo−root−name* is **−all−** and it is replaced with the actual name, as soon as it is known. Only use this part of the context for defining the **formats**, **actionformats** or **branchformat** styles, as it is guaranteed that *repo−root−name* is set up correctly for these only. For all other styles, just use **'*'** instead.

There are two pre−defined values for *user−context*:

default the one used if none is specified

command
: used by vcs_info_lastmsg to lookup its styles

You can of course use **':vcs_info:*'** to match all VCSs in all user−contexts at once.

This is a description of all styles that are looked up.

formats
: A list of formats, used when actionformats is not used (which is most of the time).

actionformats
A list of formats, used if there is a special action going on in your current repository; like an interactive rebase or a merge conflict.

branchformat
Some backends replace **%b** in the formats and actionformats styles above, not only by a branch name but also by a revision number. This style lets you modify how that string should look.

nvcsformats
These "formats" are set when we didn't detect a version control system for the current directory or **vcs_info** was disabled. This is useful if you want **vcs_info** to completely take over the generation of your prompt. You would do something like **PS1='${vcs_info_msg_0_}'** to accomplish that.

hgrevformat
hg uses both a hash and a revision number to reference a specific changeset in a repository. With this style you can format the revision string (see **branchformat**) to include either or both. It's only useful when **get−revision** is true. Note, the full 40−character revision id is not available (except when using the **use−simple** option) because executing hg more than once per prompt is too slow; you may customize this behavior using hooks.

max−exports
Defines the maximum number of **vcs_info_msg_*_** variables **vcs_info** will set.

enable A list of backends you want to use. Checked in the **−init−** context. If this list contains an item called **NONE** no backend is used at all and **vcs_info** will do nothing. If this list contains **ALL**, **vcs_info** will use all known backends. Only with **ALL** in **enable** will the **disable** style have any effect. **ALL** and **NONE** are case insensitive.

disable A list of VCSs you don't want **vcs_info** to test for repositories (checked in the **−init−** context, too). Only used if **enable** contains **ALL**.

disable−patterns
A list of patterns that are checked against **$PWD**. If a pattern matches, **vcs_info** will be disabled. This style is checked in the **:vcs_info:−init−:*:−all−** context.

Say, **~/.zsh** is a directory under version control, in which you do not want **vcs_info** to be active, do:

```
zstyle ':vcs_info:*' disable−patterns "${(b)HOME}/.zsh(|/*)"
```

use−quilt
If enabled, the **quilt** support code is active in 'addon' mode. See **Quilt Support** for details.

quilt−standalone
If enabled, 'standalone' mode detection is attempted if no VCS is active in a given directory. See **Quilt Support** for details.

quilt−patch−dir
Overwrite the value of the **$QUILT_PATCHES** environment variable. See **Quilt Support** for details.

quiltcommand
When **quilt** itself is called in quilt support, the value of this style is used as the command name.

check−for−changes
If enabled, this style causes the **%c** and **%u** format escapes to show when the working directory has uncommitted changes. The strings displayed by these escapes can be controlled via the **stagedstr** and **unstagedstr** styles. The only backends that currently support this option are **git**, **hg**, and **bzr** (the latter two only support unstaged).

For this style to be evaluated with the **hg** backend, the **get−revision** style needs to be set and the **use−simple** style needs to be unset. The latter is the default; the former is not.

With the **bzr** backend, *lightweight checkouts* only honor this style if the **use−server** style is set.

Note, the actions taken if this style is enabled are potentially expensive (read: they may be slow, depending on how big the current repository is). Therefore, it is disabled by default.

check−for−staged−changes
This style is like **check−for−changes**, but it never checks the worktree files, only the metadata in the **.${vcs}** dir. Therefore, this style initializes only the **%c** escape (with **stagedstr**) but not the **%u** escape. This style is faster than **check−for−changes**.

In the **git** backend, this style checks for changes in the index. Other backends do not currently implement this style.

This style is disabled by default.

stagedstr
This string will be used in the **%c** escape if there are staged changes in the repository.

unstagedstr
This string will be used in the **%u** escape if there are unstaged changes in the repository.

command
This style causes **vcs_info** to use the supplied string as the command to use as the VCS's binary. Note, that setting this in '**:vcs_info:***' is not a good idea.

If the value of this style is empty (which is the default), the used binary name is the name of the backend in use (e.g. **svn** is used in an **svn** repository).

The **repo−root−name** part in the context is always the default **−all−** when this style is looked up.

For example, this style can be used to use binaries from non−default installation directories. Assume, **git** is installed in /usr/bin but your sysadmin installed a newer version in /usr/local/bin. Instead of changing the order of your **$PATH** parameter, you can do this:

```
zstyle ':vcs_info:git:*:−all−' command /usr/local/bin/git
```

use−server
This is used by the Perforce backend (**p4**) to decide if it should contact the Perforce server to find out if a directory is managed by Perforce. This is the only reliable way of doing this, but runs the risk of a delay if the server name cannot be found. If the server (more specifically, the **host:port** pair describing the server) cannot be contacted, its name is put into the associative array **vcs_info_p4_dead_servers** and is not contacted again during the session until it is removed by hand. If you do not set this style, the **p4** backend is only usable if you have set the environment variable **P4CONFIG** to a file name and have corresponding files in the root directories of each Perforce client. See comments in the function **VCS_INFO_detect_p4** for more detail.

The Bazaar backend (**bzr**) uses this to permit contacting the server about lightweight checkouts, see the **check−for−changes** style.

use−simple
If there are two different ways of gathering information, you can select the simpler one by setting this style to true; the default is to use the not−that−simple code, which is potentially a lot slower but might be more accurate in all possible cases. This style is used by the **bzr** and **hg** backends. In the case of **hg** it will invoke the external hexdump program to parse the binary dirstate cache file; this method will not return the local revision number.

get−revision
If set to true, vcs_info goes the extra mile to figure out the revision of a repository's work tree (currently for the **git** and **hg** backends, where this kind of information is not always vital). For **git**, the hash value of the currently checked out commit is available via the **%i** expansion. With **hg**, the local revision number and the corresponding global hash are available via **%i**.

get−mq
If set to true, the **hg** backend will look for a Mercurial Queue (**mq**) patch directory. Information will be available via the '**%m**' replacement.

get−bookmarks
If set to true, the **hg** backend will try to get a list of current bookmarks. They will be available via the '**%m**' replacement.

The default is to generate a comma−separated list of all bookmark names that refer to the currently checked out revision. If a bookmark is active, its name is suffixed an asterisk and placed first in the list.

use−prompt−escapes
Determines if we assume that the assembled string from **vcs_info** includes prompt escapes. (Used by **vcs_info_lastmsg**.)

debug Enable debugging output to track possible problems. Currently this style is only used by **vcs_info**'s hooks system.

hooks A list style that defines hook−function names. See **Hooks in vcs_info** below for details.

patch−format
nopatch−format
This pair of styles format the patch information used by the **%m** expando in formats and actionformats for the **git** and **hg** backends. The value is subject to certain **%**−expansions described below. The expanded value is made available in the global **backend_misc** array as **${backend_misc[patches]}** (also if a **set−patch−format** hook is used).

get−unapplied
This boolean style controls whether a backend should attempt to gather a list of unapplied patches (for example with Mercurial Queue patches).

Used by the **quilt** and **hg** backends.

The default values for these styles in all contexts are:

formats	" (%s)−[%b]%u%c−"
actionformats	" (%s)−[%b|%a]%u%c−"
branchformat	"%b:%r" (for bzr, svn, svk and hg)
nvcsformats	""
hgrevformat	"%r:%h"
max−exports	2
enable	ALL
disable	(empty list)
disable−patterns	(empty list)
check−for−changes	false
check−for−staged−changes	false
stagedstr	(string: "S")
unstagedstr	(string: "U")
command	(empty string)

use−server	false
use−simple	false
get−revision	false
get−mq	true
get−bookmarks	false
use−prompt−escapes	true
debug	false
hooks	(empty list)
use−quilt	false
quilt−standalone	false
quilt−patch−dir	empty − use **$QUILT_PATCHES**
quiltcommand	quilt
patch−format	*backend dependent*
nopatch−format	*backend dependent*
get−unapplied	false

In normal **formats** and **actionformats** the following replacements are done:

%s The VCS in use (git, hg, svn, etc.).

%b Information about the current branch.

%a An identifier that describes the action. Only makes sense in **actionformats**.

%i The current revision number or identifier. For **hg** the **hgrevformat** style may be used to customize the output.

%c The string from the **stagedstr** style if there are staged changes in the repository.

%u The string from the **unstagedstr** style if there are unstaged changes in the repository.

%R The base directory of the repository.

%r The repository name. If **%R** is **/foo/bar/repoXY**, **%r** is **repoXY**.

%S A subdirectory within a repository. If **$PWD** is **/foo/bar/repoXY/beer/tasty**, **%S** is **beer/tasty**.

%m A "misc" replacement. It is at the discretion of the backend to decide what this replacement expands to.

The **hg** and **git** backends use this expando to display patch information. **hg** sources patch information from the **mq** extensions; **git** from in−progress **rebase** and **cherry−pick** operations and from the **stgit** extension. The **patch−format** and **nopatch−format** styles control the generated string. The former is used when at least one patch from the patch queue has been applied, and the latter otherwise.

The **hg** backend displays bookmark information in this expando (in addition to **mq** information). See the **get−mq** and **get−bookmarks** styles. Both of these styles may be enabled at the same time. If both are enabled, both resulting strings will be shown separated by a semicolon (that cannot currently be customized).

The **quilt** ‘standalone’ backend sets this expando to the same value as the **%Q** expando.

%Q Quilt series information. When quilt is used (either in ‘addon’ mode or as a ‘standalone’ backend), this expando is set to quilt series’ **patch−format** string. The **set−patch−format** hook and **nopatch−format** style are honoured.

See **Quilt Support** below for details.

In **branchformat** these replacements are done:

%b	The branch name.
%r	The current revision number or the **hgrevformat** style for **hg**.

In **hgrevformat** these replacements are done:

%r	The current local revision number.
%h	The current global revision identifier.

In **patch−format** and **nopatch−format** these replacements are done:

%p	The name of the top−most applied patch; may be overridden by the **applied−string** hook.
%u	The number of unapplied patches; may be overridden by the **unapplied−string** hook.
%n	The number of applied patches.
%c	The number of unapplied patches.
%a	The number of all patches (**%a = %n + %c**).
%g	The names of active **mq** guards (**hg** backend).
%G	The number of active **mq** guards (**hg** backend).

Not all VCS backends have to support all replacements. For **nvcsformats** no replacements are performed at all, it is just a string.

Oddities

If you want to use the **%b** (bold off) prompt expansion in **formats**, which expands **%b** itself, use **%%b**. That will cause the **vcs_info** expansion to replace **%%b** with **%b**, so that zsh’s prompt expansion mechanism can handle it. Similarly, to hand down **%b** from **branchformat**, use **%%%%b**. Sorry for this inconvenience, but it cannot be easily avoided. Luckily we do not clash with a lot of prompt expansions and this only needs to be done for those.

When one of the **gen−applied−string**, **gen−unapplied−string**, and **set−patch−format** hooks is defined, applying **%**−escaping (‘**foo=${foo//’%’/%%}**’) to the interpolated values for use in the prompt is the responsibility of those hooks (jointly); when neither of those hooks is defined, **vcs_info** handles escaping by itself. We regret this coupling, but it was required for backwards compatibility.

Quilt Support

Quilt is not a version control system, therefore this is not implemented as a backend. It can help keeping track of a series of patches. People use it to keep a set of changes they want to use on top of software packages (which is tightly integrated into the package build process − the Debian project does this for a large number of packages). Quilt can also help individual developers keep track of their own patches on top of real version control systems.

The **vcs_info** integration tries to support both ways of using quilt by having two slightly different modes of operation: ‘addon’ mode and ‘standalone’ mode).

Quilt integration is off by default; to enable it, set the **use−quilt** style, and add **%Q** to your **formats** or **actionformats** style:

```
zstyle ':vcs_info:*' use−quilt true
```

Styles looked up from the Quilt support code include ‘**.quilt−***quilt−mode*’ in the *vcs−string* part of the context, where *quilt−mode* is either **addon** or **standalone**. Example: **:vcs_info:git.quilt−addon:default:***repo−root−name*.

For ‘addon’ mode to become active **vcs_info** must have already detected a real version control system controlling the directory. If that is the case, a directory that holds quilt’s patches needs to be found. That directory is configurable via the ‘**QUILT_PATCHES**’ environment variable. If that variable exists its value is used, otherwise the value ‘**patches**’ is assumed. The value from **$QUILT_PATCHES** can be overwritten using the '**quilt−patches**' style. (Note: you can use **vcs_info** to keep the value of **$QUILT_PATCHES** correct all the time via the **post−quilt** hook).

When the directory in question is found, quilt is assumed to be active. To gather more information, **vcs_info** looks for a directory called '.pc'; Quilt uses that directory to track its current state. If this directory does not exist we know that quilt has not done anything to the working directory (read: no patches have been applied yet).

If patches are applied, **vcs_info** will try to find out which. If you want to know which patches of a series are not yet applied, you need to activate the **get−unapplied** style in the appropriate context.

vcs_info allows for very detailed control over how the gathered information is presented (see the **Configuration** and **Hooks in vcs_info** sections), all of which are documented below. Note there are a number of other patch tracking systems that work on top of a certain version control system (like **stgit** for **git**, or **mq** for **hg**); the configuration for systems like that are generally configured the same way as the **quilt** support.

If the **quilt** support is working in 'addon' mode, the produced string is available as a simple format replacement (**%Q** to be precise), which can be used in **formats** and **actionformats**; see below for details).

If, on the other hand, the support code is working in 'standalone' mode, **vcs_info** will pretend as if **quilt** were an actual version control system. That means that the version control system identifier (which otherwise would be something like 'svn' or 'cvs') will be set to '**−quilt−**'. This has implications on the used style context where this identifier is the second element. **vcs_info** will have filled in a proper value for the "repository's" root directory and the string containing the information about quilt's state will be available as the 'misc' replacement (and **%Q** for compatibility with 'addon' mode).

What is left to discuss is how 'standalone' mode is detected. The detection itself is a series of searches for directories. You can have this detection enabled all the time in every directory that is not otherwise under version control. If you know there is only a limited set of trees where you would like **vcs_info** to try and look for Quilt in 'standalone' mode to minimise the amount of searching on every call to **vcs_info**, there are a number of ways to do that:

Essentially, 'standalone' mode detection is controlled by a style called '**quilt−standalone**'. It is a string style and its value can have different effects. The simplest values are: '**always**' to run detection every time **vcs_info** is run, and '**never**' to turn the detection off entirely.

If the value of **quilt−standalone** is something else, it is interpreted differently. If the value is the name of a scalar variable the value of that variable is checked and that value is used in the same 'always'/'never' way as described above.

If the value of **quilt−standalone** is an array, the elements of that array are used as directory names under which you want the detection to be active.

If **quilt−standalone** is an associative array, the keys are taken as directory names under which you want the detection to be active, but only if the corresponding value is the string '**true**'.

Last, but not least, if the value of **quilt−standalone** is the name of a function, the function is called without arguments and the return value decides whether detection should be active. A '0' return value is true; a non−zero return value is interpreted as false.

Note, if there is both a function and a variable by the name of **quilt−standalone**, the function will take precedence.

Function Descriptions (Public API)

vcs_info [*user−context*]
: The main function, that runs all backends and assembles all data into **${vcs_info_msg_*_}**. This is the function you want to call from **precmd** if you want to include up−to−date information in your prompt (see **Variable Description** below). If an argument is given, that string will be used instead of **default** in the *user−context* field of the style context.

vcs_info_hookadd
: Statically registers a number of functions to a given hook. The hook needs to be given as the first argument; what follows is a list of hook−function names to register to the hook. The '**+vi−**' prefix

needs to be left out here. See **Hooks in vcs_info** below for details.

vcs_info_hookdel
Remove hook−functions from a given hook. The hook needs to be given as the first non−option argument; what follows is a list of hook−function names to un−register from the hook. If '**−a**' is used as the first argument, **all** occurrences of the functions are unregistered. Otherwise only the last occurrence is removed (if a function was registered to a hook more than once). The '**+vi−**' prefix needs to be left out here. See **Hooks in vcs_info** below for details.

vcs_info_lastmsg
Outputs the last **${vcs_info_msg_*_}** value. Takes into account the value of the **use−prompt−escapes** style in **':vcs_info:formats:command:−all−'**. It also only prints **max−exports** values.

vcs_info_printsys [*user−context*]
Prints a list of all supported version control systems. Useful to find out possible contexts (and which of them are enabled) or values for the **disable** style.

vcs_info_setsys
Initializes **vcs_info**'s internal list of available backends. With this function, you can add support for new VCSs without restarting the shell.

All functions named **VCS_INFO_*** are for internal use only.

Variable Description

${vcs_info_msg_*N*_} (Note the trailing underscore)
Where *N* is an integer, e.g., **vcs_info_msg_0_**. These variables are the storage for the informational message the last **vcs_info** call has assembled. These are strongly connected to the **formats**, **actionformats** and **nvcsformats** styles described above. Those styles are lists. The first member of that list gets expanded into **${vcs_info_msg_0_}**, the second into **${vcs_info_msg_1_}** and the Nth into **${vcs_info_msg_N−1_}**. (See the **max−exports** style above.)

All variables named **VCS_INFO_*** are for internal use only.

Hooks in vcs_info

Hooks are places in **vcs_info** where you can run your own code. That code can communicate with the code that called it and through that, change the system's behaviour.

For configuration, hooks change the style context:
:vcs_info:*vcs−string***+***hook−name***:***user−context***:***repo−root−name*

To register functions to a hook, you need to list them in the **hooks** style in the appropriate context.

Example:
```
zstyle ':vcs_info:*+foo:*' hooks bar baz
```

This registers functions to the hook 'foo' for all backends. In order to avoid namespace problems, all registered function names are prepended by a '**+vi−**', so the actual functions called for the 'foo' hook are '**+vi−bar**' and '**+vi−baz**'.

If you would like to register a function to a hook regardless of the current context, you may use the **vcs_info_hookadd** function. To remove a function that was added like that, the **vcs_info_hookdel** function can be used.

If something seems weird, you can enable the 'debug' boolean style in the proper context and the hook−calling code will print what it tried to execute and whether the function in question existed.

When you register more than one function to a hook, all functions are executed one after another until one function returns non−zero or until all functions have been called. Context−sensitive hook functions are executed **before** statically registered ones (the ones added by **vcs_info_hookadd**).

You may pass data between functions via an associative array, **user_data**. For example:
```
+vi−git−myfirsthook(){
  user_data[myval]=$myval
}
```

```
+vi-git-mysecondhook(){
    # do something with ${user_data[myval]}
}
```

There are a number of variables that are special in hook contexts:

ret The return value that the hooks system will return to the caller. The default is an integer ‘zero’. If and how a changed **ret** value changes the execution of the caller depends on the specific hook. See the hook documentation below for details.

hook_com
An associated array which is used for bidirectional communication from the caller to hook functions. The used keys depend on the specific hook.

context
The active context of the hook. Functions that wish to change this variable should make it local scope first.

vcs The current VCS after it was detected. The same values as in the enable/disable style are used. Available in all hooks except **start−up**.

Finally, the full list of currently available hooks:

start−up
Called after starting **vcs_info** but before the VCS in this directory is determined. It can be used to deactivate **vcs_info** temporarily if necessary. When **ret** is set to **1**, **vcs_info** aborts and does nothing; when set to **2**, **vcs_info** sets up everything as if no version control were active and exits.

pre−get−data
Same as **start−up** but after the VCS was detected.

gen−hg−bookmark−string
Called in the Mercurial backend when a bookmark string is generated; the **get−revision** and **get−bookmarks** styles must be true.

This hook gets the names of the Mercurial bookmarks that **vcs_info** collected from ‘hg’.

If a bookmark is active, the key **${hook_com[hg−active−bookmark]}** is set to its name. The key is otherwise unset.

When setting **ret** to non−zero, the string in **${hook_com[hg−bookmark−string]}** will be used in the **%m** escape in **formats** and **actionformats** and will be available in the global **backend_misc** array as **${backend_misc[bookmarks]}**.

gen−applied−string
Called in the **git** (with **stgit** or during rebase or merge), and **hg** (with **mq**) backends and in **quilt** support when the **applied−string** is generated; the **use−quilt** zstyle must be true for **quilt** (the **mq** and **stgit** backends are active by default).

This hook gets the names of all applied patches which **vcs_info** collected so far in the opposite order, which means that the first argument is the top−most patch and so forth.

When setting **ret** to non−zero, the string in **${hook_com[applied−string]}** will be available as **%p** in the **patch−format** and **nopatch−format** styles. This hook is, in concert with **set−patch−format**, responsible for **%**−escaping that value for use in the prompt. (See the **Oddities** section.)

gen−unapplied−string
Called in the **git** (with **stgit** or during rebase), and **hg** (with **mq**) backend and in **quilt** support when the **unapplied−string** is generated; the **get−unapplied** style must be true.

This hook gets the names of all unapplied patches which **vcs_info** collected so far in order, which means that the first argument is the patch next−in−line to be applied and so forth.

When setting **ret** to non−zero, the string in **${hook_com[unapplied−string]}** will be available as

%u in the **patch−format** and **nopatch−format** styles. This hook is, in concert with **set−patch−format**, responsible for **%**−escaping that value for use in the prompt. (See the **Oddities** section.)

gen−mqguards−string
Called in the **hg** backend when **guards−string** is generated; the **get−mq** style must be true (default).

This hook gets the names of any active **mq** guards.

When setting **ret** to non−zero, the string in **${hook_com[guards−string]}** will be used in the **%g** escape in the **patch−format** and **nopatch−format** styles.

no−vcs This hooks is called when no version control system was detected.

The '**hook_com**' parameter is not used.

post−backend
Called as soon as the backend has finished collecting information.

The '**hook_com**' keys available are as for the **set−message** hook.

post−quilt
Called after the **quilt** support is done. The following information is passed as arguments to the hook: 1. the quilt−support mode ('addon' or 'standalone'); 2. the directory that contains the patch series; 3. the directory that holds quilt's status information (the '.pc' directory) or the string **"−nopc−"** if that directory wasn't found.

The 'hook_com' parameter is not used.

set−branch−format
Called before '**branchformat**' is set. The only argument to the hook is the format that is configured at this point.

The '**hook_com**' keys considered are '**branch**' and '**revision**'. They are set to the values figured out so far by **vcs_info** and any change will be used directly when the actual replacement is done.

If **ret** is set to non−zero, the string in **${hook_com[branch−replace]}** will be used unchanged as the '**%b**' replacement in the variables set by **vcs_info**.

set−hgrev−format
Called before a '**hgrevformat**' is set. The only argument to the hook is the format that is configured at this point.

The '**hook_com**' keys considered are '**hash**' and '**localrev**'. They are set to the values figured out so far by **vcs_info** and any change will be used directly when the actual replacement is done.

If **ret** is set to non−zero, the string in **${hook_com[rev−replace]}** will be used unchanged as the '**%i**' replacement in the variables set by **vcs_info**.

pre−addon−quilt
This hook is used when **vcs_info**'s quilt functionality is active in "addon" mode (quilt used on top of a real version control system). It is activated right before any quilt specific action is taken.

Setting the '**ret**' variable in this hook to a non−zero value avoids any quilt specific actions from being run at all.

set−patch−format
This hook is used to control some of the possible expansions in **patch−format** and **nopatch−format** styles with patch queue systems such as quilt, mqueue and the like.

This hook is used in the **git**, **hg** and **quilt** backends.

The hook allows the control of the **%p** (**${hook_com[applied]}**) and **%u** (**${hook_com[unapplied]}**) expansion in all backends that use the hook. With the mercurial backend, the **%g** (**${hook_com[guards]}**) expansion is controllable in addition to that.

If **ret** is set to non−zero, the string in **${hook_com[patch−replace]}** will be used unchanged instead of an expanded format from **patch−format** or **nopatch−format**.

This hook is, in concert with the **gen−applied−string** or **gen−unapplied−string** hooks if they are defined, responsible for **%**−escaping the final **patch−format** value for use in the prompt. (See the **Oddities** section.)

set−message

Called each time before a '**vcs_info_msg_***N***_**' message is set. It takes two arguments; the first being the '*N*' in the message variable name, the second is the currently configured **formats** or **actionformats**.

There are a number of '**hook_com**' keys, that are used here: '**action**', '**branch**', '**base**', '**base−name**', '**subdir**', '**staged**', '**unstaged**', '**revision**', '**misc**', '**vcs**' and one '**miscN**' entry for each backend−specific data field (**N** starting at zero). They are set to the values figured out so far by **vcs_info** and any change will be used directly when the actual replacement is done.

Since this hook is triggered multiple times (once for each configured **formats** or **actionformats**), each of the '**hook_com**' keys mentioned above (except for the **miscN** entries) has an '**_orig**' counterpart, so even if you changed a value to your liking you can still get the original value in the next run. Changing the '**_orig**' values is probably not a good idea.

If **ret** is set to non−zero, the string in **${hook_com[message]}** will be used unchanged as the message by **vcs_info**.

If all of this sounds rather confusing, take a look at the **Examples** section below and also in the **Misc/vcs_info−examples** file in the Zsh source. They contain some explanatory code.

Examples

Don't use **vcs_info** at all (even though it's in your prompt):

```
zstyle ':vcs_info:*' enable NONE
```

Disable the backends for **bzr** and **svk**:

```
zstyle ':vcs_info:*' disable bzr svk
```

Disable everything *but* **bzr** and **svk**:

```
zstyle ':vcs_info:*' enable bzr svk
```

Provide a special formats for **git**:

```
zstyle ':vcs_info:git:*' formats       ' GIT, BABY! [%b]'
zstyle ':vcs_info:git:*' actionformats ' GIT ACTION! [%b|%a]'
```

All **%x** expansion in all sorts of formats (**formats**, **actionformats**, **branchformat**, you name it) are done using the '**zformat**' builtin from the '**zsh/zutil**' module. That means you can do everything with these **%x** items what zformat supports. In particular, if you want something that is really long to have a fixed width, like a hash in a mercurial branchformat, you can do this: **%12.12i**. That'll shrink the 40 character hash to its 12 leading characters. The form is actually '*%min.max***x**'. More is possible. See the section 'The zsh/zutil Module' in *zshmodules*(1) for details.

Use the quicker **bzr** backend

```
zstyle ':vcs_info:bzr:*' use−simple true
```

If you do use **use−simple**, please report if it does 'the−right−thing[tm]'.

Display the revision number in yellow for **bzr** and **svn**:

```
zstyle ':vcs_info:(svn|bzr):*' \
     branchformat '%b%{'${fg[yellow]}'%}:%r'
```

If you want colors, make sure you enclose the color codes in **%{...%}** if you want to use the string provided by **vcs_info** in prompts.

Here is how to print the VCS information as a command (not in a prompt):

```
alias vcsi='vcs_info command; vcs_info_lastmsg'
```

This way, you can even define different formats for output via **vcs_info_lastmsg** in the

'**:vcs_info:*:command:***' namespace.

Now as promised, some code that uses hooks: say, you'd like to replace the string 'svn' by 'subversion' in **vcs_info**'s **%s formats** replacement.

First, we will tell **vcs_info** to call a function when populating the message variables with the gathered information:

```
zstyle ':vcs_info:*+set-message:*' hooks svn2subversion
```

Nothing happens. Which is reasonable, since we didn't define the actual function yet. To see what the hooks subsystem is trying to do, enable the '**debug**' style:

```
zstyle ':vcs_info:*+*:*' debug true
```

That should give you an idea what is going on. Specifically, the function that we are looking for is '**+vi-svn2subversion**'. Note, the '**+vi-**' prefix. So, everything is in order, just as documented. When you are done checking out the debugging output, disable it again:

```
zstyle ':vcs_info:*+*:*' debug false
```

Now, let's define the function:

```
function +vi-svn2subversion() {
  [[ ${hook_com[vcs_orig]} == svn ]] && hook_com[vcs]=subversion
}
```

Simple enough. And it could have even been simpler, if only we had registered our function in a less generic context. If we do it only in the '**svn**' backend's context, we don't need to test which the active backend is:

```
zstyle ':vcs_info:svn+set-message:*' hooks svn2subversion
function +vi-svn2subversion() {
  hook_com[vcs]=subversion
}
```

And finally a little more elaborate example, that uses a hook to create a customised bookmark string for the **hg** backend.

Again, we start off by registering a function:

```
zstyle ':vcs_info:hg+gen-hg-bookmark-string:*' hooks hgbookmarks
```

And then we define the '**+vi-hgbookmarks**' function:

```
function +vi-hgbookmarks() {
  # The default is to connect all bookmark names by
  # commas. This mixes things up a little.
  # Imagine, there's one type of bookmarks that is
  # special to you. Say, because it's *your* work.
  # Those bookmarks look always like this: "sh/*"
  # (because your initials are sh, for example).
  # This makes the bookmarks string use only those
  # bookmarks. If there's more than one, it
  # concatenates them using commas.
  # The bookmarks returned by 'hg' are available in
  # the function's positional parameters.
  local s="${(Mj:,:)@:#sh/*}"
  # Now, the communication with the code that calls
  # the hook functions is done via the hook_com[]
  # hash. The key at which the 'gen-hg-bookmark-string'
  # hook looks is 'hg-bookmark-string'. So:
  hook_com[hg-bookmark-string]=$s
  # And to signal that we want to use the string we
  # just generated, set the special variable 'ret' to
  # something other than the default zero:
  ret=1
```

```
    return 0
}
```

Some longer examples and code snippets which might be useful are available in the examples file located at Misc/vcs_info−examples in the Zsh source directory.

This concludes our guided tour through zsh's **vcs_info**.

PROMPT THEMES

Installation

You should make sure all the functions from the **Functions/Prompts** directory of the source distribution are available; they all begin with the string '**prompt_**' except for the special function'**promptinit**'. You also need the '**colors**' and '**add−zsh−hook**' functions from **Functions/Misc**. All these functions may already be installed on your system; if not, you will need to find them and copy them. The directory should appear as one of the elements of the **fpath** array (this should already be the case if they were installed), and at least the function **promptinit** should be autoloaded; it will autoload the rest. Finally, to initialize the use of the system you need to call the **promptinit** function. The following code in your **.zshrc** will arrange for this; assume the functions are stored in the directory **~/myfns**:

```
fpath=(~/myfns $fpath)
autoload −U promptinit
promptinit
```

Theme Selection

Use the **prompt** command to select your preferred theme. This command may be added to your **.zshrc** following the call to **promptinit** in order to start zsh with a theme already selected.

prompt [**−c** | **−l**]
prompt [**−p** | **−h**] [*theme* ...]
prompt [**−s**] *theme* [*arg* ...]

Set or examine the prompt theme. With no options and a *theme* argument, the theme with that name is set as the current theme. The available themes are determined at run time; use the **−l** option to see a list. The special *theme* '**random**' selects at random one of the available themes and sets your prompt to that.

In some cases the *theme* may be modified by one or more arguments, which should be given after the theme name. See the help for each theme for descriptions of these arguments.

Options are:

−c Show the currently selected theme and its parameters, if any.
−l List all available prompt themes.
−p Preview the theme named by *theme*, or all themes if no *theme* is given.
−h Show help for the theme named by *theme*, or for the **prompt** function if no *theme* is given.
−s Set *theme* as the current theme and save state.

prompt_*theme***_setup**

Each available *theme* has a setup function which is called by the **prompt** function to install that theme. This function may define other functions as necessary to maintain the prompt, including functions used to preview the prompt or provide help for its use. You should not normally call a theme's setup function directly.

Utility Themes

prompt off

The theme '**off**' sets all the prompt variables to minimal values with no special effects.

prompt default

The theme '**default**' sets all prompt variables to the same state as if an interactive zsh was started with no initialization files.

prompt restore

The special theme '**restore**' erases all theme settings and sets prompt variables to their state before the first time the '**prompt**' function was run, provided each theme has properly defined its cleanup (see below).

Note that you can undo '**prompt off**' and '**prompt default**' with '**prompt restore**', but a second restore does not undo the first.

Writing Themes

The first step for adding your own theme is to choose a name for it, and create a file '**prompt_***name***_setup**' *in a directory in your* **fpath**, *such as* **~/myfns** *in the example above. The file should at minimum contain assignments for the prompt variables that your theme wishes to modify. By convention, themes use* **PS1**, **PS2**, **RPS1**, *etc., rather than the longer* **PROMPT** *and* **RPROMPT**.

The file is autoloaded as a function in the current shell context, so it may contain any necessary commands to customize your theme, including defining additional functions. To make some complex tasks easier, your setup function may also do any of the following:

Assign **prompt_opts**

The array **prompt_opts** may be assigned any of **"bang"**, **"cr"**, **"percent"**, **"sp"**, and/or **"subst"** as values. The corresponding setopts (**promptbang**, etc.) are turned on, all other prompt−related options are turned off. The **prompt_opts** array preserves setopts even beyond the scope of **localoptions**, should your function need that.

Modify precmd and preexec

Use of **add−zsh−hook** is recommended. The **precmd** and **preexec** hooks are automatically adjusted if the prompt theme changes or is disabled.

Declare cleanup

If your function makes any other changes that should be undone when the theme is disabled, your setup function may call

prompt_cleanup *command*

where command should be suitably quoted. If your theme is ever disabled or replaced by another, command is executed with **eval**. *You may declare more than one such cleanup hook.*

Define preview

Define or autoload a function **prompt_***name***_preview** *to display a simulated version of your prompt. A simple default previewer is defined by* **promptinit** *for themes that do not define their own. This preview function is called by* '**prompt −p**'.

Provide help

Define or autoload a function **prompt_***name***_help** *to display documentation or help text for your theme. This help function is called by* '**prompt −h**'.

ZLE FUNCTIONS

Widgets

These functions all implement user−defined ZLE widgets (see *zshzle*(1)) which can be bound to keystrokes in interactive shells. To use them, your **.zshrc** should contain lines of the form

```
autoload function
zle −N function
```

followed by an appropriate **bindkey** command to associate the function with a key sequence. Suggested bindings are described below.

bash−style word functions

If you are looking for functions to implement moving over and editing words in the manner of bash, where only alphanumeric characters are considered word characters, you can use the functions described in the next section. The following is sufficient:

```
autoload −U select−word−style
select−word−style bash
```

forward−word−match, **backward−word−match**
kill−word−match, **backward−kill−word−match**
transpose−words−match, **capitalize−word−match**
up−case−word−match, **down−case−word−match**
delete−whole−word−match, **select−word−match**
select−word−style, **match−word−context**, **match−words−by−style**

The first eight '**−match**' functions are drop−in replacements for the builtin widgets without the suffix. By default they behave in a similar way. However, by the use of styles and the function **select−word−style**, the way words are matched can be altered. **select−word−match** is intended to be used as a text object in vi mode but with custom word styles. For comparison, the widgets described in *zshzle*(1) under Text Objects use fixed definitions of words, compatible with the **vim** editor.

The simplest way of configuring the functions is to use **select−word−style**, which can either be called as a normal function with the appropriate argument, or invoked as a user−defined widget that will prompt for the first character of the word style to be used. The first time it is invoked, the first eight **−match** functions will automatically replace the builtin versions, so they do not need to be loaded explicitly.

The word styles available are as follows. Only the first character is examined.

bash Word characters are alphanumeric characters only.

normal
As in normal shell operation: word characters are alphanumeric characters plus any characters present in the string given by the parameter **$WORDCHARS**.

shell Words are complete shell command arguments, possibly including complete quoted strings, or any tokens special to the shell.

whitespace
Words are any set of characters delimited by whitespace.

default Restore the default settings; this is usually the same as '**normal**'.

All but '**default**' can be input as an upper case character, which has the same effect but with subword matching turned on. In this case, words with upper case characters are treated specially: each separate run of upper case characters, or an upper case character followed by any number of other characters, is considered a word. The style **subword−range** can supply an alternative character range to the default '**[:upper:]**'; the value of the style is treated as the contents of a '**[...]**' pattern (note that the outer brackets should not be supplied, only those surrounding named ranges).

More control can be obtained using the **zstyle** command, as described in *zshmodules*(1). Each style is looked up in the context **:zle:***widget* where *widget* is the name of the user−defined widget, not the name of the function implementing it, so in the case of the definitions supplied by **select−word−style** the appropriate contexts are **:zle:forward−word**, and so on. The function **select−word−style** itself always defines styles for the context '**:zle:***' which can be overridden by more specific (longer) patterns as well as explicit contexts.

The style **word−style** specifies the rules to use. This may have the following values.

normal
Use the standard shell rules, i.e. alphanumerics and **$WORDCHARS**, unless overridden by the styles **word−chars** or **word−class**.

specified
Similar to **normal**, but *only* the specified characters, and not also alphanumerics, are considered word characters.

unspecified
The negation of specified. The given characters are those which will *not* be considered part of a word.

shell Words are obtained by using the syntactic rules for generating shell command arguments. In addition, special tokens which are never command arguments such as '()' are also treated as words.

whitespace
Words are whitespace−delimited strings of characters.

The first three of those rules usually use **$WORDCHARS**, but the value in the parameter can be overridden by the style **word−chars**, which works in exactly the same way as **$WORDCHARS**. In addition, the style **word−class** uses character class syntax to group characters and takes precedence over **word−chars** if both are set. The **word−class** style does not include the surrounding brackets of the character class; for example, '**−:[:alnum:]**' is a valid **word−class** to include all alphanumerics plus the characters '**−**' and '**:**'. Be careful including ']', '^' and '**−**' as these are special inside character classes.

word−style may also have '**−subword**' appended to its value to turn on subword matching, as described above.

The style **skip−chars** is mostly useful for **transpose−words** and similar functions. If set, it gives a count of characters starting at the cursor position which will not be considered part of the word and are treated as space, regardless of what they actually are. For example, if

```
zstyle ':zle:transpose−words' skip−chars 1
```

has been set, and **transpose−words−match** is called with the cursor on the *X* of **foo***X***bar**, where *X* can be any character, then the resulting expression is **bar***X***foo**.

Finer grained control can be obtained by setting the style **word−context** to an array of pairs of entries. Each pair of entries consists of a *pattern* and a *subcontext*. The shell argument the cursor is on is matched against each *pattern* in turn until one matches; if it does, the context is extended by a colon and the corresponding *subcontext*. Note that the test is made against the original word on the line, with no stripping of quotes. Special handling is done between words: the current context is examined and if it contains the string **between** the word is set to a single space; else if it is contains the string **back**, the word before the cursor is considered, else the word after cursor is considered. Some examples are given below.

The style **skip−whitespace−first** is only used with the **forward−word** widget. If it is set to true, then **forward−word** skips any non−word−characters, followed by any non−word−characters: this is similar to the behaviour of other word−orientated widgets, and also that used by other editors, however it differs from the standard zsh behaviour. When using **select−word−style** the widget is set in the context **:zle:*** to **true** if the word style is **bash** and **false** otherwise. It may be overridden by setting it in the more specific context **:zle:forward−word***.

It is possible to create widgets with specific behaviour by defining a new widget implemented by the appropriate generic function, then setting a style for the context of the specific widget. For example, the following defines a widget **backward−kill−space−word** using **backward−kill−word−match**, the generic widget implementing **backward−kill−word** behaviour, and ensures that the new widget always implements space−delimited behaviour.

```
zle −N backward−kill−space−word backward−kill−word−match
zstyle :zle:backward−kill−space−word word−style space
```

The widget **backward−kill−space−word** can now be bound to a key.

Here are some further examples of use of the styles, actually taken from the simplified interface in **select−word−style**:

```
zstyle ':zle:*' word−style standard
zstyle ':zle:*' word−chars ''
```

Implements bash−style word handling for all widgets, i.e. only alphanumerics are word characters; equivalent to setting the parameter **WORDCHARS** empty for the given context.

```
style ':zle:*kill*' word-style space
```

Uses space−delimited words for widgets with the word 'kill' in the name. Neither of the styles **word−chars** nor **word−class** is used in this case.

Here are some examples of use of the **word−context** style to extend the context.

```
zstyle ':zle:*' word-context \
    "*/*" filename "[[:space:]]" whitespace
zstyle ':zle:transpose-words:whitespace' word-style shell
zstyle ':zle:transpose-words:filename' word-style normal
zstyle ':zle:transpose-words:filename' word-chars ''
```

This provides two different ways of using **transpose−words** depending on whether the cursor is on whitespace between words or on a filename, here any word containing a **/**. On whitespace, complete arguments as defined by standard shell rules will be transposed. In a filename, only alphanumerics will be transposed. Elsewhere, words will be transposed using the default style for **:zle:transpose−words**.

The word matching and all the handling of **zstyle** settings is actually implemented by the function **match−words−by−style**. This can be used to create new user−defined widgets. The calling function should set the local parameter **curcontext** to **:zle:***widget*, create the local parameter **matched_words** and call **match−words−by−style** with no arguments. On return, **matched_words** will be set to an array with the elements: (1) the start of the line (2) the word before the cursor (3) any non−word characters between that word and the cursor (4) any non−word character at the cursor position plus any remaining non−word characters before the next word, including all characters specified by the **skip−chars** style, (5) the word at or following the cursor (6) any non−word characters following that word (7) the remainder of the line. Any of the elements may be an empty string; the calling function should test for this to decide whether it can perform its function.

If the variable **matched_words** is defined by the caller to **match−words−by−style** as an associative array (**local −A matched_words**), then the seven values given above should be retrieved from it as elements named **start**, **word−before−cursor**, **ws−before−cursor**, **ws−after−cursor**, **word−after−cursor**, **ws−after−word**, and **end**. In addition the element **is−word−start** is 1 if the cursor is on the start of a word or subword, or on white space before it (the cases can be distinguished by testing the **ws−after−cursor** element) and 0 otherwise. This form is recommended for future compatibility.

It is possible to pass options with arguments to **match−words−by−style** to override the use of styles. The options are:

−w	*word−style*
−s	*skip−chars*
−c	*word−class*
−C	*word−chars*
−r	*subword−range*

For example, **match−words−by−style −w shell −c 0** may be used to extract the command argument around the cursor.

The **word−context** style is implemented by the function **match−word−context**. This should not usually need to be called directly.

bracketed−paste−magic

The **bracketed−paste** widget (see subsection Miscellaneous in *zshzle*(1)) inserts pasted text literally into the editor buffer rather than interpret it as keystrokes. This disables some common usages where the self−insert widget is replaced in order to accomplish some extra processing. An example is the contributed **url−quote−magic** widget described below.

The **bracketed−paste−magic** widget is meant to replace **bracketed−paste** with a wrapper that re−enables these self−insert actions, and other actions as selected by zstyles. Therefore this widget is installed with

```
autoload −Uz bracketed−paste−magic
zle −N bracketed−paste bracketed−paste−magic
```

Other than enabling some widget processing, **bracketed−paste−magic** attempts to replicate **bracketed−paste** as faithfully as possible.

The following zstyles may be set to control processing of pasted text. All are looked up in the context '**:bracketed−paste−magic**'.

active−widgets
: A list of patterns matching widget names that should be activated during the paste. All other key sequences are processed as self−insert−unmeta. The default is '**self−***' so any user−defined widgets named with that prefix are active along with the builtin self−insert.

  If this style is not set (explicitly deleted) or set to an empty value, no widgets are active and the pasted text is inserted literally. If the value includes '**undefined−key**', any unknown sequences are discarded from the pasted text.

inactive−keys
: The inverse of **active−widgets**, a list of key sequences that always use **self−insert−unmeta** even when bound to an active widget. Note that this is a list of literal key sequences, not patterns.

paste−init
: A list of function names, called in widget context (but not as widgets). The functions are called in order until one of them returns a non−zero status. The parameter '**PASTED**' contains the initial state of the pasted text. All other ZLE parameters such as '**BUFFER**' have their normal values and side−effects, and full history is available, so for example **paste−init** functions may move words from **BUFFER** into **PASTED** to make those words visible to the **active−widgets**.

  A non−zero return from a **paste−init** function does *not* prevent the paste itself from proceeding.

  Loading **bracketed−paste−magic** defines **backward−extend−paste**, a helper function for use in **paste−init**.

  ```
  zstyle :bracketed−paste−magic paste−init \
      backward−extend−paste
  ```

  When a paste would insert into the middle of a word or append text to a word already on the line, **backward−extend−paste** moves the prefix from **LBUFFER** into **PASTED** so that the **active−widgets** see the full word so far. This may be useful with **url−quote−magic**.

paste−finish
: Another list of function names called in order until one returns non−zero. These functions are called *after* the pasted text has been processed by the **active−widgets**, but *before* it is inserted into '**BUFFER**'. ZLE parameters have their normal values and side−effects.

  A non−zero return from a **paste−finish** function does *not* prevent the paste itself from proceeding.

  Loading **bracketed−paste−magic** also defines **quote−paste**, a helper function for use in **paste−finish**.

  ```
  zstyle :bracketed−paste−magic paste−finish \
      quote−paste
  zstyle :bracketed−paste−magic:finish quote−style \
      qqq
  ```

When the pasted text is inserted into **BUFFER**, it is quoted per the **quote−style** value. To forcibly turn off the built−in numeric prefix quoting of **bracketed−paste**, use:

```
zstyle :bracketed−paste−magic:finish quote−style \
    none
```

Important: During **active−widgets** processing of the paste (after **paste−init** and before **paste−finish**), **BUFFER** starts empty and history is restricted, so cursor motions, etc., may not pass outside of the pasted content. Text assigned to **BUFFER** by the active widgets is copied back into **PASTED** before **paste−finish**.

copy−earlier−word

This widget works like a combination of **insert−last−word** and **copy−prev−shell−word**. Repeated invocations of the widget retrieve earlier words on the relevant history line. With a numeric argument *N*, insert the *N*th word from the history line; *N* may be negative to count from the end of the line.

If **insert−last−word** has been used to retrieve the last word on a previous history line, repeated invocations will replace that word with earlier words from the same line.

Otherwise, the widget applies to words on the line currently being edited. The **widget** style can be set to the name of another widget that should be called to retrieve words. This widget must accept the same three arguments as **insert−last−word**.

cycle−completion−positions

After inserting an unambiguous string into the command line, the new function based completion system may know about multiple places in this string where characters are missing or differ from at least one of the possible matches. It will then place the cursor on the position it considers to be the most interesting one, i.e. the one where one can disambiguate between as many matches as possible with as little typing as possible.

This widget allows the cursor to be easily moved to the other interesting spots. It can be invoked repeatedly to cycle between all positions reported by the completion system.

delete−whole−word−match

This is another function which works like the **−match** functions described immediately above, i.e. using styles to decide the word boundaries. However, it is not a replacement for any existing function.

The basic behaviour is to delete the word around the cursor. There is no numeric argument handling; only the single word around the cursor is considered. If the widget contains the string **kill**, the removed text will be placed in the cutbuffer for future yanking. This can be obtained by defining **kill−whole−word−match** as follows:

```
zle −N kill−whole−word−match delete−whole−word−match
```

and then binding the widget **kill−whole−word−match**.

up−line−or−beginning−search, **down−line−or−beginning−search**

These widgets are similar to the builtin functions **up−line−or−search** and **down−line−or−search**: if in a multiline buffer they move up or down within the buffer, otherwise they search for a history line matching the start of the current line. In this case, however, they search for a line which matches the current line up to the current cursor position, in the manner of **history−beginning−search−backward** and **−forward**, rather than the first word on the line.

edit−command−line

Edit the command line using your visual editor, as in **ksh**.

```
bindkey −M vicmd v edit−command−line
```

expand−absolute−path

Expand the file name under the cursor to an absolute path, resolving symbolic links. Where possible, the initial path segment is turned into a named directory or reference to a user's home directory.

history−search−end

This function implements the widgets **history−beginning−search−backward−end** and **history−beginning−search−forward−end**. These commands work by first calling the corresponding builtin widget (see 'History Control' in *zshzle*(1)) and then moving the cursor to the end of the line. The original cursor position is remembered and restored before calling the builtin widget a second time, so that the same search is repeated to look farther through the history.

Although you **autoload** only one function, the commands to use it are slightly different because it implements two widgets.

```
zle −N history−beginning−search−backward−end \
    history−search−end
zle −N history−beginning−search−forward−end \
    history−search−end
bindkey '\e^P' history−beginning−search−backward−end
bindkey '\e^N' history−beginning−search−forward−end
```

history−beginning−search−menu

This function implements yet another form of history searching. The text before the cursor is used to select lines from the history, as for **history−beginning−search−backward** except that all matches are shown in a numbered menu. Typing the appropriate digits inserts the full history line. Note that leading zeroes must be typed (they are only shown when necessary for removing ambiguity). The entire history is searched; there is no distinction between forwards and backwards.

With a numeric argument, the search is not anchored to the start of the line; the string typed by the use may appear anywhere in the line in the history.

If the widget name contains '**−end**' the cursor is moved to the end of the line inserted. If the widget name contains '**−space**' any space in the text typed is treated as a wildcard and can match anything (hence a leading space is equivalent to giving a numeric argument). Both forms can be combined, for example:

```
zle −N history−beginning−search−menu−space−end \
    history−beginning−search−menu
```

history−pattern−search

The function **history−pattern−search** implements widgets which prompt for a pattern with which to search the history backwards or forwards. The pattern is in the usual zsh format, however the first character may be ^ to anchor the search to the start of the line, and the last character may be **$** to anchor the search to the end of the line. If the search was not anchored to the end of the line the cursor is positioned just after the pattern found.

The commands to create bindable widgets are similar to those in the example immediately above:

```
autoload −U history−pattern−search
zle −N history−pattern−search−backward history−pattern−search
zle −N history−pattern−search−forward history−pattern−search
```

incarg Typing the keystrokes for this widget with the cursor placed on or to the left of an integer causes that integer to be incremented by one. With a numeric argument, the number is incremented by the amount of the argument (decremented if the numeric argument is negative). The shell parameter **incarg** may be set to change the default increment to something other than one.

```
bindkey '^X+' incarg
```

incremental−complete−word

This allows incremental completion of a word. After starting this command, a list of completion choices can be shown after every character you type, which you can delete with **^H** or **DEL**. Pressing return accepts the completion so far and returns you to normal editing (that is, the command line is *not* immediately executed). You can hit **TAB** to do normal completion, **^G** to abort back to the state when you started, and **ˆD** to list the matches.

This works only with the new function based completion system.

```
bindkey 'ˆXi' incremental−complete−word
```

insert−composed−char

This function allows you to compose characters that don't appear on the keyboard to be inserted into the command line. The command is followed by two keys corresponding to ASCII characters (there is no prompt). For accented characters, the two keys are a base character followed by a code for the accent, while for other special characters the two characters together form a mnemonic for the character to be inserted. The two−character codes are a subset of those given by RFC 1345 (see for example **http://www.faqs.org/rfcs/rfc1345.html**).

The function may optionally be followed by up to two characters which replace one or both of the characters read from the keyboard; if both characters are supplied, no input is read. For example, **insert−composed−char a:** can be used within a widget to insert an a with umlaut into the command line. This has the advantages over use of a literal character that it is more portable.

For best results zsh should have been built with support for multibyte characters (configured with **−−enable−multibyte**); however, the function works for the limited range of characters available in single−byte character sets such as ISO−8859−1.

The character is converted into the local representation and inserted into the command line at the cursor position. (The conversion is done within the shell, using whatever facilities the C library provides.) With a numeric argument, the character and its code are previewed in the status line

The function may be run outside zle in which case it prints the character (together with a newline) to standard output. Input is still read from keystrokes.

See **insert−unicode−char** for an alternative way of inserting Unicode characters using their hexadecimal character number.

The set of accented characters is reasonably complete up to Unicode character U+0180, the set of special characters less so. However, it is very sporadic from that point. Adding new characters is easy, however; see the function **define−composed−chars**. Please send any additions to **zsh−workers@zsh.org**.

The codes for the second character when used to accent the first are as follows. Note that not every character can take every accent.

!	Grave.
'	Acute.
>	Circumflex.
?	Tilde. (This is not ˜ as RFC 1345 does not assume that character is present on the keyboard.)
−	Macron. (A horizontal bar over the base character.)
(	Breve. (A shallow dish shape over the base character.)
.	Dot above the base character, or in the case of **i** no dot, or in the case of **L** and **l** a centered dot.
:	Diaeresis (Umlaut).
c	Cedilla.
_	Underline, however there are currently no underlined characters.
/	Stroke through the base character.
"	Double acute (only supported on a few letters).
;	Ogonek. (A little forward facing hook at the bottom right of the character.)
<	Caron. (A little v over the letter.)
0	Circle over the base character.
2	Hook over the base character.
9	Horn over the base character.

The most common characters from the Arabic, Cyrillic, Greek and Hebrew alphabets are available; consult RFC 1345 for the appropriate sequences. In addition, a set of two letter codes not in RFC 1345 are available for the double−width characters corresponding to ASCII characters from **!** to **~** (0x21 to 0x7e) by preceding the character with **^**, for example **^A** for a double−width **A**.

The following other two−character sequences are understood.

ASCII characters
These are already present on most keyboards:

<(	Left square bracket
//	Backslash (solidus)
)>	Right square bracket
(!	Left brace (curly bracket)
!!	Vertical bar (pipe symbol)
!)	Right brace (curly bracket)
'?	Tilde

Special letters
Characters found in various variants of the Latin alphabet:

ss	Eszett (scharfes S)
D−, **d−**	Eth
TH, **th**	Thorn
kk	Kra
'n	'n
NG, **ng**	Ng
OI, **oi**	Oi
yr	yr
ED	ezh

Currency symbols

Ct	Cent
Pd	Pound sterling (also lira and others)
Cu	Currency
Ye	Yen
Eu	Euro (N.B. not in RFC 1345)

Punctuation characters
References to "right" quotes indicate the shape (like a 9 rather than 6) rather than their grammatical use. (For example, a "right" low double quote is used to open quotations in German.)

!I	Inverted exclamation mark
BB	Broken vertical bar
SE	Section
Co	Copyright
−a	Spanish feminine ordinal indicator
<<	Left guillemet
−−	Soft hyphen
Rg	Registered trade mark
PI	Pilcrow (paragraph)
−o	Spanish masculine ordinal indicator
>>	Right guillemet
?I	Inverted question mark
−1	Hyphen
−N	En dash
−M	Em dash
−3	Horizontal bar

:3	Vertical ellipsis
.3	Horizontal midline ellipsis
!2	Double vertical line
=2	Double low line
’6	Left single quote
’9	Right single quote
.9	"Right" low quote
9’	Reversed "right" quote
"6	Left double quote
"9	Right double quote
:9	"Right" low double quote
9"	Reversed "right" double quote
/−	Dagger
/=	Double dagger

Mathematical symbols

DG	Degree
−2, **+−**, **−+**	− sign, +/− sign, −/+ sign
2S	Superscript 2
3S	Superscript 3
1S	Superscript 1
My	Micro
.M	Middle dot
14	Quarter
12	Half
34	Three quarters
***X**	Multiplication
−:	Division
%0	Per mille
FA, **TE**, **/0**	For all, there exists, empty set
dP, **DE**, **NB**	Partial derivative, delta (increment), del (nabla)
(−, **−)**	Element of, contains
***P**, **+Z**	Product, sum
***−**, **Ob**, **Sb**	Asterisk, ring, bullet
RT, **0(**, **00**	Root sign, proportional to, infinity

Other symbols

cS, **cH**, **cD**, **cC**	Card suits: spades, hearts, diamonds, clubs
Md, **M8**, **M2**, **Mb**, **Mx**, **MX**	Musical notation: crotchet (quarter note), quaver (eighth note), semiquavers (sixteenth notes), flag sign, natural sign, sharp sign
Fm, **Ml**	Female, male

Accents on their own

’>	Circumflex (same as caret, ˆ)
’!	Grave (same as backtick, ‘)
’,	Cedilla
’:	Diaeresis (Umlaut)

'm	Macron
''	Acute

insert−files

This function allows you type a file pattern, and see the results of the expansion at each step. When you hit return, all expansions are inserted into the command line.

```
bindkey '^Xf' insert−files
```

insert−unicode−char

When first executed, the user inputs a set of hexadecimal digits. This is terminated with another call to **insert−unicode−char**. The digits are then turned into the corresponding Unicode character. For example, if the widget is bound to **^XU**, the character sequence '**^XU 4 c ^XU**' inserts **L** (Unicode U+004c).

See **insert−composed−char** for a way of inserting characters using a two−character mnemonic.

narrow−to−region [**−p** *pre*] [**−P** *post*]
[**−S** *statepm* | **−R** *statepm* | [**−l** *lbufvar*] [**−r** *rbufvar*]]
[**−n**] [*start end*]
narrow−to−region−invisible

Narrow the editable portion of the buffer to the region between the cursor and the mark, which may be in either order. The region may not be empty.

narrow−to−region may be used as a widget or called as a function from a user−defined widget; by default, the text outside the editable area remains visible. A **recursive−edit** is performed and the original widening status is then restored. Various options and arguments are available when it is called as a function.

The options **−p** *pretext* and **−P** *posttext* may be used to replace the text before and after the display for the duration of the function; either or both may be an empty string.

If the option **−n** is also given, *pretext* or *posttext* will only be inserted if there is text before or after the region respectively which will be made invisible.

Two numeric arguments may be given which will be used instead of the cursor and mark positions.

The option **−S** *statepm* is used to narrow according to the other options while saving the original state in the parameter with name *statepm*, while the option **−R** *statepm* is used to restore the state from the parameter; note in both cases the *name* of the parameter is required. In the second case, other options and arguments are irrelevant. When this method is used, no **recursive−edit** is performed; the calling widget should call this function with the option **−S**, perform its own editing on the command line or pass control to the user via '**zle recursive−edit**', then call this function with the option **−R**. The argument *statepm* must be a suitable name for an ordinary parameter, except that parameters beginning with the prefix **_ntr_** are reserved for use within **narrow−to−region**. Typically the parameter will be local to the calling function.

The options **−l** *lbufvar* and **−r** *rbufvar* may be used to specify parameters where the widget will store the resulting text from the operation. The parameter *lbufvar* will contain **LBUFFER** and *rbufvar* will contain **RBUFFER**. Neither of these two options may be used with **−S** or **−R**.

narrow−to−region−invisible is a simple widget which calls **narrow−to−region** with arguments which replace any text outside the region with '**...**'. It does not take any arguments.

The display is restored (and the widget returns) upon any zle command which would usually cause the line to be accepted or aborted. Hence an additional such command is required to accept or abort the current line.

The return status of both widgets is zero if the line was accepted, else non−zero.

Here is a trivial example of a widget using this feature.

```
local state
```

```
narrow−to−region −p $’Editing restricted region\n’ \
  −P ’’ −S state
zle recursive−edit
narrow−to−region −R state
```

predict−on

This set of functions implements predictive typing using history search. After **predict−on**, typing characters causes the editor to look backward in the history for the first line beginning with what you have typed so far. After **predict−off**, editing returns to normal for the line found. In fact, you often don’t even need to use **predict−off**, because if the line doesn’t match something in the history, adding a key performs standard completion, and then inserts itself if no completions were found. However, editing in the middle of a line is liable to confuse prediction; see the **toggle** style below.

With the function based completion system (which is needed for this), you should be able to type **TAB** at almost any point to advance the cursor to the next ‘‘interesting’’ character position (usually the end of the current word, but sometimes somewhere in the middle of the word). And of course as soon as the entire line is what you want, you can accept with return, without needing to move the cursor to the end first.

The first time **predict−on** is used, it creates several additional widget functions:

delete−backward−and−predict
Replaces the **backward−delete−char** widget. You do not need to bind this yourself.

insert−and−predict
Implements predictive typing by replacing the **self−insert** widget. You do not need to bind this yourself.

predict−off
Turns off predictive typing.

Although you **autoload** only the **predict−on** function, it is necessary to create a keybinding for **predict−off** as well.

```
zle −N predict−on
zle −N predict−off
bindkey ’^X^Z’ predict−on
bindkey ’^Z’ predict−off
```

read−from−minibuffer

This is most useful when called as a function from inside a widget, but will work correctly as a widget in its own right. It prompts for a value below the current command line; a value may be input using all of the standard zle operations (and not merely the restricted set available when executing, for example, **execute−named−cmd**). The value is then returned to the calling function in the parameter **$REPLY** and the editing buffer restored to its previous state. If the read was aborted by a keyboard break (typically **^G**), the function returns status 1 and **$REPLY** is not set.

If one argument is supplied to the function it is taken as a prompt, otherwise ‘**?** ’ is used. If two arguments are supplied, they are the prompt and the initial value of **$LBUFFER**, and if a third argument is given it is the initial value of **$RBUFFER**. This provides a default value and starting cursor placement. Upon return the entire buffer is the value of **$REPLY**.

One option is available: ‘**−k** *num*’ specifies that *num* characters are to be read instead of a whole line. The line editor is not invoked recursively in this case, so depending on the terminal settings the input may not be visible, and only the input keys are placed in **$REPLY**, not the entire buffer. Note that unlike the **read** builtin *num* must be given; there is no default.

The name is a slight misnomer, as in fact the shell’s own minibuffer is not used. Hence it is still possible to call **executed−named−cmd** and similar functions while reading a value.

replace−argument, **replace−argument−edit**

The function **replace−argument** can be used to replace a command line argument in the current command line or, if the current command line is empty, in the last command line executed (the new command line is not executed). Arguments are as delimited by standard shell syntax,

If a numeric argument is given, that specifies the argument to be replaced. 0 means the command name, as in history expansion. A negative numeric argument counts backward from the last word.

If no numeric argument is given, the current argument is replaced; this is the last argument if the previous history line is being used.

The function prompts for a replacement argument.

If the widget contains the string **edit**, for example is defined as

```
zle −N replace−argument−edit replace−argument
```

then the function presents the current value of the argument for editing, otherwise the editing buffer for the replacement is initially empty.

replace−string, **replace−pattern**
replace−string−again, **replace−pattern−again**

The function **replace−string** implements three widgets. If defined under the same name as the function, it prompts for two strings; the first (source) string will be replaced by the second everywhere it occurs in the line editing buffer.

If the widget name contains the word '**pattern**', for example by defining the widget using the command '**zle −N replace−pattern replace−string**', then the matching is performed using zsh patterns. All zsh extended globbing patterns can be used in the source string; note that unlike filename generation the pattern does not need to match an entire word, nor do glob qualifiers have any effect. In addition, the replacement string can contain parameter or command substitutions. Furthermore, a '**&**' in the replacement string will be replaced with the matched source string, and a backquoted digit '**\***N*' will be replaced by the *N*th parenthesised expression matched. The form '**\\{***N***}**' may be used to protect the digit from following digits.

If the widget instead contains the word '**regex**' (or '**regexp**'), then the matching is performed using regular expressions, respecting the setting of the option **RE_MATCH_PCRE** (see the description of the function **regexp−replace** below). The special replacement facilities described above for pattern matching are available.

By default the previous source or replacement string will not be offered for editing. However, this feature can be activated by setting the style **edit−previous** in the context **:zle:***widget* (for example, **:zle:replace−string**) to **true**. In addition, a positive numeric argument forces the previous values to be offered, a negative or zero argument forces them not to be.

The function **replace−string−again** can be used to repeat the previous replacement; no prompting is done. As with **replace−string**, if the name of the widget contains the word '**pattern**' or '**regex**', pattern or regular expression matching is performed, else a literal string replacement. Note that the previous source and replacement text are the same whether pattern, regular expression or string matching is used.

In addition, **replace−string** shows the previous replacement above the prompt, so long as there was one during the current session; if the source string is empty, that replacement will be repeated without the widget prompting for a replacement string.

For example, starting from the line:

```
print This line contains fan and fond
```

and invoking **replace−pattern** with the source string '**f(?)n**' and the replacement string '**c\1r**' produces the not very useful line:

```
print This line contains car and cord
```

The range of the replacement string can be limited by using the **narrow−to−region−invisible** widget. One limitation of the current version is that **undo** will cycle through changes to the replacement and source strings before undoing the replacement itself.

send−invisible

This is similar to read−from−minibuffer in that it may be called as a function from a widget or as a widget of its own, and interactively reads input from the keyboard. However, the input being typed is concealed and a string of asterisks (‘*’) is shown instead. The value is saved in the parameter **$INVISIBLE** to which a reference is inserted into the editing buffer at the restored cursor position. If the read was aborted by a keyboard break (typically **^G**) or another escape from editing such as **push−line**, **$INVISIBLE** is set to empty and the original buffer is restored unchanged.

If one argument is supplied to the function it is taken as a prompt, otherwise ‘**Non−echoed text:** ’ is used (as in emacs). If a second and third argument are supplied they are used to begin and end the reference to **$INVISIBLE** that is inserted into the buffer. The default is to open with **${**, then **INVISIBLE**, and close with **}**, but many other effects are possible.

smart−insert−last−word

This function may replace the **insert−last−word** widget, like so:

```
zle −N insert−last−word smart−insert−last−word
```

With a numeric argument, or when passed command line arguments in a call from another widget, it behaves like **insert−last−word**, except that words in comments are ignored when **INTERACTIVE_COMMENTS** is set.

Otherwise, the rightmost ‘‘interesting’’ word from the previous command is found and inserted. The default definition of ‘‘interesting’’ is that the word contains at least one alphabetic character, slash, or backslash. This definition may be overridden by use of the **match** style. The context used to look up the style is the widget name, so usually the context is **:insert−last−word**. However, you can bind this function to different widgets to use different patterns:

```
zle −N insert−last−assignment smart−insert−last−word
zstyle :insert−last−assignment match '[[:alpha:]][][[:alnum:]]#=*'
bindkey '\e=' insert−last−assignment
```

If no interesting word is found and the **auto−previous** style is set to a true value, the search continues upward through the history. When **auto−previous** is unset or false (the default), the widget must be invoked repeatedly in order to search earlier history lines.

transpose−lines

Only useful with a multi−line editing buffer; the lines here are lines within the current on−screen buffer, not history lines. The effect is similar to the function of the same name in Emacs.

Transpose the current line with the previous line and move the cursor to the start of the next line. Repeating this (which can be done by providing a positive numeric argument) has the effect of moving the line above the cursor down by a number of lines.

With a negative numeric argument, requires two lines above the cursor. These two lines are transposed and the cursor moved to the start of the previous line. Using a numeric argument less than −1 has the effect of moving the line above the cursor up by minus that number of lines.

url−quote−magic

This widget replaces the built−in **self−insert** to make it easier to type URLs as command line arguments. As you type, the input character is analyzed and, if it may need quoting, the current word is checked for a URI scheme. If one is found and the current word is not already in quotes, a backslash is inserted before the input character.

Styles to control quoting behavior:

url−metas

This style is looked up in the context ‘**:url−quote−magic:***scheme*’ (where *scheme* is that of the current URL, e.g. "**ftp**"). The value is a string listing the characters to be treated as globbing metacharacters when appearing in a URL using that scheme. The default is to quote all zsh extended globbing characters, excluding '**<**' and '**>**' but including braces (as in brace expansion). See also **url−seps**.

url−seps
Like **url−metas**, but lists characters that should be considered command separators, redirections, history references, etc. The default is to quote the standard set of shell separators, excluding those that overlap with the extended globbing characters, but including '**<**' and '**>**' and the first character of **$histchars**.

url−globbers
This style is looked up in the context '**:url−quote−magic**'. The values form a list of command names that are expected to do their own globbing on the URL string. This implies that they are aliased to use the '**noglob**' modifier. When the first word on the line matches one of the values *and* the URL refers to a local file (see **url−local−schema**), only the **url−seps** characters are quoted; the **url−metas** are left alone, allowing them to affect command−line parsing, completion, etc. The default values are a literal '**noglob**' plus (when the **zsh/parameter** module is available) any commands aliased to the helper function '**urlglobber**' or its alias '**globurl**'.

url−local−schema
This style is always looked up in the context '**:urlglobber**', even though it is used by both url−quote−magic and urlglobber. The values form a list of URI schema that should be treated as referring to local files by their real local path names, as opposed to files which are specified relative to a web−server−defined document root. The defaults are "**ftp**" and "**file**".

url−other−schema
Like **url−local−schema**, but lists all other URI schema upon which **urlglobber** and **url−quote−magic** should act. If the URI on the command line does not have a scheme appearing either in this list or in **url−local−schema**, it is not magically quoted. The default values are "**http**", "**https**", and "**ftp**". When a scheme appears both here and in **url−local−schema**, it is quoted differently depending on whether the command name appears in **url−globbers**.

Loading **url−quote−magic** also defines a helper function '**urlglobber**' and aliases '**globurl**' to '**noglob urlglobber**'. This function takes a local URL apart, attempts to pattern−match the local file portion of the URL path, and then puts the results back into URL format again.

vi−pipe
This function reads a movement command from the keyboard and then prompts for an external command. The part of the buffer covered by the movement is piped to the external command and then replaced by the command's output. If the movement command is bound to vi−pipe, the current line is used.

The function serves as an example for reading a vi movement command from within a user−defined widget.

which−command
This function is a drop−in replacement for the builtin widget **which−command**. It has enhanced behaviour, in that it correctly detects whether or not the command word needs to be expanded as an alias; if so, it continues tracing the command word from the expanded alias until it reaches the command that will be executed.

The style **whence** is available in the context **:zle:$WIDGET**; this may be set to an array to give the command and options that will be used to investigate the command word found. The default is **whence −c**.

zcalc−auto−insert
This function is useful together with the **zcalc** function described in the section Mathematical Functions. It should be bound to a key representing a binary operator such as '**+**', '**−**', '*****' or '**/**'.

When running in zcalc, if the key occurs at the start of the line or immediately following an open parenthesis, the text **"ans "** is inserted before the representation of the key itself. This allows easy use of the answer from the previous calculation in the current line. The text to be inserted before the symbol typed can be modified by setting the variable **ZCALC_AUTO_INSERT_PREFIX**.

Hence, for example, typing '**+12**' followed by return adds 12 to the previous result.

If zcalc is in RPN mode (**−r** option) the effect of this binding is automatically suppressed as operators alone on a line are meaningful.

When not in zcalc, the key simply inserts the symbol itself.

Utility Functions

These functions are useful in constructing widgets. They should be loaded with '**autoload −U** *function*' and called as indicated from user−defined widgets.

split−shell−arguments

This function splits the line currently being edited into shell arguments and whitespace. The result is stored in the array **reply**. The array contains all the parts of the line in order, starting with any whitespace before the first argument, and finishing with any whitespace after the last argument. Hence (so long as the option **KSH_ARRAYS** is not set) whitespace is given by odd indices in the array and arguments by even indices. Note that no stripping of quotes is done; joining together all the elements of **reply** in order is guaranteed to produce the original line.

The parameter **REPLY** is set to the index of the word in **reply** which contains the character after the cursor, where the first element has index 1. The parameter **REPLY2** is set to the index of the character under the cursor in that word, where the first character has index 1.

Hence **reply**, **REPLY** and **REPLY2** should all be made local to the enclosing function.

See the function **modify−current−argument**, described below, for an example of how to call this function.

modify−current−argument [*expr−using−***$ARG** | *func*]

This function provides a simple method of allowing user−defined widgets to modify the command line argument under the cursor (or immediately to the left of the cursor if the cursor is between arguments).

The argument can be an expression which when evaluated operates on the shell parameter **ARG**, which will have been set to the command line argument under the cursor. The expression should be suitably quoted to prevent it being evaluated too early.

Alternatively, if the argument does not contain the string **ARG**, it is assumed to be a shell function, to which the current command line argument is passed as the only argument. The function should set the variable **REPLY** to the new value for the command line argument. If the function returns non−zero status, so does the calling function.

For example, a user−defined widget containing the following code converts the characters in the argument under the cursor into all upper case:

```
modify−current−argument '${(U)ARG}'
```

The following strips any quoting from the current word (whether backslashes or one of the styles of quotes), and replaces it with single quoting throughout:

```
modify−current−argument '${(qq)${(Q)ARG}}'
```

The following performs directory expansion on the command line argument and replaces it by the absolute path:

```
expand−dir() {
  REPLY=${~1}
  REPLY=${REPLY:a}
}
modify−current−argument expand−dir
```

In practice the function **expand−dir** would probably not be defined within the widget where **modify−current−argument** is called.

Styles

The behavior of several of the above widgets can be controlled by the use of the **zstyle** mechanism. In particular, widgets that interact with the completion system pass along their context to any completions that they invoke.

break−keys
This style is used by the **incremental−complete−word** widget. Its value should be a pattern, and all keys matching this pattern will cause the widget to stop incremental completion without the key having any further effect. Like all styles used directly by **incremental−complete−word**, this style is looked up using the context '**:incremental**'.

completer
The **incremental−complete−word** and **insert−and−predict** widgets set up their top−level context name before calling completion. This allows one to define different sets of completer functions for normal completion and for these widgets. For example, to use completion, approximation and correction for normal completion, completion and correction for incremental completion and only completion for prediction one could use:

```
zstyle ':completion:*' completer \
      _complete _correct _approximate
zstyle ':completion:incremental:*' completer \
      _complete _correct
zstyle ':completion:predict:*' completer \
      _complete
```

It is a good idea to restrict the completers used in prediction, because they may be automatically invoked as you type. The **_list** and **_menu** completers should never be used with prediction. The **_approximate**, **_correct**, **_expand**, and **_match** completers may be used, but be aware that they may change characters anywhere in the word behind the cursor, so you need to watch carefully that the result is what you intended.

cursor The **insert−and−predict** widget uses this style, in the context '**:predict**', to decide where to place the cursor after completion has been tried. Values are:

complete
The cursor is left where it was when completion finished, but only if it is after a character equal to the one just inserted by the user. If it is after another character, this value is the same as '**key**'.

key The cursor is left after the *n*th occurrence of the character just inserted, where *n* is the number of times that character appeared in the word before completion was attempted. In short, this has the effect of leaving the cursor after the character just typed even if the completion code found out that no other characters need to be inserted at that position.

Any other value for this style unconditionally leaves the cursor at the position where the completion code left it.

list When using the **incremental−complete−word** widget, this style says if the matches should be listed on every key press (if they fit on the screen). Use the context prefix '**:completion:incremental**'.

The **insert−and−predict** widget uses this style to decide if the completion should be shown even if there is only one possible completion. This is done if the value of this style is the string **always**. In this case the context is '**:predict**' (*not* '**:completion:predict**').

match This style is used by **smart−insert−last−word** to provide a pattern (using full **EXTENDED_GLOB** syntax) that matches an interesting word. The context is the name of the widget to which **smart−insert−last−word** is bound (see above). The default behavior of **smart−insert−last−word** is equivalent to:

> **zstyle :insert−last−word match '*[[:alpha:]/\\]*'**

However, you might want to include words that contain spaces:

> **zstyle :insert−last−word match '*[[:alpha:][:space:]/\\]*'**

Or include numbers as long as the word is at least two characters long:

> **zstyle :insert−last−word match '*([[:digit:]]?|[[:alpha:]/\\])*'**

The above example causes redirections like "2>" to be included.

prompt
The **incremental−complete−word** widget shows the value of this style in the status line during incremental completion. The string value may contain any of the following substrings in the manner of the **PS1** and other prompt parameters:

%c Replaced by the name of the completer function that generated the matches (without the leading underscore).

%l When the **list** style is set, replaced by '**...**' if the list of matches is too long to fit on the screen and with an empty string otherwise. If the **list** style is 'false' or not set, '**%l**' is always removed.

%n Replaced by the number of matches generated.

%s Replaced by '**−no match−**', '**−no prefix−**', or an empty string if there is no completion matching the word on the line, if the matches have no common prefix different from the word on the line, or if there is such a common prefix, respectively.

%u Replaced by the unambiguous part of all matches, if there is any, and if it is different from the word on the line.

Like '**break−keys**', this uses the '**:incremental**' context.

stop−keys
This style is used by the **incremental−complete−word** widget. Its value is treated similarly to the one for the **break−keys** style (and uses the same context: '**:incremental**'). However, in this case all keys matching the pattern given as its value will stop incremental completion and will then execute their usual function.

toggle This boolean style is used by **predict−on** and its related widgets in the context '**:predict**'. If set to one of the standard 'true' values, predictive typing is automatically toggled off in situations where it is unlikely to be useful, such as when editing a multi−line buffer or after moving into the middle of a line and then deleting a character. The default is to leave prediction turned on until an explicit call to **predict−off**.

verbose
This boolean style is used by **predict−on** and its related widgets in the context '**:predict**'. If set to one of the standard 'true' values, these widgets display a message below the prompt when the predictive state is toggled. This is most useful in combination with the **toggle** style. The default does not display these messages.

widget This style is similar to the **command** style: For widget functions that use **zle** to call other widgets, this style can sometimes be used to override the widget which is called. The context for this style is the name of the calling widget (*not* the name of the calling function, because one function may be bound to multiple widget names).

> **zstyle :copy−earlier−word widget smart−insert−last−word**

Check the documentation for the calling widget or function to determine whether the **widget** style is used.

EXCEPTION HANDLING

Two functions are provided to enable zsh to provide exception handling in a form that should be familiar from other languages.

throw *exception*
: The function **throw** throws the named *exception*. The name is an arbitrary string and is only used by the **throw** and **catch** functions. An exception is for the most part treated the same as a shell error, i.e. an unhandled exception will cause the shell to abort all processing in a function or script and to return to the top level in an interactive shell.

catch *exception−pattern*
: The function **catch** returns status zero if an exception was thrown and the pattern *exception−pattern* matches its name. Otherwise it returns status 1. *exception−pattern* is a standard shell pattern, respecting the current setting of the **EXTENDED_GLOB** option. An alias **catch** is also defined to prevent the argument to the function from matching filenames, so patterns may be used unquoted. Note that as exceptions are not fundamentally different from other shell errors it is possible to catch shell errors by using an empty string as the exception name. The shell variable **CAUGHT** is set by **catch** to the name of the exception caught. It is possible to rethrow an exception by calling the **throw** function again once an exception has been caught.

The functions are designed to be used together with the **always** construct described in *zshmisc*(1). This is important as only this construct provides the required support for exceptions. A typical example is as follows.

```
{
  # "try" block
  # ... nested code here calls "throw MyExcept"
} always {
  # "always" block
  if catch MyExcept; then
    print "Caught exception MyExcept"
  elif catch ''; then
    print "Caught a shell error.  Propagating..."
    throw ''
  fi
  # Other exceptions are not handled but may be caught further
  # up the call stack.
}
```

If all exceptions should be caught, the following idiom might be preferable.

```
{
  # ... nested code here throws an exception
} always {
  if catch *; then
    case $CAUGHT in
      (MyExcept)
      print "Caught my own exception"
      ;;
      (*)
      print "Caught some other exception"
      ;;
    esac
  fi
}
```

In common with exception handling in other languages, the exception may be thrown by code deeply nested inside the 'try' block. However, note that it must be thrown inside the current shell, not in a subshell forked for a pipeline, parenthesised current−shell construct, or some form of command or process substitution.

The system internally uses the shell variable **EXCEPTION** to record the name of the exception between throwing and catching. One drawback of this scheme is that if the exception is not handled the variable

EXCEPTION remains set and may be incorrectly recognised as the name of an exception if a shell error subsequently occurs. Adding **unset EXCEPTION** at the start of the outermost layer of any code that uses exception handling will eliminate this problem.

MIME FUNCTIONS

Three functions are available to provide handling of files recognised by extension, for example to dispatch a file **text.ps** when executed as a command to an appropriate viewer.

zsh−mime−setup [**−fv**] [**−l** [*suffix* ...]]
zsh−mime−handler [**−l**] *command argument* ...

These two functions use the files **~/.mime.types** and **/etc/mime.types**, which associate types and extensions, as well as **~/.mailcap** and **/etc/mailcap** files, which associate types and the programs that handle them. These are provided on many systems with the Multimedia Internet Mail Extensions.

To enable the system, the function **zsh−mime−setup** should be autoloaded and run. This allows files with extensions to be treated as executable; such files be completed by the function completion system. The function **zsh−mime−handler** should not need to be called by the user.

The system works by setting up suffix aliases with '**alias −s**'. Suffix aliases already installed by the user will not be overwritten.

For suffixes defined in lower case, upper case variants will also automatically be handled (e.g. **PDF** is automatically handled if handling for the suffix **pdf** is defined), but not vice versa.

Repeated calls to **zsh−mime−setup** do not override the existing mapping between suffixes and executable files unless the option **−f** is given. Note, however, that this does not override existing suffix aliases assigned to handlers other than **zsh−mime−handler**.

Calling **zsh−mime−setup** with the option **−l** lists the existing mappings without altering them. Suffixes to list (which may contain pattern characters that should be quoted from immediate interpretation on the command line) may be given as additional arguments, otherwise all suffixes are listed.

Calling **zsh−mime−setup** with the option **−v** causes verbose output to be shown during the setup operation.

The system respects the **mailcap** flags **needsterminal** and **copiousoutput**, see *mailcap*(4).

The functions use the following styles, which are defined with the **zstyle** builtin command (see *zshmodules*(1)). They should be defined before **zsh−mime−setup** is run. The contexts used all start with **:mime:**, with additional components in some cases. It is recommended that a trailing ***** (suitably quoted) be appended to style patterns in case the system is extended in future. Some examples are given below.

For files that have multiple suffixes, e.g. **.pdf.gz**, where the context includes the suffix it will be looked up starting with the longest possible suffix until a match for the style is found. For example, if **.pdf.gz** produces a match for the handler, that will be used; otherwise the handler for **.gz** will be used. Note that, owing to the way suffix aliases work, it is always required that there be a handler for the shortest possible suffix, so in this example **.pdf.gz** can only be handled if **.gz** is also handled (though not necessarily in the same way). Alternatively, if no handling for **.gz** on its own is needed, simply adding the command

```
alias −s gz=zsh−mime−handler
```

to the initialisation code is sufficient; **.gz** will not be handled on its own, but may be in combination with other suffixes.

current−shell
If this boolean style is true, the mailcap handler for the context in question is run using the **eval** builtin instead of by starting a new **sh** process. This is more efficient, but may not work in the occasional cases where the mailcap handler uses strict POSIX syntax.

disown If this boolean style is true, mailcap handlers started in the background will be disowned, i.e. not subject to job control within the parent shell. Such handlers nearly always produce their own windows, so the only likely harmful side effect of setting the style is that it becomes harder to kill jobs from within the shell.

execute−as−is
This style gives a list of patterns to be matched against files passed for execution with a handler program. If the file matches the pattern, the entire command line is executed in its current form, with no handler. This is useful for files which might have suffixes but nonetheless be executable in their own right. If the style is not set, the pattern ***(*) *(/)** is used; hence executable files are executed directly and not passed to a handler, and the option **AUTO_CD** may be used to change to directories that happen to have MIME suffixes.

execute−never
This style is useful in combination with **execute−as−is**. It is set to an array of patterns corresponding to full paths to files that should never be treated as executable, even if the file passed to the MIME handler matches **execute−as−is**. This is useful for file systems that don't handle execute permission or that contain executables from another operating system. For example, if **/mnt/windows** is a Windows mount, then

```
zstyle ':mime:*' execute−never '/mnt/windows/*'
```

will ensure that any files found in that area will be executed as MIME types even if they are executable. As this example shows, the complete file name is matched against the pattern, regardless of how the file was passed to the handler. The file is resolved to a full path using the **:P** modifier described in the subsection Modifiers in *zshexpn*(1); this means that symbolic links are resolved where possible, so that links into other file systems behave in the correct fashion.

file−path
Used if the style **find−file−in−path** is true for the same context. Set to an array of directories that are used for searching for the file to be handled; the default is the command path given by the special parameter **path**. The shell option **PATH_DIRS** is respected; if that is set, the appropriate path will be searched even if the name of the file to be handled as it appears on the command line contains a '**/**'. The full context is **:mime:.***suffix***:**, as described for the style **handler**.

find−file−in−path
If set, allows files whose names do not contain absolute paths to be searched for in the command path or the path specified by the **file−path** style. If the file is not found in the path, it is looked for locally (whether or not the current directory is in the path); if it is not found locally, the handler will abort unless the **handle−nonexistent** style is set. Files found in the path are tested as described for the style **execute−as−is**. The full context is **:mime:.***suffix***:**, as described for the style **handler**.

flags Defines flags to go with a handler; the context is as for the **handler** style, and the format is as for the flags in **mailcap**.

handle−nonexistent
By default, arguments that don't correspond to files are not passed to the MIME handler in order to prevent it from intercepting commands found in the path that happen to have suffixes. This style may be set to an array of extended glob patterns for arguments that will be passed to the handler even if they don't exist. If it is not explicitly set it defaults to **[[:alpha:]]#:/*** which allows URLs to be passed to the MIME handler even though they don't exist in that format in the file system. The full context is **:mime:.***suffix***:**, as described for the style **handler**.

handler
Specifies a handler for a suffix; the suffix is given by the context as **:mime:.***suffix***:**, and the format of the handler is exactly that in **mailcap**. Note in particular the '**.**' and trailing

colon to distinguish this use of the context. This overrides any handler specified by the **mailcap** files. If the handler requires a terminal, the **flags** style should be set to include the word **needsterminal**, or if the output is to be displayed through a pager (but not if the handler is itself a pager), it should include **copiousoutput**.

mailcap
A list of files in the format of **~/.mailcap** and **/etc/mailcap** to be read during setup, replacing the default list which consists of those two files. The context is **:mime:**. A **+** in the list will be replaced by the default files.

mailcap−priorities
This style is used to resolve multiple mailcap entries for the same MIME type. It consists of an array of the following elements, in descending order of priority; later entries will be used if earlier entries are unable to resolve the entries being compared. If none of the tests resolve the entries, the first entry encountered is retained.

files The order of files (entries in the **mailcap** style) read. Earlier files are preferred. (Note this does not resolve entries in the same file.)

priority
The priority flag from the mailcap entry. The priority is an integer from 0 to 9 with the default value being 5.

flags The test given by the **mailcap−prio−flags** option is used to resolve entries.

place Later entries are preferred; as the entries are strictly ordered, this test always succeeds.

Note that as this style is handled during initialisation, the context is always **:mime:**, with no discrimination by suffix.

mailcap−prio−flags
This style is used when the keyword **flags** is encountered in the list of tests specified by the **mailcap−priorities** style. It should be set to a list of patterns, each of which is tested against the flags specified in the mailcap entry (in other words, the sets of assignments found with some entries in the mailcap file). Earlier patterns in the list are preferred to later ones, and matched patterns are preferred to unmatched ones.

mime−types
A list of files in the format of **~/.mime.types** and **/etc/mime.types** to be read during setup, replacing the default list which consists of those two files. The context is **:mime:**. A **+** in the list will be replaced by the default files.

never−background
If this boolean style is set, the handler for the given context is always run in the foreground, even if the flags provided in the mailcap entry suggest it need not be (for example, it doesn't require a terminal).

pager If set, will be used instead of **$PAGER** or **more** to handle suffixes where the **copiousoutput** flag is set. The context is as for **handler**, i.e. **:mime:.***suffix***:** for handling a file with the given *suffix*.

Examples:

```
zstyle ':mime:*' mailcap ~/.mailcap /usr/local/etc/mailcap
zstyle ':mime:.txt:' handler less %s
zstyle ':mime:.txt:' flags needsterminal
```

When **zsh−mime−setup** is subsequently run, it will look for **mailcap** entries in the two files given. Files of suffix **.txt** will be handled by running '**less** *file.txt*'. The flag **needsterminal** is set to show that this program must run attached to a terminal.

As there are several steps to dispatching a command, the following should be checked if attempting to execute a file by extension *.ext* does not have the expected effect.

The command '**alias −s** *ext*' should show '**ps=zsh−mime−handler**'. If it shows something else, another suffix alias was already installed and was not overwritten. If it shows nothing, no handler was installed: this is most likely because no handler was found in the **.mime.types** and **mailcap** combination for **.ext** files. In that case, appropriate handling should be added to **~/.mime.types** and **mailcap**.

If the extension is handled by **zsh−mime−handler** but the file is not opened correctly, either the handler defined for the type is incorrect, or the flags associated with it are in appropriate. Running **zsh−mime−setup −l** will show the handler and, if there are any, the flags. A **%s** in the handler is replaced by the file (suitably quoted if necessary). Check that the handler program listed lists and can be run in the way shown. Also check that the flags **needsterminal** or **copiousoutput** are set if the handler needs to be run under a terminal; the second flag is used if the output should be sent to a pager. An example of a suitable **mailcap** entry for such a program is:

```
text/html; /usr/bin/lynx '%s'; needsterminal
```

Running '**zsh−mime−handler −l** *command line*' prints the command line that would be executed, simplified to remove the effect of any flags, and quoted so that the output can be run as a complete zsh command line. This is used by the completion system to decide how to complete after a file handled by **zsh−mime−setup**.

pick−web−browser

This function is separate from the two MIME functions described above and can be assigned directly to a suffix:

```
autoload −U pick−web−browser
alias −s html=pick−web−browser
```

It is provided as an intelligent front end to dispatch a web browser. It may be run as either a function or a shell script. The status 255 is returned if no browser could be started.

Various styles are available to customize the choice of browsers:

browser−style

The value of the style is an array giving preferences in decreasing order for the type of browser to use. The values of elements may be

running

Use a GUI browser that is already running when an X Window display is available. The browsers listed in the **x−browsers** style are tried in order until one is found; if it is, the file will be displayed in that browser, so the user may need to check whether it has appeared. If no running browser is found, one is not started. Browsers other than Firefox, Opera and Konqueror are assumed to understand the Mozilla syntax for opening a URL remotely.

x Start a new GUI browser when an X Window display is available. Search for the availability of one of the browsers listed in the **x−browsers** style and start the first one that is found. No check is made for an already running browser.

tty Start a terminal−based browser. Search for the availability of one of the browsers listed in the **tty−browsers** style and start the first one that is found.

If the style is not set the default **running x tty** is used.

x−browsers

An array in decreasing order of preference of browsers to use when running under the X Window System. The array consists of the command name under which to start the browser. They are looked up in the context **:mime:** (which may be extended in future, so appending '*' is recommended). For example,

```
zstyle ':mime:*' x−browsers opera konqueror firefox
```

specifies that **pick−web−browser** should first look for a running instance of Opera, Konqueror or Firefox, in that order, and if it fails to find any should attempt to start Opera. The default is **firefox mozilla netscape opera konqueror**.

tty−browsers
An array similar to **x−browsers**, except that it gives browsers to use when no X Window display is available. The default is **elinks links lynx**.

command
If it is set this style is used to pick the command used to open a page for a browser. The context is **:mime:browser:new:$browser:** to start a new browser or **:mime:browser:running:$browser:** to open a URL in a browser already running on the current X display, where **$browser** is the value matched in the **x−browsers** or **tty−browsers** style. The escape sequence **%b** in the style's value will be replaced by the browser, while **%u** will be replaced by the URL. If the style is not set, the default for all new instances is equivalent to **%b %u** and the defaults for using running browsers are equivalent to the values **kfmclient openURL %u** for Konqueror, **firefox −new−tab %u** for Firefox, **opera −newpage %u** for Opera, and **%b −remote "openUrl(%u)"** for all others.

MATHEMATICAL FUNCTIONS

zcalc [**−erf**] [*expression* ...]
A reasonably powerful calculator based on zsh's arithmetic evaluation facility. The syntax is similar to that of formulae in most programming languages; see the section 'Arithmetic Evaluation' in *zshmisc*(1) for details.

Non−programmers should note that, as in many other programming languages, expressions involving only integers (whether constants without a '**.**', variables containing such constants as strings, or variables declared to be integers) are by default evaluated using integer arithmetic, which is not how an ordinary desk calculator operates. To force floating point operation, pass the option **−f**; see further notes below.

If the file **~/.zcalcrc** exists it will be sourced inside the function once it is set up and about to process the command line. This can be used, for example, to set shell options; **emulate −L zsh** and **setopt extendedglob** are in effect at this point. Any failure to source the file if it exists is treated as fatal. As with other initialisation files, the directory **$ZDOTDIR** is used instead of **$HOME** if it is set.

The mathematical library **zsh/mathfunc** will be loaded if it is available; see the section 'The zsh/mathfunc Module' in *zshmodules*(1). The mathematical functions correspond to the raw system libraries, so trigonometric functions are evaluated using radians, and so on.

Each line typed is evaluated as an expression. The prompt shows a number, which corresponds to a positional parameter where the result of that calculation is stored. For example, the result of the calculation on the line preceded by '**4>** ' is available as **$4**. The last value calculated is available as **ans**. Full command line editing, including the history of previous calculations, is available; the history is saved in the file **~/.zcalc_history**. To exit, enter a blank line or type '**:q**' on its own ('**q**' is allowed for historical compatibility).

A line ending with a single backslash is treated in the same fashion as it is in command line editing: the backslash is removed, the function prompts for more input (the prompt is preceded by '**...**' to indicate this), and the lines are combined into one to get the final result. In addition, if the input so far contains more open than close parentheses **zcalc** will prompt for more input.

If arguments are given to **zcalc** on start up, they are used to prime the first few positional parameters. A visual indication of this is given when the calculator starts.

The constants **PI** (3.14159...) and **E** (2.71828...) are provided. Parameter assignment is possible, but note that all parameters will be put into the global namespace unless the **:local** special command is used. The function creates local variables whose names start with **_**, so users should avoid doing so. The variables **ans** (the last answer) and **stack** (the stack in RPN mode) may be referred to directly; **stack** is an array but elements of it are numeric. Various other special variables are used locally with their standard meaning, for example **compcontext**, **match**, **mbegin**, **mend**, **psvar**.

The output base can be initialised by passing the option '**−#***base*', for example '**zcalc −#16**' (the '**#**' may have to be quoted, depending on the globbing options set).

If the option '**−e**' is set, the function runs non−interactively: the arguments are treated as expressions to be evaluated as if entered interactively line by line.

If the option '**−f**' is set, all numbers are treated as floating point, hence for example the expression '**3/4**' evaluates to 0.75 rather than 0. Options must appear in separate words.

If the option '**−r**' is set, RPN (Reverse Polish Notation) mode is entered. This has various additional properties:

Stack
: Evaluated values are maintained in a stack; this is contained in an array named **stack** with the most recent value in **${stack[1]}**.

Operators and functions
: If the line entered matches an operator (**+**, **−**, *****, **/**, ******, **^**, **|** or **&**) or a function supplied by the **zsh/mathfunc** library, the bottom element or elements of the stack are popped to use as the argument or arguments. The higher elements of stack (least recent) are used as earlier arguments. The result is then pushed into **${stack[1]}**.

Expressions
: Other expressions are evaluated normally, printed, and added to the stack as numeric values. The syntax within expressions on a single line is normal shell arithmetic (not RPN).

Stack listing
: If an integer follows the option **−r** with no space, then on every evaluation that many elements of the stack, where available, are printed instead of just the most recent result. Hence, for example, **zcalc −r4** shows **$stack[4]** to **$stack[1]** each time results are printed.

Duplication: **=**
: The pseudo−operator **=** causes the most recent element of the stack to be duplicated onto the stack.

pop
: The pseudo−function **pop** causes the most recent element of the stack to be popped. A '**>**' on its own has the same effect.

>*ident*
: The expression **>** followed (with no space) by a shell identifier causes the most recent element of the stack to be popped and assigned to the variable with that name. The variable is local to the **zcalc** function.

<*ident*
: The expression **<** followed (with no space) by a shell identifier causes the value of the variable with that name to be pushed onto the stack. *ident* may be an integer, in which case the previous result with that number (as shown before the **>** in the standard **zcalc** prompt) is put on the stack.

Exchange: **xy**
: The pseudo−function **xy** causes the most recent two elements of the stack to be exchanged. '**<>**' has the same effect.

The prompt is configurable via the parameter **ZCALCPROMPT**, which undergoes standard prompt expansion. The index of the current entry is stored locally in the first element of the array **psvar**, which can be referred to in **ZCALCPROMPT** as '**%1v**'. The default prompt is '**%1v>** '.

The variable **ZCALC_ACTIVE** is set within the function and can be tested by nested functions; it has the value **rpn** if RPN mode is active, else 1.

A few special commands are available; these are introduced by a colon. For backward compatibility, the colon may be omitted for certain commands. Completion is available if **compinit** has been run.

The output precision may be specified within zcalc by special commands familiar from many calculators.

:norm The default output format. It corresponds to the printf **%g** specification. Typically this shows six decimal digits.

:sci *digits*
Scientific notation, corresponding to the printf **%g** output format with the precision given by *digits*. This produces either fixed point or exponential notation depending on the value output.

:fix *digits*
Fixed point notation, corresponding to the printf **%f** output format with the precision given by *digits*.

:eng *digits*
Exponential notation, corresponding to the printf **%E** output format with the precision given by *digits*.

:raw Raw output: this is the default form of the output from a math evaluation. This may show more precision than the number actually possesses.

Other special commands:

:!*line...* Execute *line...* as a normal shell command line. Note that it is executed in the context of the function, i.e. with local variables. Space is optional after **:!**.

:local *arg ...*
Declare variables local to the function. Other variables may be used, too, but they will be taken from or put into the global scope.

:function *name* [*body*]
Define a mathematical function or (with no *body*) delete it. **:function** may be abbreviated to **:func** or simply **:f**. The *name* may contain the same characters as a shell function name. The function is defined using **zmathfuncdef**, see below.

Note that **zcalc** takes care of all quoting. Hence for example:

```
:f cube $1 * $1 * $1
```

defines a function to cube the sole argument. Functions so defined, or indeed any functions defined directly or indirectly using **functions –M**, are available to execute by typing only the name on the line in RPN mode; this pops the appropriate number of arguments off the stack to pass to the function, i.e. 1 in the case of the example **cube** function. If there are optional arguments only the mandatory arguments are supplied by this means.

[**#***base*] This is not a special command, rather part of normal arithmetic syntax; however, when this form appears on a line by itself the default output radix is set to *base*. Use, for example, '**[#16]**' to display hexadecimal output preceded by an indication of the base, or '**[##16]**' just to display the raw number in the given base. Bases themselves are always specified in decimal. '**[#]**' restores the normal output format. Note that setting an output base suppresses floating point output; use '**[#]**' to return to normal operation.

$*var* Print out the value of var literally; does not affect the calculation. To use the value of var, omit the leading '**$**'.

See the comments in the function for a few extra tips.

min(*arg***, ...)**
max(*arg***, ...)**

sum(*arg***, ...)**
zmathfunc

The function **zmathfunc** defines the three mathematical functions **min**, **max**, and **sum**. The functions **min** and **max** take one or more arguments. The function **sum** takes zero or more arguments. Arguments can be of different types (ints and floats).

Not to be confused with the **zsh/mathfunc** module, described in the section 'The zsh/mathfunc Module' in *zshmodules*(1).

zmathfuncdef [*mathfunc* [*body*]]

A convenient front end to **functions −M**.

With two arguments, define a mathematical function named *mathfunc* which can be used in any form of arithmetic evaluation. *body* is a mathematical expression to implement the function. It may contain references to position parameters **$1**, **$2**, ... to refer to mandatory parameters and **${1:−***defvalue***}** ... to refer to optional parameters. Note that the forms must be strictly adhered to for the function to calculate the correct number of arguments. The implementation is held in a shell function named **zsh_math_func_***mathfunc*; usually the user will not need to refer to the shell function directly. Any existing function of the same name is silently replaced.

With one argument, remove the mathematical function *mathfunc* as well as the shell function implementation.

With no arguments, list all *mathfunc* functions in a form suitable for restoring the definition. The functions have not necessarily been defined by **zmathfuncdef**.

USER CONFIGURATION FUNCTIONS

The **zsh/newuser** module comes with a function to aid in configuring shell options for new users. If the module is installed, this function can also be run by hand. It is available even if the module's default behaviour, namely running the function for a new user logging in without startup files, is inhibited.

zsh−newuser−install [**−f**]

The function presents the user with various options for customizing their initialization scripts. Currently only **~/.zshrc** is handled. **$ZDOTDIR/.zshrc** is used instead if the parameter **ZDOTDIR** is set; this provides a way for the user to configure a file without altering an existing **.zshrc**.

By default the function exits immediately if it finds any of the files **.zshenv**, **.zprofile**, **.zshrc**, or **.zlogin** in the appropriate directory. The option **−f** is required in order to force the function to continue. Note this may happen even if **.zshrc** itself does not exist.

As currently configured, the function will exit immediately if the user has root privileges; this behaviour cannot be overridden.

Once activated, the function's behaviour is supposed to be self−explanatory. Menus are present allowing the user to alter the value of options and parameters. Suggestions for improvements are always welcome.

When the script exits, the user is given the opportunity to save the new file or not; changes are not irreversible until this point. However, the script is careful to restrict changes to the file only to a group marked by the lines '**# Lines configured by zsh−newuser−install**' and '**# End of lines configured by zsh−newuser−install**'. In addition, the old version of **.zshrc** is saved to a file with the suffix **.zni** appended.

If the function edits an existing **.zshrc**, it is up to the user to ensure that the changes made will take effect. For example, if control usually returns early from the existing **.zshrc** the lines will not be executed; or a later initialization file may override options or parameters, and so on. The function itself does not attempt to detect any such conflicts.

OTHER FUNCTIONS

There are a large number of helpful functions in the **Functions/Misc** directory of the zsh distribution. Most are very simple and do not require documentation here, but a few are worthy of special mention.

Descriptions

colors This function initializes several associative arrays to map color names to (and from) the ANSI standard eight−color terminal codes. These are used by the prompt theme system (see above). You seldom should need to run **colors** more than once.

The eight base colors are: **black**, **red**, **green**, **yellow**, **blue**, **magenta**, **cyan**, and **white**. Each of these has codes for foreground and background. In addition there are seven intensity attributes: **bold**, **faint**, **standout**, **underline**, **blink**, **reverse**, and **conceal**. Finally, there are seven codes used to negate attributes: **none** (reset all attributes to the defaults), **normal** (neither bold nor faint), **no−standout**, **no−underline**, **no−blink**, **no−reverse**, and **no−conceal**.

Some terminals do not support all combinations of colors and intensities.

The associative arrays are:

color
colour Map all the color names to their integer codes, and integer codes to the color names. The eight base names map to the foreground color codes, as do names prefixed with '**fg−**', such as '**fg−red**'. Names prefixed with '**bg−**', such as '**bg−blue**', refer to the background codes. The reverse mapping from code to color yields base name for foreground codes and the **bg−** form for backgrounds.

Although it is a misnomer to call them 'colors', these arrays also map the other fourteen attributes from names to codes and codes to names.

fg
fg_bold
fg_no_bold
Map the eight basic color names to ANSI terminal escape sequences that set the corresponding foreground text properties. The **fg** sequences change the color without changing the eight intensity attributes.

bg
bg_bold
bg_no_bold
Map the eight basic color names to ANSI terminal escape sequences that set the corresponding background properties. The **bg** sequences change the color without changing the eight intensity attributes.

In addition, the scalar parameters **reset_color** and **bold_color** are set to the ANSI terminal escapes that turn off all attributes and turn on bold intensity, respectively.

fned [**−x** *num*] *name*
Same as **zed −f**. This function does not appear in the zsh distribution, but can be created by linking **zed** to the name **fned** in some directory in your **fpath**.

is−at−least *needed* [*present*]
Perform a greater−than−or−equal−to comparison of two strings having the format of a zsh version number; that is, a string of numbers and text with segments separated by dots or dashes. If the *present* string is not provided, **$ZSH_VERSION** is used. Segments are paired left−to−right in the two strings with leading non−number parts ignored. If one string has fewer segments than the other, the missing segments are considered zero.

This is useful in startup files to set options and other state that are not available in all versions of zsh.

```
is-at-least 3.1.6-15 && setopt NO_GLOBAL_RCS
is-at-least 3.1.0 && setopt HIST_REDUCE_BLANKS
is-at-least 2.6-17 || print "You can't use is-at-least here."
```

nslookup [*arg* ...]

This wrapper function for the **nslookup** command requires the **zsh/zpty** module (see *zshmodules*(1)). It behaves exactly like the standard **nslookup** except that it provides customizable prompts (including a right−side prompt) and completion of nslookup commands, host names, etc. (if you use the function−based completion system). Completion styles may be set with the context prefix '**:completion:nslookup**'.

See also the **pager**, **prompt** and **rprompt** styles below.

regexp−replace *var regexp replace*

Use regular expressions to perform a global search and replace operation on a variable. POSIX extended regular expressions are used, unless the option **RE_MATCH_PCRE** has been set, in which case Perl−compatible regular expressions are used (this requires the shell to be linked against the **pcre** library).

var is the name of the variable containing the string to be matched. The variable will be modified directly by the function. The variables **MATCH**, **MBEGIN**, **MEND**, **match**, **mbegin**, **mend** should be avoided as these are used by the regular expression code.

regexp is the regular expression to match against the string.

replace is the replacement text. This can contain parameter, command and arithmetic expressions which will be replaced: in particular, a reference to **$MATCH** will be replaced by the text matched by the pattern.

The return status is 0 if at least one match was performed, else 1.

run−help *cmd*

This function is designed to be invoked by the **run−help** ZLE widget, in place of the default alias. See 'Accessing On−Line Help' above for setup instructions.

In the discussion which follows, if *cmd* is a file system path, it is first reduced to its rightmost component (the file name).

Help is first sought by looking for a file named *cmd* in the directory named by the **HELPDIR** parameter. If no file is found, an assistant function, alias, or command named **run−help−***cmd is sought. If found, the assistant is executed with the rest of the current command line (everything after the command name cmd) as its arguments. When neither file nor assistant is found, the external command '***man** *cmd' is run.*

An example assistant for the "ssh" command:

```
run−help−ssh() {
  emulate −LR zsh
  local −a args
  # Delete the "−l username" option
  zparseopts −D −E −a args l:
  # Delete other options, leaving: host command
  args=(${@:#−*})
  if [[ ${#args} −lt 2 ]]; then
    man ssh
  else
    run−help $args[2]
  fi
}
```

Several of these assistants are provided in the **Functions/Misc** directory. These must be autoloaded, or placed as executable scripts in your search path, in order to be found and used by **run−help**.

run−help−git
run−help−ip
run−help−openssl
run−help−p4
run−help−sudo
run−help−svk
run−help−svn
Assistant functions for the **git**, **ip**, **openssl**, **p4**, **sudo**, **svk**, and **svn**, commands.

tetris Zsh was once accused of not being as complete as Emacs, because it lacked a Tetris game. This function was written to refute this vicious slander.

This function must be used as a ZLE widget:

autoload −U tetris
zle −N tetris
bindkey *keys* **tetris**

To start a game, execute the widget by typing the *keys*. Whatever command line you were editing disappears temporarily, and your keymap is also temporarily replaced by the Tetris control keys. The previous editor state is restored when you quit the game (by pressing '**q**') or when you lose.

If you quit in the middle of a game, the next invocation of the **tetris** widget will continue where you left off. If you lost, it will start a new game.

tetriscurses
This is a port of the above to zcurses. The input handling is improved a bit so that moving a block sideways doesn't automatically advance a timestep, and the graphics use unicode block graphics.

This version does not save the game state between invocations, and is not invoked as a widget, but rather as:

autoload −U tetriscurses
tetriscurses

zargs [*option* ... **−−**] [*input* ...] [**−−** *command* [*arg* ...]]
This function has a similar purpose to GNU xargs. Instead of reading lines of arguments from the standard input, it takes them from the command line. This is useful because zsh, especially with recursive glob operators, often can construct a command line for a shell function that is longer than can be accepted by an external command.

The *option* list represents options of the **zargs** command itself, which are the same as those of **xargs**. The *input* list is the collection of strings (often file names) that become the arguments of the **command**, analogous to the standard input of **xargs**. Finally, the *arg* list consists of those arguments (usually options) that are passed to the *command* each time it runs. The *arg* list precedes the elements from the **input** list in each run. If no *command* is provided, then no *arg* list may be provided, and in that event the default command is '**print**' with arguments '**−r −−**'.

For example, to get a long **ls** listing of all non−hidden plain files in the current directory or its subdirectories:

autoload −U zargs
zargs −− **/*(.) −− ls −ld −−

The first and third occurrences of '**−−**' are used to mark the end of options for **zargs** and **ls** respectively to guard against filenames starting with '**−**', while the second is used to separate the list of files from the command to run ('**ls −ld −−**').

The first '**−−**' would also be needed if there was a chance the list might be empty as in:

zargs −r −− ./*.back(#qN) −− rm −f

In the event that the string '**−−**' is or may be an *input*, the **−e** option may be used to change the end−of−inputs marker. Note that this does *not* change the end−of−options marker. For example, to use '**..**' as the marker:

> **zargs −e.. −− **/*(.) .. ls −ld −−**

This is a good choice in that example because no plain file can be named '**..**', but the best end−marker depends on the circumstances.

The options **−i**, **−I**, **−l**, **−L**, and **−n** differ slightly from their usage in **xargs**. There are no input lines for **zargs** to count, so **−l** and **−L** count through the *input* list, and **−n** counts the number of arguments passed to each execution of *command*, *including* any *arg* list. Also, any time **−i** or **−I** is used, each *input* is processed separately as if by '**−L 1**'.

For details of the other **zargs** options, see *xargs*(1) (but note the difference in function between **zargs** and **xargs**) or run **zargs** with the **−−help** option.

zed [**−f** [**−x** *num*]] *name*
zed −b This function uses the ZLE editor to edit a file or function.

Only one *name* argument is allowed. If the **−f** option is given, the name is taken to be that of a function; if the function is marked for autoloading, **zed** searches for it in the **fpath** and loads it. Note that functions edited this way are installed into the current shell, but *not* written back to the autoload file. In this case the **−x** option specifies that leading tabs indenting the function according to syntax should be converted into the given number of spaces; '**−x 2**' is consistent with the layout of functions distributed with the shell.

Without **−f**, *name* is the path name of the file to edit, which need not exist; it is created on write, if necessary.

While editing, the function sets the main keymap to **zed** and the vi command keymap to **zed−vicmd**. These will be copied from the existing **main** and **vicmd** keymaps if they do not exist the first time **zed** is run. They can be used to provide special key bindings used only in zed.

If it creates the keymap, **zed** rebinds the return key to insert a line break and '**^X^W**' to accept the edit in the **zed** keymap, and binds '**ZZ**' to accept the edit in the **zed−vicmd** keymap.

The bindings alone can be installed by running '**zed −b**'. This is suitable for putting into a startup file. Note that, if rerun, this will overwrite the existing **zed** and **zed−vicmd** keymaps.

Completion is available, and styles may be set with the context prefix '**:completion:zed**'.

A zle widget **zed−set−file−name** is available. This can be called by name from within zed using '**\ex zed−set−file−name**' (note, however, that because of zed's rebindings you will have to type **^j** at the end instead of the return key), or can be bound to a key in either of the **zed** or **zed−vicmd** keymaps after '**zed −b**' has been run. When the widget is called, it prompts for a new name for the file being edited. When zed exits the file will be written under that name and the original file will be left alone. The widget has no effect with '**zed −f**'.

While **zed−set−file−name** is running, zed uses the keymap **zed−normal−keymap**, which is linked from the main keymap in effect at the time zed initialised its bindings. (This is to make the return key operate normally.) The result is that if the main keymap has been changed, the widget won't notice. This is not a concern for most users.

zcp [**−finqQvwW**] *srcpat dest*
zln [**−finqQsvwW**] *srcpat dest*
Same as **zmv −C** and **zmv −L**, respectively. These functions do not appear in the zsh distribution, but can be created by linking **zmv** to the names **zcp** and **zln** in some directory in your **fpath**.

zkbd See 'Keyboard Definition' above.

zmv [**−finqQsvwW**] [**−C** | **−L** | **−M** | −{**p**|**P**} *program*] [**−o** *optstring*]
srcpat dest
Move (usually, rename) files matching the pattern *srcpat* to corresponding files having names of the form given by *dest*, where *srcpat* contains parentheses surrounding patterns which will be

replaced in turn by **$1**, **$2**, ... in *dest*. For example,

```
zmv '(*).lis' '$1.txt'
```

renames '**foo.lis**' to '**foo.txt**', '**my.old.stuff.lis**' to '**my.old.stuff.txt**', and so on.

The pattern is always treated as an **EXTENDED_GLOB** pattern. Any file whose name is not changed by the substitution is simply ignored. Any error (a substitution resulted in an empty string, two substitutions gave the same result, the destination was an existing regular file and **−f** was not given) causes the entire function to abort without doing anything.

In addition to pattern replacement, the variable **$f** can be referrred to in the second (replacement) argument. This makes it possible to use variable substitution to alter the argument; see examples below.

Options:

−f Force overwriting of destination files. Not currently passed down to the **mv/cp/ln** command due to vagaries of implementations (but you can use **−o−f** to do that).

−i Interactive: show each line to be executed and ask the user whether to execute it. '**Y**' or '**y**' will execute it, anything else will skip it. Note that you just need to type one character.

−n No execution: print what would happen, but don't do it.

−q Turn bare glob qualifiers off: now assumed by default, so this has no effect.

−Q Force bare glob qualifiers on. Don't turn this on unless you are actually using glob qualifiers in a pattern.

−s Symbolic, passed down to **ln**; only works with **−L**.

−v Verbose: print each command as it's being executed.

−w Pick out wildcard parts of the pattern, as described above, and implicitly add parentheses for referring to them.

−W Just like **−w**, with the addition of turning wildcards in the replacement pattern into sequential **${1}** .. **${N}** references.

−C
−L
−M Force **cp**, **ln** or **mv**, respectively, regardless of the name of the function.

−p *program*
Call *program* instead of **cp**, **ln** or **mv**. Whatever it does, it should at least understand the form '*program* **−−** *oldname newname*' where *oldname* and *newname* are filenames generated by **zmv**. *program* will be split into words, so might be e.g. the name of an archive tool plus a copy or rename subcommand.

−P *program*
As **−p** *program*, except that *program* does not accept a following **−−** to indicate the end of options. In this case filenames must already be in a sane form for the program in question.

−o *optstring*
The *optstring* is split into words and passed down verbatim to the **cp**, **ln** or **mv** command called to perform the work. It should probably begin with a '**−**'.

Further examples:

```
zmv −v '(* *)' '${1// /_}'
```

For any file in the current directory with at least one space in the name, replace every space by an underscore and display the commands executed.

```
zmv −v '* *' '${f// /_}'
```

This does exactly the same by referring to the file name stored in **$f**.

For more complete examples and other implementation details, see the **zmv** source file, usually located in one of the directories named in your **fpath**, or in **Functions/Misc/zmv** in the zsh distribution.

zrecompile
See 'Recompiling Functions' above.

zstyle+ *context style value* [**+** *subcontext style value* ...]
This makes defining styles a bit simpler by using a single '**+**' as a special token that allows you to append a context name to the previously used context name. Like this:

```
zstyle+ ':foo:bar' style1 value1 \
      +':baz'     style2 value2 \
      +':frob'    style3 value3
```

This defines *style1* with *value1* for the context **:foo:bar** as usual, but it also defines *style2* with *value2* for the context **:foo:bar:baz** and *style3* with *value3* for **:foo:bar:frob**. Any *subcontext* may be the empty string to re−use the first context unchanged.

Styles

insert−tab
The **zed** function *sets* this style in context '**:completion:zed:***' to turn off completion when **TAB** is typed at the beginning of a line. You may override this by setting your own value for this context and style.

pager The **nslookup** function looks up this style in the context '**:nslookup**' to determine the program used to display output that does not fit on a single screen.

prompt
rprompt
The **nslookup** function looks up this style in the context '**:nslookup**' to set the prompt and the right−side prompt, respectively. The usual expansions for the **PS1** and **RPS1** parameters may be used (see EXPANSION OF PROMPT SEQUENCES in *zshmisc*(1)).