NAME

zshmisc − everything and then some

SIMPLE COMMANDS & PIPELINES

A *simple command* is a sequence of optional parameter assignments followed by blank−separated words, with optional redirections interspersed. For a description of assignment, see the beginning of *zshparam*(1).

The first word is the command to be executed, and the remaining words, if any, are arguments to the command. If a command name is given, the parameter assignments modify the environment of the command when it is executed. The value of a simple command is its exit status, or 128 plus the signal number if terminated by a signal. For example,

echo foo

is a simple command with arguments.

A *pipeline* is either a simple command, or a sequence of two or more simple commands where each command is separated from the next by '**|**' or '**|&**'. Where commands are separated by '**|**', the standard output of the first command is connected to the standard input of the next. '**|&**' is shorthand for '**2>&1 |**', which connects both the standard output and the standard error of the command to the standard input of the next. The value of a pipeline is the value of the last command, unless the pipeline is preceded by '**!**' in which case the value is the logical inverse of the value of the last command. For example,

echo foo | sed 's/foo/bar/'

is a pipeline, where the output ('**foo**' plus a newline) of the first command will be passed to the input of the second.

If a pipeline is preceded by '**coproc**', it is executed as a coprocess; a two−way pipe is established between it and the parent shell. The shell can read from or write to the coprocess by means of the '**>&p**' and '**<&p**' redirection operators or with '**print −p**' and '**read −p**'. A pipeline cannot be preceded by both '**coproc**' and '**!**'. If job control is active, the coprocess can be treated in other than input and output as an ordinary background job.

A *sublist* is either a single pipeline, or a sequence of two or more pipelines separated by '**&&**' or '**||**'. If two pipelines are separated by '**&&**', the second pipeline is executed only if the first succeeds (returns a zero status). If two pipelines are separated by '**||**', the second is executed only if the first fails (returns a nonzero status). Both operators have equal precedence and are left associative. The value of the sublist is the value of the last pipeline executed. For example,

dmesg | grep panic && print yes

is a sublist consisting of two pipelines, the second just a simple command which will be executed if and only if the **grep** command returns a zero status. If it does not, the value of the sublist is that return status, else it is the status returned by the **print** (almost certainly zero).

A *list* is a sequence of zero or more sublists, in which each sublist is terminated by '**;**', '**&**', '**&|**', '**&!**', or a newline. This terminator may optionally be omitted from the last sublist in the list when the list appears as a complex command inside '**(...)**' or '**{...}**'. When a sublist is terminated by '**;**' or newline, the shell waits for it to finish before executing the next sublist. If a sublist is terminated by a '**&**', '**&|**', or '**&!**', the shell executes the last pipeline in it in the background, and does not wait for it to finish (note the difference from other shells which execute the whole sublist in the background). A backgrounded pipeline returns a status of zero.

More generally, a list can be seen as a set of any shell commands whatsoever, including the complex commands below; this is implied wherever the word 'list' appears in later descriptions. For example, the commands in a shell function form a special sort of list.

PRECOMMAND MODIFIERS

A simple command may be preceded by a *precommand modifier*, which will alter how the command is interpreted. These modifiers are shell builtin commands with the exception of **nocorrect** which is a reserved word.

− The command is executed with a '**−**' prepended to its **argv[0]** string.

builtin The command word is taken to be the name of a builtin command, rather than a shell function or external command.

command [**−pvV**]
The command word is taken to be the name of an external command, rather than a shell function or builtin. If the **POSIX_BUILTINS** option is set, builtins will also be executed but certain special properties of them are suppressed. The **−p** flag causes a default path to be searched instead of that in **$path**. With the **−v** flag, **command** is similar to **whence** and with **−V**, it is equivalent to **whence −v**.

exec [**−cl**] [**−a** *argv0*]
The following command together with any arguments is run in place of the current process, rather than as a sub−process. The shell does not fork and is replaced. The shell does not invoke **TRAPEXIT**, nor does it source **zlogout** files. The options are provided for compatibility with other shells.

The **−c** option clears the environment.

The **−l** option is equivalent to the **−** precommand modifier, to treat the replacement command as a login shell; the command is executed with a **−** prepended to its **argv[0]** string. This flag has no effect if used together with the **−a** option.

The **−a** option is used to specify explicitly the **argv[0]** string (the name of the command as seen by the process itself) to be used by the replacement command and is directly equivalent to setting a value for the **ARGV0** environment variable.

nocorrect
Spelling correction is not done on any of the words. This must appear before any other precommand modifier, as it is interpreted immediately, before any parsing is done. It has no effect in non−interactive shells.

noglob Filename generation (globbing) is not performed on any of the words.

COMPLEX COMMANDS

A *complex command* in zsh is one of the following:

if *list* **then** *list* [**elif** *list* **then** *list*] ... [**else** *list*] **fi**
The **if** *list* is executed, and if it returns a zero exit status, the **then** *list* is executed. Otherwise, the **elif** *list* is executed and if its status is zero, the **then** *list* is executed. If each **elif** *list* returns nonzero status, the **else** *list* is executed.

for *name* ... [**in** *word* ...] *term* **do** *list* **done**
Expand the list of *word*s, and set the parameter *name* to each of them in turn, executing *list* each time. If the '**in** *word*' is omitted, use the positional parameters instead of the *word*s.

The *term* consists of one or more newline or **;** which terminate the *word*s, and are optional when the '**in** *word*' is omitted.

More than one parameter *name* can appear before the list of *word*s. If *N name*s are given, then on each execution of the loop the next *N word*s are assigned to the corresponding parameters. If there are more *name*s than remaining *word*s, the remaining parameters are each set to the empty string. Execution of the loop ends when there is no remaining *word* to assign to the first *name*. It is only possible for **in** to appear as the first *name* in the list, else it will be treated as marking the end of the list.

for (([*expr1*] **;** [*expr2*] **;** [*expr3*] **)) do** *list* **done**
The arithmetic expression *expr1* is evaluated first (see the section 'Arithmetic Evaluation'). The arithmetic expression *expr2* is repeatedly evaluated until it evaluates to zero and when non−zero, *list* is executed and the arithmetic expression *expr3* evaluated. If any expression is omitted, then it behaves as if it evaluated to 1.

while *list* **do** *list* **done**
Execute the **do** *list* as long as the **while** *list* returns a zero exit status.

until *list* **do** *list* **done**
Execute the **do** *list* as long as **until** *list* returns a nonzero exit status.

repeat *word* **do** *list* **done**
word is expanded and treated as an arithmetic expression, which must evaluate to a number *n*. *list* is then executed *n* times.

The **repeat** syntax is disabled by default when the shell starts in a mode emulating another shell. It can be enabled with the command '**enable −r repeat**'

case *word* **in** [[(] *pattern* [**|** *pattern*] ... **)** *list* (**;;**|**;&**|**;|**)] ... **esac**
Execute the *list* associated with the first *pattern* that matches *word*, if any. The form of the patterns is the same as that used for filename generation. See the section 'Filename Generation'.

Note further that, unless the **SH_GLOB** option is set, the whole pattern with alternatives is treated by the shell as equivalent to a group of patterns within parentheses, although white space may appear about the parentheses and the vertical bar and will be stripped from the pattern at those points. White space may appear elsewhere in the pattern; this is not stripped. If the **SH_GLOB** option is set, so that an opening parenthesis can be unambiguously treated as part of the case syntax, the expression is parsed into separate words and these are treated as strict alternatives (as in other shells).

If the *list* that is executed is terminated with **;&** rather than **;;**, the following list is also executed. The rule for the terminator of the following list **;;**, **;&** or **;|** is applied unless the **esac** is reached.

If the *list* that is executed is terminated with **;|** the shell continues to scan the *pattern*s looking for the next match, executing the corresponding *list*, and applying the rule for the corresponding terminator **;;**, **;&** or **;|**. Note that *word* is not re−expanded; all applicable *pattern*s are tested with the same *word*.

select *name* [**in** *word ... term*] **do** *list* **done**
where *term* is one or more newline or **;** to terminate the *word*s. Print the set of *word*s, each preceded by a number. If the **in** *word* is omitted, use the positional parameters. The **PROMPT3** prompt is printed and a line is read from the line editor if the shell is interactive and that is active, or else standard input. If this line consists of the number of one of the listed *word*s, then the parameter *name* is set to the *word* corresponding to this number. If this line is empty, the selection list is printed again. Otherwise, the value of the parameter *name* is set to null. The contents of the line read from standard input is saved in the parameter **REPLY**. *list* is executed for each selection until a break or end−of−file is encountered.

(*list* **)** Execute *list* in a subshell. Traps set by the **trap** builtin are reset to their default values while executing *list*.

{ *list* **}** Execute *list*.

{ *try−list* **} always {** *always−list* **}**
First execute *try−list*. Regardless of errors, or **break** or **continue** commands encountered within *try−list*, execute *always−list*. Execution then continues from the result of the execution of *try−list*; in other words, any error, or **break** or **continue** command is treated in the normal way, as if *always−list* were not present. The two chunks of code are referred to as the 'try block' and the 'always block'.

Optional newlines or semicolons may appear after the **always**; note, however, that they may *not* appear between the preceding closing brace and the **always**.

An 'error' in this context is a condition such as a syntax error which causes the shell to abort execution of the current function, script, or list. Syntax errors encountered while the shell is parsing the code do not cause the *always−list* to be executed. For example, an erroneously constructed **if** block in *try−list* would cause the shell to abort during parsing, so that *always−list* would not be

executed, while an erroneous substitution such as **${*foo*}** would cause a run−time error, after which *always−list* would be executed.

An error condition can be tested and reset with the special integer variable **TRY_BLOCK_ERROR**. Outside an *always−list* the value is irrelevant, but it is initialised to **−1**. Inside *always−list*, the value is 1 if an error occurred in the *try−list*, else 0. If **TRY_BLOCK_ERROR** is set to 0 during the *always−list*, the error condition caused by the *try−list* is reset, and shell execution continues normally after the end of *always−list*. Altering the value during the *try−list* is not useful (unless this forms part of an enclosing **always** block).

Regardless of **TRY_BLOCK_ERROR**, after the end of *always−list* the normal shell status **$?** is the value returned from *try−list*. This will be non−zero if there was an error, even if **TRY_BLOCK_ERROR** was set to zero.

The following executes the given code, ignoring any errors it causes. This is an alternative to the usual convention of protecting code by executing it in a subshell.

```
{
  # code which may cause an error
 } always {
  # This code is executed regardless of the error.
  (( TRY_BLOCK_ERROR = 0 ))
}
# The error condition has been reset.
```

When a **try** block occurs outside of any function, a **return** or a **exit** encountered in *try−list* does *not* cause the execution of *always−list*. Instead, the shell exits immediately after any **EXIT** trap has been executed. Otherwise, a **return** command encountered in *try−list* will cause the execution of *always−list*, just like **break** and **continue**.

function *word ...* [**()**] [*term*] **{** *list* **}**
word ... **()** [*term*] **{** *list* **}**
word ... **()** [*term*] *command*

where *term* is one or more newline or **;**. Define a function which is referenced by any one of *word*. Normally, only one *word* is provided; multiple *word*s are usually only useful for setting traps. The body of the function is the *list* between the **{** and **}**. See the section 'Functions'.

If the option **SH_GLOB** is set for compatibility with other shells, then whitespace may appear between the left and right parentheses when there is a single *word*; otherwise, the parentheses will be treated as forming a globbing pattern in that case.

In any of the forms above, a redirection may appear outside the function body, for example

```
func() { ... } 2>&1
```

The redirection is stored with the function and applied whenever the function is executed. Any variables in the redirection are expanded at the point the function is executed, but outside the function scope.

time [*pipeline*]

The *pipeline* is executed, and timing statistics are reported on the standard error in the form specified by the **TIMEFMT** parameter. If *pipeline* is omitted, print statistics about the shell process and its children.

[[*exp* **]]**

Evaluates the conditional expression *exp* and return a zero exit status if it is true. See the section 'Conditional Expressions' for a description of *exp*.

ALTERNATE FORMS FOR COMPLEX COMMANDS

Many of zsh's complex commands have alternate forms. These are non−standard and are likely not to be obvious even to seasoned shell programmers; they should not be used anywhere that portability of shell code is a concern.

The short versions below only work if *sublist* is of the form '**{** *list* **}**' or if the **SHORT_LOOPS** option is set. For the **if**, **while** and **until** commands, in both these cases the test part of the loop must also be suitably delimited, such as by '**[[** ... **]]**' or '((...))', else the end of the test will not be recognized. For the **for**, **repeat**, **case** and **select** commands no such special form for the arguments is necessary, but the other condition (the special form of *sublist* or use of the **SHORT_LOOPS** option) still applies.

if *list* **{** *list* **}** [**elif** *list* **{** *list* **}**] ... [**else {** *list* **}**]
An alternate form of **if**. The rules mean that

```
if [[ -o ignorebraces ]] {
  print yes
}
```

works, but

```
if true {  # Does not work!
  print yes
}
```

does *not*, since the test is not suitably delimited.

if *list sublist*
A short form of the alternate **if**. The same limitations on the form of *list* apply as for the previous form.

for *name* ... **(** *word* ... **)** *sublist*
A short form of **for**.

for *name* ... [**in** *word* ...] *term sublist*
where *term* is at least one newline or **;**. Another short form of **for**.

for (([*expr1*] **;** [*expr2*] **;** [*expr3*] **))** *sublist*
A short form of the arithmetic **for** command.

foreach *name* ... **(** *word* ... **)** *list* **end**
Another form of **for**.

while *list* **{** *list* **}**
An alternative form of **while**. Note the limitations on the form of *list* mentioned above.

until *list* **{** *list* **}**
An alternative form of **until**. Note the limitations on the form of *list* mentioned above.

repeat *word sublist*
This is a short form of **repeat**.

case *word* **{** [[(] *pattern* [**|** *pattern*] ... **)** *list* (**;;**|**;&**|**;|**)] ... **}**
An alternative form of **case**.

select *name* [**in** *word ... term*] *sublist*
where *term* is at least one newline or **;**. A short form of **select**.

function *word* ... [**()**] [*term*] *sublist*
This is a short form of **function**.

RESERVED WORDS

The following words are recognized as reserved words when used as the first word of a command unless quoted or disabled using **disable -r**:

do done esac then elif else fi for case if while function repeat time until select coproc nocorrect foreach end ! [[{ } declare export float integer local readonly typeset

Additionally, '**}**' is recognized in any position if neither the **IGNORE_BRACES** option nor the **IGNORE_CLOSE_BRACES** option is set.

ERRORS

Certain errors are treated as fatal by the shell: in an interactive shell, they cause control to return to the command line, and in a non−interactive shell they cause the shell to be aborted. In older versions of zsh, a non−interactive shell running a script would not abort completely, but would resume execution at the next command to be read from the script, skipping the remainder of any functions or shell constructs such as loops or conditions; this somewhat illogical behaviour can be recovered by setting the option **CONTINUE_ON_ERROR**.

Fatal errors found in non−interactive shells include:

- Failure to parse shell options passed when invoking the shell
- Failure to change options with the **set** builtin
- Parse errors of all sorts, including failures to parse mathematical expressions
- Failures to set or modify variable behaviour with **typeset**, **local**, **declare**, **export**, **integer**, **float**
- Execution of incorrectly positioned loop control structures (**continue**, **break**)
- Attempts to use regular expression with no regular expression module available
- Disallowed operations when the **RESTRICTED** options is set
- Failure to create a pipe needed for a pipeline
- Failure to create a multio
- Failure to autoload a module needed for a declared shell feature
- Errors creating command or process substitutions
- Syntax errors in glob qualifiers
- File generation errors where not caught by the option **BAD_PATTERN**
- All bad patterns used for matching within case statements
- File generation failures where not caused by **NO_MATCH** or similar options
- All file generation errors where the pattern was used to create a multio
- Memory errors where detected by the shell
- Invalid subscripts to shell variables
- Attempts to assign read−only variables
- Logical errors with variables such as assignment to the wrong type
- Use of invalid variable names
- Errors in variable substitution syntax
- Failure to convert characters in **$'**...**'** expressions

If the **POSIX_BUILTINS** option is set, more errors associated with shell builtin commands are treated as fatal, as specified by the POSIX standard.

COMMENTS

In non−interactive shells, or in interactive shells with the **INTERACTIVE_COMMENTS** option set, a word beginning with the third character of the **histchars** parameter (‘**#**’ by default) causes that word and all the following characters up to a newline to be ignored.

ALIASING

Every eligible *word* in the shell input is checked to see if there is an alias defined for it. If so, it is replaced by the text of the alias if it is in command position (if it could be the first word of a simple command), or if the alias is global. If the replacement text ends with a space, the next word in the shell input is always eligible for purposes of alias expansion. An alias is defined using the **alias** builtin; global aliases may be defined using the **−g** option to that builtin.

A *word* is defined as:

- Any plain string or glob pattern
- Any quoted string, using any quoting method (note that the quotes must be part of the alias definition for this to be eligible)
- Any parameter reference or command substitution
- Any series of the foregoing, concatenated without whitespace or other tokens between them
- Any reserved word (**case**, **do**, **else**, etc.)
- With global aliasing, any command separator, any redirection operator, and '(' or ')' when not part of a glob pattern

Alias expansion is done on the shell input before any other expansion except history expansion. Therefore, if an alias is defined for the word **foo**, alias expansion may be avoided by quoting part of the word, e.g. **\foo**. Any form of quoting works, although there is nothing to prevent an alias being defined for the quoted form such as **\foo** as well.

When **POSIX_ALIASES** is set, only plain unquoted strings are eligible for aliasing. The **alias** builtin does not reject ineligible aliases, but they are not expanded.

For use with completion, which would remove an initial backslash followed by a character that isn't special, it may be more convenient to quote the word by starting with a single quote, i.e. **'foo**; completion will automatically add the trailing single quote.

Alias difficulties

Although aliases can be used in ways that bend normal shell syntax, not every string of non−white−space characters can be used as an alias.

Any set of characters not listed as a word above is not a word, hence no attempt is made to expand it as an alias, no matter how it is defined (i.e. via the builtin or the special parameter **aliases** described in the section THE ZSH/PARAMETER MODULE in *zshmodules*(1)). However, as noted in the case of **POSIX_ALIASES** above, the shell does not attempt to deduce whether the string corresponds to a word at the time the alias is created.

For example, an expression containing an **=** at the start of a command line is an assignment and cannot be expanded as an alias; a lone **=** is not an assignment but can only be set as an alias using the parameter, as otherwise the **=** is taken part of the syntax of the builtin command.

It is not presently possible to alias the '((' token that introduces arithmetic expressions, because until a full statement has been parsed, it cannot be distinguished from two consecutive '(' tokens introducing nested subshells. Also, if a separator such as **&&** is aliased, **\&&** turns into the two tokens **\&** and **&**, each of which may have been aliased separately. Similarly for **\<<**, **\>|**, etc.

There is a commonly encountered problem with aliases illustrated by the following code:

```
alias echobar='echo bar'; echobar
```

This prints a message that the command **echobar** could not be found. This happens because aliases are expanded when the code is read in; the entire line is read in one go, so that when **echobar** is executed it is too late to expand the newly defined alias. This is often a problem in shell scripts, functions, and code executed with '**source**' or '**.**'. Consequently, use of functions rather than aliases is recommended in non−interactive code.

Note also the unhelpful interaction of aliases and function definitions:

```
alias func='noglob func'
func() {
  echo Do something with $*
}
```

Because aliases are expanded in function definitions, this causes the following command to be executed:

```
noglob func() {
  echo Do something with $*
}
```

which defines **noglob** as well as **func** as functions with the body given. To avoid this, either quote the name **func** or use the alternative function definition form '**function func**'. Ensuring the alias is defined after the function works but is problematic if the code fragment might be re−executed.

QUOTING

A character may be *quoted* (that is, made to stand for itself) by preceding it with a '\'. '\' followed by a newline is ignored.

A string enclosed between '**$'** and '**'** is processed the same way as the string arguments of the **print** builtin, and the resulting string is considered to be entirely quoted. A literal '**'** character can be included in the string by using the '**\'** escape.

All characters enclosed between a pair of single quotes (**'**') that is not preceded by a '**$**' are quoted. A single quote cannot appear within single quotes unless the option **RC_QUOTES** is set, in which case a pair of single quotes are turned into a single quote. For example,

```
print ''''
```

outputs nothing apart from a newline if **RC_QUOTES** is not set, but one single quote if it is set.

Inside double quotes ('**"**'), parameter and command substitution occur, and '\' quotes the characters '\', '**`**', '**"**', '**$**', and the first character of **$histchars** (default '**!**').

REDIRECTION

If a command is followed by **&** and job control is not active, then the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

The following may appear anywhere in a simple command or may precede or follow a complex command. Expansion occurs before *word* or *digit* is used except as noted below. If the result of substitution on *word* produces more than one filename, redirection occurs for each separate filename in turn.

< *word* Open file *word* for reading as standard input. It is an error to open a file in this fashion if it does not exist.

<> *word*
Open file *word* for reading and writing as standard input. If the file does not exist then it is created.

> *word* Open file *word* for writing as standard output. If the file does not exist then it is created. If the file exists, and the **CLOBBER** option is unset, this causes an error; otherwise, it is truncated to zero length.

>| *word*
>! *word*
Same as **>**, except that the file is truncated to zero length if it exists, regardless of **CLOBBER**.

>> *word*
Open file *word* for writing in append mode as standard output. If the file does not exist, and the **CLOBBER** and **APPEND_CREATE** options are both unset, this causes an error; otherwise, the file is created.

>>| *word*
>>! *word*
Same as **>>**, except that the file is created if it does not exist, regardless of **CLOBBER** and **APPEND_CREATE**.

<<[**−**] *word*
The shell input is read up to a line that is the same as *word*, or to an end−of−file. No parameter expansion, command substitution or filename generation is performed on *word*. The resulting

document, called a *here−document*, becomes the standard input.

If any character of *word* is quoted with single or double quotes or a '\', no interpretation is placed upon the characters of the document. Otherwise, parameter and command substitution occurs, '\' followed by a newline is removed, and '\' must be used to quote the characters '\', '**$**', '**`**' and the first character of *word*.

Note that *word* itself does not undergo shell expansion. Backquotes in *word* do not have their usual effect; instead they behave similarly to double quotes, except that the backquotes themselves are passed through unchanged. (This information is given for completeness and it is not recommended that backquotes be used.) Quotes in the form **$'**...**'** have their standard effect of expanding backslashed references to special characters.

If **<<−** is used, then all leading tabs are stripped from *word* and from the document.

<<< *word*
: Perform shell expansion on *word* and pass the result to standard input. This is known as a *here−string*. Compare the use of *word* in here−documents above, where *word* does not undergo shell expansion.

<& *number*
>& *number*
: The standard input/output is duplicated from file descriptor *number* (see *dup2*(2)).

<& −
>& − Close the standard input/output.

<& p
>& p The input/output from/to the coprocess is moved to the standard input/output.

>& *word*
&> *word*
: (Except where '**>&** *word*' matches one of the above syntaxes; '**&>**' can always be used to avoid this ambiguity.) Redirects both standard output and standard error (file descriptor 2) in the manner of '**>** *word*'. Note that this does *not* have the same effect as '**>** *word* **2>&1**' in the presence of multios (see the section below).

>&| *word*
>&! *word*
&>| *word*
&>! *word*
: Redirects both standard output and standard error (file descriptor 2) in the manner of '**>|** *word*'.

>>& *word*
&>> *word*
: Redirects both standard output and standard error (file descriptor 2) in the manner of '**>>** *word*'.

>>&| *word*
>>&! *word*
&>>| *word*
&>>! *word*
: Redirects both standard output and standard error (file descriptor 2) in the manner of '**>>|** *word*'.

If one of the above is preceded by a digit, then the file descriptor referred to is that specified by the digit instead of the default 0 or 1. The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example:

... **1>***fname* **2>&1**

first associates file descriptor 1 with file *fname*. It then associates file descriptor 2 with the file associated with file descriptor 1 (that is, *fname*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and then file descriptor 1 would be associated with file *fname*.

The '**|&**' command separator described in *Simple Commands & Pipelines* in *zshmisc*(1) is a shorthand for '**2>&1 |**'.

The various forms of process substitution, '**<(***list***)**', and '**=(***list***)**' for input and '**>(***list***)**' for output, are often used together with redirection. For example, if *word* in an output redirection is of the form '**>(***list***)**' then the output is piped to the command represented by *list*. See *Process Substitution* in *zshexpn*(1).

OPENING FILE DESCRIPTORS USING PARAMETERS

When the shell is parsing arguments to a command, and the shell option **IGNORE_BRACES** is not set, a different form of redirection is allowed: instead of a digit before the operator there is a valid shell identifier enclosed in braces. The shell will open a new file descriptor that is guaranteed to be at least 10 and set the parameter named by the identifier to the file descriptor opened. No whitespace is allowed between the closing brace and the redirection character. For example:

... **{myfd}>&1**

This opens a new file descriptor that is a duplicate of file descriptor 1 and sets the parameter **myfd** to the number of the file descriptor, which will be at least 10. The new file descriptor can be written to using the syntax **>&$myfd**. The file descriptor remains open in subshells and forked external executables.

The syntax **{***varid***}>&−**, for example **{myfd}>&−**, may be used to close a file descriptor opened in this fashion. Note that the parameter given by *varid* must previously be set to a file descriptor in this case.

It is an error to open or close a file descriptor in this fashion when the parameter is readonly. However, it is not an error to read or write a file descriptor using **<&$***param* or **>&$***param* if *param* is readonly.

If the option **CLOBBER** is unset, it is an error to open a file descriptor using a parameter that is already set to an open file descriptor previously allocated by this mechanism. Unsetting the parameter before using it for allocating a file descriptor avoids the error.

Note that this mechanism merely allocates or closes a file descriptor; it does not perform any redirections from or to it. It is usually convenient to allocate a file descriptor prior to use as an argument to **exec**. The syntax does not in any case work when used around complex commands such as parenthesised subshells or loops, where the opening brace is interpreted as part of a command list to be executed in the current shell.

The following shows a typical sequence of allocation, use, and closing of a file descriptor:

```
integer myfd
exec {myfd}>~/logs/mylogfile.txt
print This is a log message. >&$myfd
exec {myfd}>&−
```

Note that the expansion of the variable in the expression **>&$myfd** occurs at the point the redirection is opened. This is after the expansion of command arguments and after any redirections to the left on the command line have been processed.

MULTIOS

If the user tries to open a file descriptor for writing more than once, the shell opens the file descriptor as a pipe to a process that copies its input to all the specified outputs, similar to **tee**, provided the **MULTIOS** option is set, as it is by default. Thus:

date >foo >bar

writes the date to two files, named '**foo**' and '**bar**'. Note that a pipe is an implicit redirection; thus

date >foo | cat

writes the date to the file '**foo**', and also pipes it to cat.

Note that the shell opens all the files to be used in the multio process immediately, not at the point they are about to be written.

Note also that redirections are always expanded in order. This happens regardless of the setting of the **MULTIOS** option, but with the option in effect there are additional consequences. For example, the meaning of the expression **>&1** will change after a previous redirection:

date >&1 >output

In the case above, the **>&1** refers to the standard output at the start of the line; the result is similar to the **tee** command. However, consider:

date >output >&1

As redirections are evaluated in order, when the **>&1** is encountered the standard output is set to the file **output** and another copy of the output is therefore sent to that file. This is unlikely to be what is intended.

If the **MULTIOS** option is set, the word after a redirection operator is also subjected to filename generation (globbing). Thus

: > *

will truncate all files in the current directory, assuming there's at least one. (Without the **MULTIOS** option, it would create an empty file called '*'.) Similarly, you can do

echo exit 0 >> *.sh

If the user tries to open a file descriptor for reading more than once, the shell opens the file descriptor as a pipe to a process that copies all the specified inputs to its output in the order specified, provided the **MULTIOS** option is set. It should be noted that each file is opened immediately, not at the point where it is about to be read: this behaviour differs from **cat**, so if strictly standard behaviour is needed, **cat** should be used instead.

Thus

sort <foo <fubar

or even

sort <f{oo,ubar}

is equivalent to '**cat foo fubar | sort**'.

Expansion of the redirection argument occurs at the point the redirection is opened, at the point described above for the expansion of the variable in **>&$myfd**.

Note that a pipe is an implicit redirection; thus

cat bar | sort <foo

is equivalent to '**cat bar foo | sort**' (note the order of the inputs).

If the **MULTIOS** option is *un*set, each redirection replaces the previous redirection for that file descriptor. However, all files redirected to are actually opened, so

echo Hello > bar > baz

when **MULTIOS** is unset will truncate '**bar**', and write '**Hello**' into '**baz**'.

There is a problem when an output multio is attached to an external program. A simple example shows this:

cat file >file1 >file2
cat file1 file2

Here, it is possible that the second '**cat**' will not display the full contents of **file1** and **file2** (i.e. the original contents of **file** repeated twice).

The reason for this is that the multios are spawned after the **cat** process is forked from the parent shell, so the parent shell does not wait for the multios to finish writing data. This means the command as shown can exit before **file1** and **file2** are completely written. As a workaround, it is possible to run the **cat** process as part of a job in the current shell:

{ cat file } >file >file2

Here, the **{...}** job will pause to wait for both files to be written.

REDIRECTIONS WITH NO COMMAND

When a simple command consists of one or more redirection operators and zero or more parameter assignments, but no command name, zsh can behave in several ways.

If the parameter **NULLCMD** is not set or the option **CSH_NULLCMD** is set, an error is caused. This is the **csh** behavior and **CSH_NULLCMD** is set by default when emulating **csh**.

If the option **SH_NULLCMD** is set, the builtin '**:**' is inserted as a command with the given redirections. This is the default when emulating **sh** or **ksh**.

Otherwise, if the parameter **NULLCMD** is set, its value will be used as a command with the given redirections. If both **NULLCMD** and **READNULLCMD** are set, then the value of the latter will be used instead of that of the former when the redirection is an input. The default for **NULLCMD** is '**cat**' and for **READNULLCMD** is '**more**'. Thus

< file

shows the contents of **file** on standard output, with paging if that is a terminal. **NULLCMD** and **READNULLCMD** may refer to shell functions.

COMMAND EXECUTION

If a command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, the function is invoked as described in the section 'Functions'. If there exists a shell builtin by that name, the builtin is invoked.

Otherwise, the shell searches each element of **$path** for a directory containing an executable file by that name. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.

If execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a shell script. **/bin/sh** is spawned to execute it. If the program is a file beginning with '**#!**', the remainder of the first line specifies an interpreter for the program. The shell will execute the specified interpreter on operating systems that do not handle this executable format in the kernel.

If no external command is found but a function **command_not_found_handler** exists the shell executes this function with all command line arguments. The return status of the function becomes the status of the command. If the function wishes to mimic the behaviour of the shell when the command is not found, it should print the message '**command not found:** *cmd*' to standard error and return status 127. Note that the handler is executed in a subshell forked to execute an external command, hence changes to directories, shell parameters, etc. have no effect on the main shell.

FUNCTIONS

Shell functions are defined with the **function** reserved word or the special syntax '*funcname* ()'. Shell functions are read in and stored internally. Alias names are resolved when the function is read. Functions are executed like commands with the arguments passed as positional parameters. (See the section 'Command Execution'.)

Functions execute in the same process as the caller and share all files and present working directory with the caller. A trap on **EXIT** set inside a function is executed after the function completes in the environment of the caller.

The **return** builtin is used to return from function calls.

Function identifiers can be listed with the **functions** builtin. Functions can be undefined with the **unfunction** builtin.

AUTOLOADING FUNCTIONS

A function can be marked as *undefined* using the **autoload** builtin (or '**functions –u**' or '**typeset –fu**'). Such a function has no body. When the function is first executed, the shell searches for its definition using the elements of the **fpath** variable. Thus to define functions for autoloading, a typical sequence is:

```
fpath=(~/myfuncs $fpath)
autoload myfunc1 myfunc2 ...
```

The usual alias expansion during reading will be suppressed if the **autoload** builtin or its equivalent is given

the option **−U**. This is recommended for the use of functions supplied with the zsh distribution. Note that for functions precompiled with the **zcompile** builtin command the flag **−U** must be provided when the **.zwc** file is created, as the corresponding information is compiled into the latter.

For each *element* in **fpath**, the shell looks for three possible files, the newest of which is used to load the definition for the function:

*element***.zwc**
: A file created with the **zcompile** builtin command, which is expected to contain the definitions for all functions in the directory named *element*. The file is treated in the same manner as a directory containing files for functions and is searched for the definition of the function. If the definition is not found, the search for a definition proceeds with the other two possibilities described below.

  If *element* already includes a **.zwc** extension (i.e. the extension was explicitly given by the user), *element* is searched for the definition of the function without comparing its age to that of other files; in fact, there does not need to be any directory named *element* without the suffix. Thus including an element such as '**/usr/local/funcs.zwc**' in **fpath** will speed up the search for functions, with the disadvantage that functions included must be explicitly recompiled by hand before the shell notices any changes.

element/*function***.zwc**
: A file created with **zcompile**, which is expected to contain the definition for *function*. It may include other function definitions as well, but those are neither loaded nor executed; a file found in this way is searched *only* for the definition of *function*.

element/*function*
: A file of zsh command text, taken to be the definition for *function*.

In summary, the order of searching is, first, in the *parents of* directories in **fpath** for the newer of either a compiled directory or a directory in **fpath**; second, if more than one of these contains a definition for the function that is sought, the leftmost in the **fpath** is chosen; and third, within a directory, the newer of either a compiled function or an ordinary function definition is used.

If the **KSH_AUTOLOAD** option is set, or the file contains only a simple definition of the function, the file's contents will be executed. This will normally define the function in question, but may also perform initialization, which is executed in the context of the function execution, and may therefore define local parameters. It is an error if the function is not defined by loading the file.

Otherwise, the function body (with no surrounding '*funcname***() {**...**}**') is taken to be the complete contents of the file. This form allows the file to be used directly as an executable shell script. If processing of the file results in the function being re−defined, the function itself is not re−executed. To force the shell to perform initialization and then call the function defined, the file should contain initialization code (which will be executed then discarded) in addition to a complete function definition (which will be retained for subsequent calls to the function), and a call to the shell function, including any arguments, at the end.

For example, suppose the autoload file **func** contains

```
func() { print This is func; }
print func is initialized
```

then '**func; func**' with **KSH_AUTOLOAD** set will produce both messages on the first call, but only the message '**This is func**' on the second and subsequent calls. Without **KSH_AUTOLOAD** set, it will produce the initialization message on the first call, and the other message on the second and subsequent calls.

It is also possible to create a function that is not marked as autoloaded, but which loads its own definition by searching **fpath**, by using '**autoload −X**' within a shell function. For example, the following are equivalent:

```
myfunc() {
 autoload −X
}
myfunc args...
```

and

```
unfunction myfunc   # if myfunc was defined
autoload myfunc
myfunc args...
```

In fact, the **functions** command outputs '**builtin autoload −X**' as the body of an autoloaded function. This is done so that

```
eval "$(functions)"
```

produces a reasonable result. A true autoloaded function can be identified by the presence of the comment '**# undefined**' in the body, because all comments are discarded from defined functions.

To load the definition of an autoloaded function **myfunc** without executing **myfunc**, use:

```
autoload +X myfunc
```

ANONYMOUS FUNCTIONS

If no name is given for a function, it is 'anonymous' and is handled specially. Either form of function definition may be used: a '()' with no preceding name, or a '**function**' with an immediately following open brace. The function is executed immediately at the point of definition and is not stored for future use. The function name is set to '**(anon)**'.

Arguments to the function may be specified as words following the closing brace defining the function, hence if there are none no arguments (other than **$0**) are set. This is a difference from the way other functions are parsed: normal function definitions may be followed by certain keywords such as '**else**' or '**fi**', which will be treated as arguments to anonymous functions, so that a newline or semicolon is needed to force keyword interpretation.

Note also that the argument list of any enclosing script or function is hidden (as would be the case for any other function called at this point).

Redirections may be applied to the anonymous function in the same manner as to a current−shell structure enclosed in braces. The main use of anonymous functions is to provide a scope for local variables. This is particularly convenient in start−up files as these do not provide their own local variable scope.

For example,

```
variable=outside
function {
 local variable=inside
 print "I am $variable with arguments $*"
} this and that
print "I am $variable"
```

outputs the following:

```
I am inside with arguments this and that
I am outside
```

Note that function definitions with arguments that expand to nothing, for example '**name=; function $name { ... }**', are not treated as anonymous functions. Instead, they are treated as normal function definitions where the definition is silently discarded.

SPECIAL FUNCTIONS

Certain functions, if defined, have special meaning to the shell.

Hook Functions

For the functions below, it is possible to define an array that has the same name as the function with '**_functions**' appended. Any element in such an array is taken as the name of a function to execute; it is executed in the same context and with the same arguments as the basic function. For example, if **$chpwd_functions** is an array containing the values '**mychpwd**', '**chpwd_save_dirstack**', then the shell attempts to execute the functions '**chpwd**', '**mychpwd**' and '**chpwd_save_dirstack**', in that order. Any function that does not exist is silently ignored. A function found by this mechanism is referred to elsewhere as a 'hook function'.

An error in any function causes subsequent functions not to be run. Note further that an error in a **precmd** hook causes an immediately following **periodic** function not to run (though it may run at the next opportunity).

chpwd Executed whenever the current working directory is changed.

periodic

If the parameter **PERIOD** is set, this function is executed every **$PERIOD** seconds, just before a prompt. Note that if multiple functions are defined using the array **periodic_functions** only one period is applied to the complete set of functions, and the scheduled time is not reset if the list of functions is altered. Hence the set of functions is always called together.

precmd

Executed before each prompt. Note that precommand functions are not re−executed simply because the command line is redrawn, as happens, for example, when a notification about an exiting job is displayed.

preexec

Executed just after a command has been read and is about to be executed. If the history mechanism is active (regardless of whether the line was discarded from the history buffer), the string that the user typed is passed as the first argument, otherwise it is an empty string. The actual command that will be executed (including expanded aliases) is passed in two different forms: the second argument is a single−line, size−limited version of the command (with things like function bodies elided); the third argument contains the full text that is being executed.

zshaddhistory

Executed when a history line has been read interactively, but before it is executed. The sole argument is the complete history line (so that any terminating newline will still be present).

If any of the hook functions returns status 1 (or any non−zero value other than 2, though this is not guaranteed for future versions of the shell) the history line will not be saved, although it lingers in the history until the next line is executed, allowing you to reuse or edit it immediately.

If any of the hook functions returns status 2 the history line will be saved on the internal history list, but not written to the history file. In case of a conflict, the first non−zero status value is taken.

A hook function may call '**fc −p** ...' to switch the history context so that the history is saved in a different file from the that in the global **HISTFILE** parameter. This is handled specially: the history context is automatically restored after the processing of the history line is finished.

The following example function works with one of the options **INC_APPEND_HISTORY** or **SHARE_HISTORY** set, in order that the line is written out immediately after the history entry is added. It first adds the history line to the normal history with the newline stripped, which is usually the correct behaviour. Then it switches the history context so that the line will be written to a history file in the current directory.

```
zshaddhistory() {
  print −sr −− ${1%%$'\n'}
  fc −p .zsh_local_history
}
```

zshexit Executed at the point where the main shell is about to exit normally. This is not called by exiting subshells, nor when the **exec** precommand modifier is used before an external command. Also, unlike **TRAPEXIT**, it is not called when functions exit.

Trap Functions

The functions below are treated specially but do not have corresponding hook arrays.

TRAP*NAL*

If defined and non−null, this function will be executed whenever the shell catches a signal **SIG***NAL*, where *NAL* is a signal name as specified for the **kill** builtin. The signal number will be passed as the first parameter to the function.

If a function of this form is defined and null, the shell and processes spawned by it will ignore **SIG***NAL*.

The return status from the function is handled specially. If it is zero, the signal is assumed to have been handled, and execution continues normally. Otherwise, the shell will behave as interrupted except that the return status of the trap is retained.

Programs terminated by uncaught signals typically return the status 128 plus the signal number. Hence the following causes the handler for **SIGINT** to print a message, then mimic the usual effect of the signal.

```
TRAPINT() {
  print "Caught SIGINT, aborting."
  return $(( 128 + $1 ))
}
```

The functions **TRAPZERR**, **TRAPDEBUG** and **TRAPEXIT** are never executed inside other traps.

TRAPDEBUG
: If the option **DEBUG_BEFORE_CMD** is set (as it is by default), executed before each command; otherwise executed after each command. See the description of the **trap** builtin in *zshbuiltins*(1) for details of additional features provided in debug traps.

TRAPEXIT
: Executed when the shell exits, or when the current function exits if defined inside a function. The value of **$?** at the start of execution is the exit status of the shell or the return status of the function exiting.

TRAPZERR
: Executed whenever a command has a non−zero exit status. However, the function is not executed if the command occurred in a sublist followed by '**&&**' or '**||**'; only the final command in a sublist of this type causes the trap to be executed. The function **TRAPERR** acts the same as **TRAPZERR** on systems where there is no **SIGERR** (this is the usual case).

The functions beginning '**TRAP**' may alternatively be defined with the **trap** builtin: this may be preferable for some uses. Setting a trap with one form removes any trap of the other form for the same signal; removing a trap in either form removes all traps for the same signal. The forms

```
TRAPNAL() {
 # code
}
```

('function traps') and

```
trap '
 # code
' NAL
```

('list traps') are equivalent in most ways, the exceptions being the following:

- Function traps have all the properties of normal functions, appearing in the list of functions and being called with their own function context rather than the context where the trap was triggered.
- The return status from function traps is special, whereas a return from a list trap causes the surrounding context to return with the given status.
- Function traps are not reset within subshells, in accordance with zsh behaviour; list traps are reset, in accordance with POSIX behaviour.

JOBS

If the **MONITOR** option is set, an interactive shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the **jobs** command, and assigns them small integer numbers. When a job is started asynchronously with '**&**', the shell prints a line to standard error which looks like:

[1] 1234

indicating that the job which was started asynchronously was job number 1 and had one (top−level) process, whose process ID was 1234.

If a job is started with '**&|**' or '**&!**', then that job is immediately disowned. After startup, it does not have a place in the job table, and is not subject to the job control features described here.

If you are running a job and wish to do something else you may hit the key ^Z (control−Z) which sends a **TSTP** signal to the current job: this key may be redefined by the **susp** option of the external **stty** command. The shell will then normally indicate that the job has been 'suspended', and print another prompt. You can then manipulate the state of this job, putting it in the background with the **bg** command, or run some other commands and then eventually bring the job back into the foreground with the foreground command **fg**. A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will suspend if it tries to read from the terminal.

Note that if the job running in the foreground is a shell function, then suspending it will have the effect of causing the shell to fork. This is necessary to separate the function's state from that of the parent shell performing the job control, so that the latter can return to the command line prompt. As a result, even if **fg** is used to continue the job the function will no longer be part of the parent shell, and any variables set by the function will not be visible in the parent shell. Thus the behaviour is different from the case where the function was never suspended. Zsh is different from many other shells in this regard.

One additional side effect is that use of **disown** with a job created by suspending shell code in this fashion is delayed: the job can only be disowned once any process started from the parent shell has terminated. At that point, the disowned job disappears silently from the job list.

The same behaviour is found when the shell is executing code as the right hand side of a pipeline or any complex shell construct such as **if**, **for**, etc., in order that the entire block of code can be managed as a single job. Background jobs are normally allowed to produce output, but this can be disabled by giving the command '**stty tostop**'. If you set this tty option, then background jobs will suspend when they try to produce output like they do when they try to read input.

When a command is suspended and continued later with the **fg** or **wait** builtins, zsh restores tty modes that were in effect when it was suspended. This (intentionally) does not apply if the command is continued via '**kill −CONT**', nor when it is continued with **bg**.

There are several ways to refer to jobs in the shell. A job can be referred to by the process ID of any process of the job or by one of the following:

%*number*
: The job with the given number.

%*string*
: The last job whose command line begins with *string*.

%?*string*
: The last job whose command line contains *string*.

%% Current job.

%+ Equivalent to '**%%**'.

%− Previous job.

The shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible. If the **NOTIFY** option is not set, it waits until just before it prints a prompt before it informs you. All such notifications are sent directly to the terminal, not to the standard output or standard error.

When the monitor mode is on, each background job that completes triggers any trap set for **CHLD**.

When you try to leave the shell while jobs are running or suspended, you will be warned that 'You have suspended (running) jobs'. You may use the **jobs** command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time; the suspended jobs will be terminated, and

the running jobs will be sent a **SIGHUP** signal, if the **HUP** option is set.

To avoid having the shell terminate the running jobs, either use the **nohup** command (see *nohup*(1)) or the **disown** builtin.

SIGNALS

The **INT** and **QUIT** signals for an invoked command are ignored if the command is followed by '**&**' and the **MONITOR** option is not active. The shell itself always ignores the **QUIT** signal. Otherwise, signals have the values inherited by the shell from its parent (but see the **TRAP***NAL* special functions in the section 'Functions').

Certain jobs are run asynchronously by the shell other than those explicitly put into the background; even in cases where the shell would usually wait for such jobs, an explicit **exit** command or exit due to the option **ERR_EXIT** will cause the shell to exit without waiting. Examples of such asynchronous jobs are process substitution, see the section PROCESS SUBSTITUTION in the *zshexpn*(1) manual page, and the handler processes for multios, see the section MULTIOS in the *zshmisc*(1) manual page.

ARITHMETIC EVALUATION

The shell can perform integer and floating point arithmetic, either using the builtin **let**, or via a substitution of the form **$((...))**. For integers, the shell is usually compiled to use 8−byte precision where this is available, otherwise precision is 4 bytes. This can be tested, for example, by giving the command '**print − $((12345678901))**'; if the number appears unchanged, the precision is at least 8 bytes. Floating point arithmetic always uses the 'double' type with whatever corresponding precision is provided by the compiler and the library.

The **let** builtin command takes arithmetic expressions as arguments; each is evaluated separately. Since many of the arithmetic operators, as well as spaces, require quoting, an alternative form is provided: for any command which begins with a '**((**', all the characters until a matching '**))**' are treated as a quoted expression and arithmetic expansion performed as for an argument of **let**. More precisely, '**((...))**' is equivalent to '**let "..."**'. The return status is 0 if the arithmetic value of the expression is non−zero, 1 if it is zero, and 2 if an error occurred.

For example, the following statement

((val = 2 + 1))

is equivalent to

let "val = 2 + 1"

both assigning the value 3 to the shell variable **val** and returning a zero status.

Integers can be in bases other than 10. A leading '**0x**' or '**0X**' denotes hexadecimal and a leading '**0b**' or '**0B**' binary. Integers may also be of the form '*base***#***n*', where *base* is a decimal number between two and thirty−six representing the arithmetic base and *n* is a number in that base (for example, '**16#ff**' is 255 in hexadecimal). The *base***#** may also be omitted, in which case base 10 is used. For backwards compatibility the form '**[***base***]***n*' is also accepted.

An integer expression or a base given in the form '*base***#***n*' may contain underscores ('**_**') after the leading digit for visual guidance; these are ignored in computation. Examples are **1_000_000** or **0xffff_ffff** which are equivalent to **1000000** and **0xffffffff** respectively.

It is also possible to specify a base to be used for output in the form '**[#***base***]**', for example '**[#16]**'. This is used when outputting arithmetical substitutions or when assigning to scalar parameters, but an explicitly defined integer or floating point parameter will not be affected. If an integer variable is implicitly defined by an arithmetic expression, any base specified in this way will be set as the variable's output arithmetic base as if the option '**−i** *base*' to the **typeset** builtin had been used. The expression has no precedence and if it occurs more than once in a mathematical expression, the last encountered is used. For clarity it is recommended that it appear at the beginning of an expression. As an example:

```
typeset −i 16 y
print $(( [#8] x = 32, y = 32 ))
print $x $y
```

outputs first '**8#40**', the rightmost value in the given output base, and then '**8#40 16#20**', because **y** has been explicitly declared to have output base 16, while **x** (assuming it does not already exist) is implicitly typed by the arithmetic evaluation, where it acquires the output base 8.

The *base* may be replaced or followed by an underscore, which may itself be followed by a positive integer (if it is missing the value 3 is used). This indicates that underscores should be inserted into the output string, grouping the number for visual clarity. The following integer specifies the number of digits to group together. For example:

setopt cbases
print $(([#16_4] 65536 ** 2))

outputs '**0x1_0000_0000**'.

The feature can be used with floating point numbers, in which case the base must be omitted; grouping is away from the decimal point. For example,

zmodload zsh/mathfunc
print $(([#_] sqrt(1e7)))

outputs '**3_162.277_660_168_379_5**' (the number of decimal places shown may vary).

If the **C_BASES** option is set, hexadecimal numbers are output in the standard C format, for example '**0xFF**' instead of the usual '**16#FF**'. If the option **OCTAL_ZEROES** is also set (it is not by default), octal numbers will be treated similarly and hence appear as '**077**' instead of '**8#77**'. This option has no effect on the output of bases other than hexadecimal and octal, and these formats are always understood on input.

When an output base is specified using the '**[#***base***]**' syntax, an appropriate base prefix will be output if necessary, so that the value output is valid syntax for input. If the **#** is doubled, for example '**[##16]**', then no base prefix is output.

Floating point constants are recognized by the presence of a decimal point or an exponent. The decimal point may be the first character of the constant, but the exponent character **e** or **E** may not, as it will be taken for a parameter name. All numeric parts (before and after the decimal point and in the exponent) may contain underscores after the leading digit for visual guidance; these are ignored in computation.

An arithmetic expression uses nearly the same syntax and associativity of expressions as in C.

In the native mode of operation, the following operators are supported (listed in decreasing order of precedence):

+ − ! ~ ++ −−
: unary plus/minus, logical NOT, complement, {pre,post}{in,de}crement

<< >> : bitwise shift left, right

& : bitwise AND

^ : bitwise XOR

| : bitwise OR

****** : exponentiation

*** / %** : multiplication, division, modulus (remainder)

+ − : addition, subtraction

< > <= >=
: comparison

== != : equality and inequality

&& : logical AND

|| ^^ : logical OR, XOR

? : : ternary operator

= += −= *= /= %= &= ^= |= <<= >>= &&= ||= ^^= **=
: assignment

, : comma operator

The operators '**&&**', '**||**', '**&&=**', and '**||=**' are short−circuiting, and only one of the latter two expressions in a ternary operator is evaluated. Note the precedence of the bitwise AND, OR, and XOR operators.

With the option **C_PRECEDENCES** the precedences (but no other properties) of the operators are altered to be the same as those in most other languages that support the relevant operators:

+ − ! ~ ++ −−
: unary plus/minus, logical NOT, complement, {pre,post}{in,de}crement

****** : exponentiation

*** / %** : multiplication, division, modulus (remainder)

+ − : addition, subtraction

<< >> : bitwise shift left, right

< > <= >=
: comparison

== != : equality and inequality

& : bitwise AND

^ : bitwise XOR

| : bitwise OR

&& : logical AND

^^ : logical XOR

|| : logical OR

? : : ternary operator

= += −= *= /= %= &= ^= |= <<= >>= &&= ||= ^^= **=
: assignment

, : comma operator

Note the precedence of exponentiation in both cases is below that of unary operators, hence '**−3**2**' evaluates as '**9**', not '**−9**'. Use parentheses where necessary: '**−(3**2)**'. This is for compatibility with other shells.

Mathematical functions can be called with the syntax '*func*(*args*)', where the function decides if the *args* is used as a string or a comma−separated list of arithmetic expressions. The shell currently defines no mathematical functions by default, but the module **zsh/mathfunc** may be loaded with the **zmodload** builtin to provide standard floating point mathematical functions.

An expression of the form '**##***x*' where *x* is any character sequence such as '**a**', '**^A**', or '**\M−\C−x**' gives the value of this character and an expression of the form '**#***name*' gives the value of the first character of the contents of the parameter *name*. Character values are according to the character set used in the current locale; for multibyte character handling the option **MULTIBYTE** must be set. Note that this form is different from '**$#***name*', a standard parameter substitution which gives the length of the parameter *name*. '**#**' is accepted instead of '**##**', but its use is deprecated.

Named parameters and subscripted arrays can be referenced by name within an arithmetic expression without using the parameter expansion syntax. For example,

((val2 = val1 * 2))

assigns twice the value of **$val1** to the parameter named **val2**.

An internal integer representation of a named parameter can be specified with the **integer** builtin. Arithmetic evaluation is performed on the value of each assignment to a named parameter declared integer in this manner. Assigning a floating point number to an integer results in rounding towards zero.

Likewise, floating point numbers can be declared with the **float** builtin; there are two types, differing only in their output format, as described for the **typeset** builtin. The output format can be bypassed by using arithmetic substitution instead of the parameter substitution, i.e. '**${***float***}**' uses the defined format, but '**$((***float***))**' uses a generic floating point format.

Promotion of integer to floating point values is performed where necessary. In addition, if any operator which requires an integer ('**&**', '**|**', '**^**', '**<<**', '**>>**' and their equivalents with assignment) is given a floating point argument, it will be silently rounded towards zero except for '**~**' which rounds down.

Users should beware that, in common with many other programming languages but not software designed for calculation, the evaluation of an expression in zsh is taken a term at a time and promotion of integers to

floating point does not occur in terms only containing integers. A typical result of this is that a division such as **6/8** is truncated, in this being rounded towards 0. The **FORCE_FLOAT** shell option can be used in scripts or functions where floating point evaluation is required throughout.

Scalar variables can hold integer or floating point values at different times; there is no memory of the numeric type in this case.

If a variable is first assigned in a numeric context without previously being declared, it will be implicitly typed as **integer** or **float** and retain that type either until the type is explicitly changed or until the end of the scope. This can have unforeseen consequences. For example, in the loop

```
for (( f = 0; f < 1; f += 0.1 )); do
# use $f
done
```

if **f** has not already been declared, the first assignment will cause it to be created as an integer, and consequently the operation '**f += 0.1**' will always cause the result to be truncated to zero, so that the loop will fail. A simple fix would be to turn the initialization into '**f = 0.0**'. It is therefore best to declare numeric variables with explicit types.

CONDITIONAL EXPRESSIONS

A *conditional expression* is used with the **[[** compound command to test attributes of files and to compare strings. Each expression can be constructed from one or more of the following unary or binary expressions:

−a *file* true if *file* exists.

−b *file* true if *file* exists and is a block special file.

−c *file* true if *file* exists and is a character special file.

−d *file* true if *file* exists and is a directory.

−e *file* true if *file* exists.

−f *file* true if *file* exists and is a regular file.

−g *file* true if *file* exists and has its setgid bit set.

−h *file* true if *file* exists and is a symbolic link.

−k *file* true if *file* exists and has its sticky bit set.

−n *string*
true if length of *string* is non−zero.

−o *option*
true if option named *option* is on. *option* may be a single character, in which case it is a single letter option name. (See the section 'Specifying Options'.)

When no option named *option* exists, and the **POSIX_BUILTINS** option hasn't been set, return 3 with a warning. If that option is set, return 1 with no warning.

−p *file* true if *file* exists and is a FIFO special file (named pipe).

−r *file* true if *file* exists and is readable by current process.

−s *file* true if *file* exists and has size greater than zero.

−t *fd* true if file descriptor number *fd* is open and associated with a terminal device. (note: *fd* is not optional)

−u *file* true if *file* exists and has its setuid bit set.

−v *varname*
true if shell variable *varname* is set.

−w *file* true if *file* exists and is writable by current process.

−x *file* true if *file* exists and is executable by current process. If *file* exists and is a directory, then the current process has permission to search in the directory.

−z *string*
true if length of *string* is zero.

−L *file* true if *file* exists and is a symbolic link.

−O *file* true if *file* exists and is owned by the effective user ID of this process.

−G *file* true if *file* exists and its group matches the effective group ID of this process.

−S *file* true if *file* exists and is a socket.

−N *file* true if *file* exists and its access time is not newer than its modification time.

file1 **−nt** *file2*
true if *file1* exists and is newer than *file2*.

file1 **−ot** *file2*
true if *file1* exists and is older than *file2*.

file1 **−ef** *file2*
true if *file1* and *file2* exist and refer to the same file.

string **=** *pattern*
string **==** *pattern*
true if *string* matches *pattern*. The two forms are exactly equivalent. The '**=**' form is the traditional shell syntax (and hence the only one generally used with the **test** and **[** builtins); the '**==**' form provides compatibility with other sorts of computer language.

string **!=** *pattern*
true if *string* does not match *pattern*.

string **=~** *regexp*
true if *string* matches the regular expression *regexp*. If the option **RE_MATCH_PCRE** is set *regexp* is tested as a PCRE regular expression using the **zsh/pcre** module, else it is tested as a POSIX extended regular expression using the **zsh/regex** module. Upon successful match, some variables will be updated; no variables are changed if the matching fails.

If the option **BASH_REMATCH** is not set the scalar parameter **MATCH** is set to the substring that matched the pattern and the integer parameters **MBEGIN** and **MEND** to the index of the start and end, respectively, of the match in *string*, such that if *string* is contained in variable **var** the expression '**${var[$MBEGIN,$MEND]}**' is identical to '**$MATCH**'. The setting of the option **KSH_ARRAYS** is respected. Likewise, the array **match** is set to the substrings that matched parenthesised subexpressions and the arrays **mbegin** and **mend** to the indices of the start and end positions, respectively, of the substrings within *string*. The arrays are not set if there were no parenthesised subexpressions. For example, if the string '**a short string**' is matched against the regular expression '**s(...)t**', then (assuming the option **KSH_ARRAYS** is not set) **MATCH**, **MBEGIN** and **MEND** are '**short**', **3** and **7**, respectively, while **match**, **mbegin** and **mend** are single entry arrays containing the strings '**hor**', '**4**' and '**6**', respectively.

If the option **BASH_REMATCH** is set the array **BASH_REMATCH** is set to the substring that matched the pattern followed by the substrings that matched parenthesised subexpressions within the pattern.

string1 **<** *string2*
true if *string1* comes before *string2* based on ASCII value of their characters.

string1 **>** *string2*
true if *string1* comes after *string2* based on ASCII value of their characters.

exp1 **−eq** *exp2*
true if *exp1* is numerically equal to *exp2*. Note that for purely numeric comparisons use of the **((...))** builtin described in the section 'ARITHMETIC EVALUATION' is more convenient than

conditional expressions.

exp1 **−ne** *exp2*
: true if *exp1* is numerically not equal to *exp2*.

exp1 **−lt** *exp2*
: true if *exp1* is numerically less than *exp2*.

exp1 **−gt** *exp2*
: true if *exp1* is numerically greater than *exp2*.

exp1 **−le** *exp2*
: true if *exp1* is numerically less than or equal to *exp2*.

exp1 **−ge** *exp2*
: true if *exp1* is numerically greater than or equal to *exp2*.

(*exp* **)**
: true if *exp* is true.

! *exp*
: true if *exp* is false.

exp1 **&&** *exp2*
: true if *exp1* and *exp2* are both true.

exp1 **||** *exp2*
: true if either *exp1* or *exp2* is true.

For compatibility, if there is a single argument that is not syntactically significant, typically a variable, the condition is treated as a test for whether the expression expands as a string of non−zero length. In other words, **[[$var]]** is the same as **[[−n $var]]**. It is recommended that the second, explicit, form be used where possible.

Normal shell expansion is performed on the *file*, *string* and *pattern* arguments, but the result of each expansion is constrained to be a single word, similar to the effect of double quotes.

Filename generation is not performed on any form of argument to conditions. However, it can be forced in any case where normal shell expansion is valid and when the option **EXTENDED_GLOB** is in effect by using an explicit glob qualifier of the form **(#q)** at the end of the string. A normal glob qualifier expression may appear between the '**q**' and the closing parenthesis; if none appears the expression has no effect beyond causing filename generation. The results of filename generation are joined together to form a single word, as with the results of other forms of expansion.

This special use of filename generation is only available with the **[[** syntax. If the condition occurs within the **[** or **test** builtin commands then globbing occurs instead as part of normal command line expansion before the condition is evaluated. In this case it may generate multiple words which are likely to confuse the syntax of the test command.

For example,

[[−n file*(#qN)]]

produces status zero if and only if there is at least one file in the current directory beginning with the string '**file**'. The globbing qualifier **N** ensures that the expression is empty if there is no matching file.

Pattern metacharacters are active for the *pattern* arguments; the patterns are the same as those used for filename generation, see *zshexpn*(1), but there is no special behaviour of '**/**' nor initial dots, and no glob qualifiers are allowed.

In each of the above expressions, if *file* is of the form '**/dev/fd/***n*', where *n* is an integer, then the test applied to the open file whose descriptor number is *n*, even if the underlying system does not support the **/dev/fd** directory.

In the forms which do numeric comparison, the expressions *exp* undergo arithmetic expansion as if they were enclosed in **$((...))**.

For example, the following:

[[(−f foo || −f bar) && $report = y*]] && print File exists.

tests if either file **foo** or file **bar** exists, and if so, if the value of the parameter **report** begins with '**y**'; if the complete condition is true, the message '**File exists.**' is printed.

EXPANSION OF PROMPT SEQUENCES

Prompt sequences undergo a special form of expansion. This type of expansion is also available using the **−P** option to the **print** builtin.

If the **PROMPT_SUBST** option is set, the prompt string is first subjected to *parameter expansion*, *command substitution* and *arithmetic expansion*. See *zshexpn*(1).

Certain escape sequences may be recognised in the prompt string.

If the **PROMPT_BANG** option is set, a '**!**' in the prompt is replaced by the current history event number. A literal '**!**' may then be represented as '**!!**'.

If the **PROMPT_PERCENT** option is set, certain escape sequences that start with '**%**' are expanded. Many escapes are followed by a single character, although some of these take an optional integer argument that should appear between the '**%**' and the next character of the sequence. More complicated escape sequences are available to provide conditional expansion.

SIMPLE PROMPT ESCAPES

Special characters

%% A '**%**'.

%) A ')'.

Login information

%l The line (tty) the user is logged in on, without '**/dev/**' prefix. If the name starts with '**/dev/tty**', that prefix is stripped.

%M The full machine hostname.

%m The hostname up to the first '**.**'. An integer may follow the '**%**' to specify how many components of the hostname are desired. With a negative integer, trailing components of the hostname are shown.

%n **$USERNAME**.

%y The line (tty) the user is logged in on, without '**/dev/**' prefix. This does not treat '**/dev/tty**' names specially.

Shell state

%# A '**#**' if the shell is running with privileges, a '**%**' if not. Equivalent to '**%(!.#.%%)**'. The definition of 'privileged', for these purposes, is that either the effective user ID is zero, or, if POSIX.1e capabilities are supported, that at least one capability is raised in either the Effective or Inheritable capability vectors.

%? The return status of the last command executed just before the prompt.

%_ The status of the parser, i.e. the shell constructs (like '**if**' and '**for**') that have been started on the command line. If given an integer number that many strings will be printed; zero or negative or no integer means print as many as there are. This is most useful in prompts **PS2** for continuation lines and **PS4** for debugging with the **XTRACE** option; in the latter case it will also work non−interactively.

%^ The status of the parser in reverse. This is the same as '**%_**' other than the order of strings. It is often used in **RPS2**.

%d

%/ Current working directory. If an integer follows the '**%**', it specifies a number of trailing components of the current working directory to show; zero means the whole path. A negative integer specifies leading components, i.e. **%−1d** specifies the first component.

%~ As **%d** and **%/**, but if the current working directory starts with **$HOME**, that part is replaced by a '~'. Furthermore, if it has a named directory as its prefix, that part is replaced by a '~' followed by the name of the directory, but only if the result is shorter than the full path; see *Dynamic* and *Static named directories* in *zshexpn*(1).

%e Evaluation depth of the current sourced file, shell function, or **eval**. This is incremented or decremented every time the value of **%N** is set or reverted to a previous value, respectively. This is most useful for debugging as part of **$PS4**.

%h
%! Current history event number.

%i The line number currently being executed in the script, sourced file, or shell function given by **%N**. This is most useful for debugging as part of **$PS4**.

%I The line number currently being executed in the file **%x**. This is similar to **%i**, but the line number is always a line number in the file where the code was defined, even if the code is a shell function.

%j The number of jobs.

%L The current value of **$SHLVL**.

%N The name of the script, sourced file, or shell function that zsh is currently executing, whichever was started most recently. If there is none, this is equivalent to the parameter **$0**. An integer may follow the '**%**' to specify a number of trailing path components to show; zero means the full path. A negative integer specifies leading components.

%x The name of the file containing the source code currently being executed. This behaves as **%N** except that function and eval command names are not shown, instead the file where they were defined.

%c
%.
%C Trailing component of the current working directory. An integer may follow the '**%**' to get more than one component. Unless '**%C**' is used, tilde contraction is performed first. These are deprecated as **%c** and **%C** are equivalent to **%1~** and **%1/**, respectively, while explicit positive integers have the same effect as for the latter two sequences.

Date and time

%D The date in *yy*−*mm*−*dd* format.

%T Current time of day, in 24−hour format.

%t
%@ Current time of day, in 12−hour, am/pm format.

%* Current time of day in 24−hour format, with seconds.

%w The date in *day*−*dd* format.

%W The date in *mm*/*dd*/*yy* format.

%D{*string***}**

string is formatted using the **strftime** function. See *strftime*(3) for more details. Various zsh extensions provide numbers with no leading zero or space if the number is a single digit:

%f a day of the month
%K the hour of the day on the 24−hour clock
%L the hour of the day on the 12−hour clock

In addition, if the system supports the POSIX **gettimeofday** system call, **%.** provides decimal fractions of a second since the epoch with leading zeroes. By default three decimal places are provided, but a number of digits up to 9 may be given following the **%**; hence **%6.** outputs microseconds, and **%9.** outputs nanoseconds. (The latter requires a nanosecond−precision **clock_gettime**;

systems lacking this will return a value multiplied by the appropriate power of 10.) A typical example of this is the format '**%D{%H:%M:%S.%.}**'.

The GNU extension **%N** is handled as a synonym for **%9.**.

Additionally, the GNU extension that a '**-**' between the **%** and the format character causes a leading zero or space to be stripped is handled directly by the shell for the format characters **d**, **f**, **H**, **k**, **l**, **m**, **M**, **S** and **y**; any other format characters are provided to the system's strftime(3) with any leading '**-**' present, so the handling is system dependent. Further GNU (or other) extensions are also passed to strftime(3) and may work if the system supports them.

Visual effects

%B (**%b**)
Start (stop) boldface mode.

%E Clear to end of line.

%U (**%u**)
Start (stop) underline mode.

%S (**%s**)
Start (stop) standout mode.

%F (**%f**)
Start (stop) using a different foreground colour, if supported by the terminal. The colour may be specified two ways: either as a numeric argument, as normal, or by a sequence in braces following the **%F**, for example **%F{red}**. In the latter case the values allowed are as described for the **fg zle_highlight** attribute; see *Character Highlighting* in *zshzle*(1). This means that numeric colours are allowed in the second format also.

%K (**%k**)
Start (stop) using a different bacKground colour. The syntax is identical to that for **%F** and **%f**.

%{...**%}**
Include a string as a literal escape sequence. The string within the braces should not change the cursor position. Brace pairs can nest.

A positive numeric argument between the **%** and the **{** is treated as described for **%G** below.

%G Within a **%{**...**%}** sequence, include a 'glitch': that is, assume that a single character width will be output. This is useful when outputting characters that otherwise cannot be correctly handled by the shell, such as the alternate character set on some terminals. The characters in question can be included within a **%{**...**%}** sequence together with the appropriate number of **%G** sequences to indicate the correct width. An integer between the '**%**' and '**G**' indicates a character width other than one. Hence **%{***seq***%2G%}** outputs *seq* and assumes it takes up the width of two standard characters.

Multiple uses of **%G** accumulate in the obvious fashion; the position of the **%G** is unimportant. Negative integers are not handled.

Note that when prompt truncation is in use it is advisable to divide up output into single characters within each **%{**...**%}** group so that the correct truncation point can be found.

CONDITIONAL SUBSTRINGS IN PROMPTS

%v The value of the first element of the **psvar** array parameter. Following the '**%**' with an integer gives that element of the array. Negative integers count from the end of the array.

%(*x***.***true*−*text***.***false*−*text***)**
Specifies a ternary expression. The character following the *x* is arbitrary; the same character is used to separate the text for the 'true' result from that for the 'false' result. This separator may not appear in the *true*−*text*, except as part of a %−escape sequence. A '**)**' may appear in the *false*−*text* as '**%)**'. *true*−*text* and *false*−*text* may both contain arbitrarily−nested escape sequences, including further ternary expressions.

The left parenthesis may be preceded or followed by a positive integer *n*, which defaults to zero. A negative integer will be multiplied by −1, except as noted below for '**l**'. The test character *x* may be any of the following:

! True if the shell is running with privileges.
True if the effective uid of the current process is *n*.
? True if the exit status of the last command was *n*.
_ True if at least *n* shell constructs were started.
C
/ True if the current absolute path has at least *n* elements relative to the root directory, hence **/** is counted as 0 elements.
c
.
~ True if the current path, with prefix replacement, has at least *n* elements relative to the root directory, hence **/** is counted as 0 elements.
D True if the month is equal to *n* (January = 0).
d True if the day of the month is equal to *n*.
e True if the evaluation depth is at least *n*.
g True if the effective gid of the current process is *n*.
j True if the number of jobs is at least *n*.
L True if the **SHLVL** parameter is at least *n*.
l True if at least *n* characters have already been printed on the current line. When *n* is negative, true if at least **abs**(*n*) characters remain before the opposite margin (thus the left margin for **RPROMPT**).
S True if the **SECONDS** parameter is at least *n*.
T True if the time in hours is equal to *n*.
t True if the time in minutes is equal to *n*.
v True if the array **psvar** has at least *n* elements.
V True if element *n* of the array **psvar** is set and non−empty.
w True if the day of the week is equal to *n* (Sunday = 0).

%<*string***<**
%>*string***>**
%[*xstring***]**

Specifies truncation behaviour for the remainder of the prompt string. The third, deprecated, form is equivalent to '**%***x***string***x***', i.e. *x* may be '**<**' or '**>**'. The *string* will be displayed in place of the truncated portion of any string; note this does not undergo prompt expansion.

The numeric argument, which in the third form may appear immediately after the '**[**', specifies the maximum permitted length of the various strings that can be displayed in the prompt. In the first two forms, this numeric argument may be negative, in which case the truncation length is determined by subtracting the absolute value of the numeric argument from the number of character positions remaining on the current prompt line. If this results in a zero or negative length, a length of 1 is used. In other words, a negative argument arranges that after truncation at least *n* characters remain before the right margin (left margin for **RPROMPT**).

The forms with '**<**' truncate at the left of the string, and the forms with '**>**' truncate at the right of the string. For example, if the current directory is '**/home/pike**', the prompt '**%8<..<%/**' will expand to '**..e/pike**'. In this string, the terminating character ('**<**', '**>**' or '**]**'), or in fact any character, may be quoted by a preceding '****'; note when using **print −P**, however, that this must be doubled as the string is also subject to standard **print** processing, in addition to any backslashes removed by a double quoted string: the worst case is therefore '**print −P "%<\\\\<<..."**'.

If the *string* is longer than the specified truncation length, it will appear in full, completely replacing the truncated string.

The part of the prompt string to be truncated runs to the end of the string, or to the end of the next enclosing group of the '**%(**' construct, or to the next truncation encountered at the same grouping

level (i.e. truncations inside a '**%(**' are separate), which ever comes first. In particular, a truncation with argument zero (e.g., '**%<<**') marks the end of the range of the string to be truncated while turning off truncation from there on. For example, the prompt '**%10<...<%~%<<%#** ' will print a truncated representation of the current directory, followed by a '**%**' or '**#**', followed by a space. Without the '**%<<**', those two characters would be included in the string to be truncated. Note that '**%-0<<**' is not equivalent to '**%<<**' but specifies that the prompt is truncated at the right margin.

Truncation applies only within each individual line of the prompt, as delimited by embedded newlines (if any). If the total length of any line of the prompt after truncation is greater than the terminal width, or if the part to be truncated contains embedded newlines, truncation behavior is undefined and may change in a future version of the shell. Use '**%-***n***(l.***true-text***.***false-text***)**' to remove parts of the prompt when the available space is less than *n*.