NAME

zshparam − zsh parameters

DESCRIPTION

A parameter has a name, a value, and a number of attributes. A name may be any sequence of alphanumeric characters and underscores, or the single characters '*****', '**@**', '**#**', '**?**', '**−**', '**$**', or '**!**'. A parameter whose name begins with an alphanumeric or underscore is also referred to as a *variable*.

The attributes of a parameter determine the *type* of its value, often referred to as the parameter type or variable type, and also control other processing that may be applied to the value when it is referenced. The value type may be a *scalar* (a string, an integer, or a floating point number), an array (indexed numerically), or an *associative* array (an unordered set of name−value pairs, indexed by name, also referred to as a *hash*).

Named scalar parameters may have the *exported*, **−x**, attribute, to copy them into the process environment, which is then passed from the shell to any new processes that it starts. Exported parameters are called *environment variables*. The shell also *imports* environment variables at startup time and automatically marks the corresponding parameters as exported. Some environment variables are not imported for reasons of security or because they would interfere with the correct operation of other shell features.

Parameters may also be *special*, that is, they have a predetermined meaning to the shell. Special parameters cannot have their type changed or their readonly attribute turned off, and if a special parameter is unset, then later recreated, the special properties will be retained.

To declare the type of a parameter, or to assign a string or numeric value to a scalar parameter, use the **typeset** builtin.

The value of a scalar parameter may also be assigned by writing:

*name***=***value*

In scalar assignment, *value* is expanded as a single string, in which the elements of arrays are joined together; filename expansion is not performed unless the option **GLOB_ASSIGN** is set.

When the integer attribute, **−i**, or a floating point attribute, **−E** or **−F**, is set for *name*, the *value* is subject to arithmetic evaluation. Furthermore, by replacing '**=**' with '**+=**', a parameter can be incremented or appended to. See the section 'Array Parameters' and *Arithmetic Evaluation* (in *zshmisc*(1)) for additional forms of assignment.

Note that assignment may implicitly change the attributes of a parameter. For example, assigning a number to a variable in arithmetic evaluation may change its type to integer or float, and with **GLOB_ASSIGN** assigning a pattern to a variable may change its type to an array.

To reference the value of a parameter, write '**$***name*' or '**${***name***}**'. See *Parameter Expansion* in *zshexpn*(1) for complete details. That section also explains the effect of the difference between scalar and array assignment on parameter expansion.

ARRAY PARAMETERS

To assign an array value, write one of:

set −A *name value ...*
*name***=(***value ...***)**
*name***=([***key***]=***value ...***)**

If no parameter *name* exists, an ordinary array parameter is created. If the parameter *name* exists and is a scalar, it is replaced by a new array.

In the third form, *key* is an expression that will be evaluated in arithmetic context (in its simplest form, an integer) that gives the index of the element to be assigned with *value*. In this form any elements not explicitly mentioned that come before the largest index to which a value is assigned are assigned an empty string. The indices may be in any order. Note that this syntax is strict: **[** and **]=** must not be quoted, and *key* may not consist of the unquoted string **]=**, but is otherwise treated as a simple string. The enhanced forms of subscript expression that may be used when directly subscripting a variable name, described in the section

Array Subscripts below, are not available.

The syntaxes with and without the explicit key may be mixed. An implicit *key* is deduced by incrementing the index from the previously assigned element. Note that it is not treated as an error if latter assignments in this form overwrite earlier assignments.

For example, assuming the option **KSH_ARRAYS** is not set, the following:

array=(one [3]=three four)

causes the array variable **array** to contain four elements **one**, an empty string, **three** and **four**, in that order.

In the forms where only *value* is specified, full command line expansion is performed.

In the **[***key***]=***value* form, both *key* and *value* undergo all forms of expansion allowed for single word shell expansions (this does not include filename generation); these are as performed by the parameter expansion flag **(e)** as described in *zshexpn*(1). Nested parentheses may surround *value* and are included as part of the value, which is joined into a plain string; this differs from ksh which allows the values themselves to be arrays. A future version of zsh may support that. To cause the brackets to be interpreted as a character class for filename generation, and therefore to treat the resulting list of files as a set of values, quote the equal sign using any form of quoting. Example:

*name***=([a−z]'='*)**

To append to an array without changing the existing values, use one of the following:

*name***+=(***value ...***)**
*name***+=([***key***]=***value ...***)**

In the second form *key* may specify an existing index as well as an index off the end of the old array; any existing value is overwritten by *value*. Also, it is possible to use **[***key***]+=***value* to append to the existing value at that index.

Within the parentheses on the right hand side of either form of the assignment, newlines and semicolons are treated the same as white space, separating individual *value*s. Any consecutive sequence of such characters has the same effect.

Ordinary array parameters may also be explicitly declared with:

typeset −a *name*

Associative arrays *must* be declared before assignment, by using:

typeset −A *name*

When *name* refers to an associative array, the list in an assignment is interpreted as alternating keys and values:

set −A *name key value ...*
*name***=(***key value ...***)**
*name***=([***key***]=***value ...***)**

Note that only one of the two syntaxes above may be used in any given assignment; the forms may not be mixed. This is unlike the case of numerically indexed arrays.

Every *key* must have a *value* in this case. Note that this assigns to the entire array, deleting any elements that do not appear in the list. The append syntax may also be used with an associative array:

*name***+=(***key value ...***)**
*name***+=([***key***]=***value ...***)**

This adds a new key/value pair if the key is not already present, and replaces the value for the existing key if it is. In the second form it is also possible to use **[***key***]+=***value* to append to the existing value at that key.

Expansion is performed identically to the corresponding forms for normal arrays, as described above.

To create an empty array (including associative arrays), use one of:

set −A *name*
*name***=()**

Array Subscripts

Individual elements of an array may be selected using a subscript. A subscript of the form '**[***exp***]**' selects the single element *exp*, where *exp* is an arithmetic expression which will be subject to arithmetic expansion as if it were surrounded by '**$((...))**'. The elements are numbered beginning with 1, unless the **KSH_ARRAYS** option is set in which case they are numbered from zero.

Subscripts may be used inside braces used to delimit a parameter name, thus '**${foo[2]}**' is equivalent to '**$foo[2]**'. If the **KSH_ARRAYS** option is set, the braced form is the only one that works, as bracketed expressions otherwise are not treated as subscripts.

If the **KSH_ARRAYS** option is not set, then by default accesses to an array element with a subscript that evaluates to zero return an empty string, while an attempt to write such an element is treated as an error. For backward compatibility the **KSH_ZERO_SUBSCRIPT** option can be set to cause subscript values 0 and 1 to be equivalent; see the description of the option in *zshoptions*(1).

The same subscripting syntax is used for associative arrays, except that no arithmetic expansion is applied to *exp*. However, the parsing rules for arithmetic expressions still apply, which affects the way that certain special characters must be protected from interpretation. See *Subscript Parsing* below for details.

A subscript of the form '**[*]**' or '**[@]**' evaluates to all elements of an array; there is no difference between the two except when they appear within double quotes. '**"$foo[*]"**' evaluates to '**"$foo[1] $foo[2] ..."**', whereas '**"$foo[@]"**' evaluates to '**"$foo[1]" "$foo[2]" ...**'. For associative arrays, '**[*]**' or '**[@]**' evaluate to all the values, in no particular order. Note that this does not substitute the keys; see the documentation for the '**k**' flag under *Parameter Expansion Flags* in *zshexpn*(1) for complete details. When an array parameter is referenced as '**$***name*' (with no subscript) it evaluates to '**$***name***[*]**', unless the **KSH_ARRAYS** option is set in which case it evaluates to '**${***name***[0]}**' (for an associative array, this means the value of the key '**0**', which may not exist even if there are values for other keys).

A subscript of the form '**[***exp1***,***exp2***]**' selects all elements in the range *exp1* to *exp2*, inclusive. (Associative arrays are unordered, and so do not support ranges.) If one of the subscripts evaluates to a negative number, say **−***n*, then the *n*th element from the end of the array is used. Thus '**$foo[−3]**' is the third element from the end of the array **foo**, and '**$foo[1,−1]**' is the same as '**$foo[*]**'.

Subscripting may also be performed on non−array values, in which case the subscripts specify a substring to be extracted. For example, if **FOO** is set to '**foobar**', then '**echo $FOO[2,5]**' prints '**ooba**'. Note that some forms of subscripting described below perform pattern matching, and in that case the substring extends from the start of the match of the first subscript to the end of the match of the second subscript. For example,

```
string="abcdefghijklm"
print ${string[(r)d?,(r)h?]}
```

prints '**defghi**'. This is an obvious generalisation of the rule for single−character matches. For a single subscript, only a single character is referenced (not the range of characters covered by the match).

Note that in substring operations the second subscript is handled differently by the **r** and **R** subscript flags: the former takes the shortest match as the length and the latter the longest match. Hence in the former case a ***** at the end is redundant while in the latter case it matches the whole remainder of the string. This does not affect the result of the single subscript case as here the length of the match is irrelevant.

Array Element Assignment

A subscript may be used on the left side of an assignment like so:

*name***[***exp***]=***value*

In this form of assignment the element or range specified by *exp* is replaced by the expression on the right side. An array (but not an associative array) may be created by assignment to a range or element. Arrays do not nest, so assigning a parenthesized list of values to an element or range changes the number of elements in the array, shifting the other elements to accommodate the new values. (This is not supported for associative arrays.)

This syntax also works as an argument to the **typeset** command:

typeset "*name***[***exp***]"=***value*

The *value* may *not* be a parenthesized list in this case; only single−element assignments may be made with **typeset**. Note that quotes are necessary in this case to prevent the brackets from being interpreted as filename generation operators. The **noglob** precommand modifier could be used instead.

To delete an element of an ordinary array, assign '()' to that element. To delete an element of an associative array, use the **unset** command:

unset "*name***[***exp***]"**

Subscript Flags

If the opening bracket, or the comma in a range, in any subscript expression is directly followed by an opening parenthesis, the string up to the matching closing one is considered to be a list of flags, as in '*name***[(***flags***)***exp***]**'.

The flags **s**, **n** and **b** take an argument; the delimiter is shown below as '**:**', but any character, or the matching pairs '(...)', '{...}', '[...]', or '<...>', may be used, but note that '<...>' can only be used if the subscript is inside a double quoted expression or a parameter substitution enclosed in braces as otherwise the expression is interpreted as a redirection.

The flags currently understood are:

w If the parameter subscripted is a scalar then this flag makes subscripting work on words instead of characters. The default word separator is whitespace. When combined with the **i** or **I** flag, the effect is to produce the index of the first character of the first/last word which matches the given pattern; note that a failed match in this case always yields 0.

s:*string***:**
This gives the *string* that separates words (for use with the **w** flag). The delimiter character **:** is arbitrary; see above.

p Recognize the same escape sequences as the **print** builtin in the string argument of a subsequent '**s**' flag.

f If the parameter subscripted is a scalar then this flag makes subscripting work on lines instead of characters, i.e. with elements separated by newlines. This is a shorthand for '**pws:\n:**'.

r Reverse subscripting: if this flag is given, the *exp* is taken as a pattern and the result is the first matching array element, substring or word (if the parameter is an array, if it is a scalar, or if it is a scalar and the '**w**' flag is given, respectively). The subscript used is the number of the matching element, so that pairs of subscripts such as '**$foo[(r)??,3]**' and '**$foo[(r)??,(r)f*]**' are possible if the parameter is not an associative array. If the parameter is an associative array, only the value part of each pair is compared to the pattern, and the result is that value.

If a search through an ordinary array failed, the search sets the subscript to one past the end of the array, and hence **${array[(r)***pattern***]}** will substitute the empty string. Thus the success of a search can be tested by using the **(i)** flag, for example (assuming the option **KSH_ARRAYS** is not in effect):

[[${array[(i)pattern]} −le ${#array}]]

If **KSH_ARRAYS** is in effect, the **−le** should be replaced by **−lt**.

R Like '**r**', but gives the last match. For associative arrays, gives all possible matches. May be used for assigning to ordinary array elements, but not for assigning to associative arrays. On failure, for normal arrays this has the effect of returning the element corresponding to subscript 0; this is empty unless one of the options **KSH_ARRAYS** or **KSH_ZERO_SUBSCRIPT** is in effect.

Note that in subscripts with both '**r**' and '**R**' pattern characters are active even if they were substituted for a parameter (regardless of the setting of **GLOB_SUBST** which controls this feature in normal pattern matching). The flag '**e**' can be added to inhibit pattern matching. As this flag does not inhibit other forms of substitution, care is still required; using a parameter to hold the key has the desired effect:

```
key2='original key'
print ${array[(Re)$key2]}
```

i Like '**r**', but gives the index of the match instead; this may not be combined with a second argument. On the left side of an assignment, behaves like '**r**'. For associative arrays, the key part of each pair is compared to the pattern, and the first matching key found is the result. On failure substitutes the length of the array plus one, as discussed under the description of '**r**', or the empty string for an associative array.

I Like '**i**', but gives the index of the last match, or all possible matching keys in an associative array. On failure substitutes 0, or the empty string for an associative array. This flag is best when testing for values or keys that do not exist.

k If used in a subscript on an associative array, this flag causes the keys to be interpreted as patterns, and returns the value for the first key found where *exp* is matched by the key. Note this could be any such key as no ordering of associative arrays is defined. This flag does not work on the left side of an assignment to an associative array element. If used on another type of parameter, this behaves like '**r**'.

K On an associative array this is like '**k**' but returns all values where *exp* is matched by the keys. On other types of parameters this has the same effect as '**R**'.

n:*expr***:** If combined with '**r**', '**R**', '**i**' or '**I**', makes them give the *n*th or *n*th last match (if *expr* evaluates to *n*). This flag is ignored when the array is associative. The delimiter character **:** is arbitrary; see above.

b:*expr***:** If combined with '**r**', '**R**', '**i**' or '**I**', makes them begin at the *n*th or *n*th last element, word, or character (if *expr* evaluates to *n*). This flag is ignored when the array is associative. The delimiter character **:** is arbitrary; see above.

e This flag causes any pattern matching that would be performed on the subscript to use plain string matching instead. Hence '**${array[(re)*]}**' matches only the array element whose value is *****. Note that other forms of substitution such as parameter substitution are not inhibited.

This flag can also be used to force ***** or **@** to be interpreted as a single key rather than as a reference to all values. It may be used for either purpose on the left side of an assignment.

See *Parameter Expansion Flags* (*zshexpn*(1)) for additional ways to manipulate the results of array subscripting.

Subscript Parsing

This discussion applies mainly to associative array key strings and to patterns used for reverse subscripting (the '**r**', '**R**', '**i**', etc. flags), but it may also affect parameter substitutions that appear as part of an arithmetic expression in an ordinary subscript.

To avoid subscript parsing limitations in assignments to associative array elements, use the append syntax:

```
aa+=('key with "*strange*" characters' 'value string')
```

The basic rule to remember when writing a subscript expression is that all text between the opening '**[**' and the closing '**]**' is interpreted *as if* it were in double quotes (see *zshmisc*(1)). However, unlike double quotes which normally cannot nest, subscript expressions may appear inside double−quoted strings or inside other

subscript expressions (or both!), so the rules have two important differences.

The first difference is that brackets (‘**[**’ and ‘**]**’) must appear as balanced pairs in a subscript expression unless they are preceded by a backslash (‘\’). Therefore, within a subscript expression (and unlike true double−quoting) the sequence ‘**\[**’ becomes ‘**[**’, and similarly ‘**\]**’ becomes ‘**]**’. This applies even in cases where a backslash is not normally required; for example, the pattern ‘**[^[]**’ (to match any character other than an open bracket) should be written ‘**[^\[]**’ in a reverse−subscript pattern. However, note that ‘**\[^\[\]**’ and even ‘**\[^[]**’ mean the *same* thing, because backslashes are always stripped when they appear before brackets!

The same rule applies to parentheses (‘**(**’ and ‘**)**’) and braces (‘**{**’ and ‘**}**’): they must appear either in balanced pairs or preceded by a backslash, and backslashes that protect parentheses or braces are removed during parsing. This is because parameter expansions may be surrounded by balanced braces, and subscript flags are introduced by balanced parentheses.

The second difference is that a double−quote (‘**"**’) may appear as part of a subscript expression without being preceded by a backslash, and therefore that the two characters ‘**\"**’ remain as two characters in the subscript (in true double−quoting, ‘**\"**’ becomes ‘**"**’). However, because of the standard shell quoting rules, any double−quotes that appear must occur in balanced pairs unless preceded by a backslash. This makes it more difficult to write a subscript expression that contains an odd number of double−quote characters, but the reason for this difference is so that when a subscript expression appears inside true double−quotes, one can still write ‘**\"**’ (rather than ‘**\\\"**’) for ‘**"**’.

To use an odd number of double quotes as a key in an assignment, use the **typeset** builtin and an enclosing pair of double quotes; to refer to the value of that key, again use double quotes:

```
typeset −A aa
typeset "aa[one\"two\"three\"quotes]"=QQQ
print "$aa[one\"two\"three\"quotes]"
```

It is important to note that the quoting rules do not change when a parameter expansion with a subscript is nested inside another subscript expression. That is, it is not necessary to use additional backslashes within the inner subscript expression; they are removed only once, from the innermost subscript outwards. Parameters are also expanded from the innermost subscript first, as each expansion is encountered left to right in the outer expression.

A further complication arises from a way in which subscript parsing is *not* different from double quote parsing. As in true double−quoting, the sequences ‘*****’, and ‘**\@**’ remain as two characters when they appear in a subscript expression. To use a literal ‘*****’ or ‘**@**’ as an associative array key, the ‘**e**’ flag must be used:

```
typeset −A aa
aa[(e)*]=star
print $aa[(e)*]
```

A last detail must be considered when reverse subscripting is performed. Parameters appearing in the subscript expression are first expanded and then the complete expression is interpreted as a pattern. This has two effects: first, parameters behave as if **GLOB_SUBST** were on (and it cannot be turned off); second, backslashes are interpreted twice, once when parsing the array subscript and again when parsing the pattern. In a reverse subscript, it’s necessary to use *four* backslashes to cause a single backslash to match literally in the pattern. For complex patterns, it is often easiest to assign the desired pattern to a parameter and then refer to that parameter in the subscript, because then the backslashes, brackets, parentheses, etc., are seen only when the complete expression is converted to a pattern. To match the value of a parameter literally in a reverse subscript, rather than as a pattern, use ‘**${(q)***name***}**’ (see *zshexpn*(1)) to quote the expanded value.

Note that the ‘**k**’ and ‘**K**’ flags are reverse subscripting for an ordinary array, but are *not* reverse subscripting for an associative array! (For an associative array, the keys in the array itself are interpreted as patterns by those flags; the subscript is a plain string in that case.)

One final note, not directly related to subscripting: the numeric names of positional parameters (described below) are parsed specially, so for example ‘**$2foo**’ is equivalent to ‘**${2}foo**’. Therefore, to use subscript

syntax to extract a substring from a positional parameter, the expansion must be surrounded by braces; for example, '**${2[3,5]}**' evaluates to the third through fifth characters of the second positional parameter, but '**$2[3,5]**' is the entire second parameter concatenated with the filename generation pattern '**[3,5]**'.

POSITIONAL PARAMETERS

The positional parameters provide access to the command−line arguments of a shell function, shell script, or the shell itself; see the section 'Invocation', and also the section 'Functions'. The parameter *n*, where *n* is a number, is the *n*th positional parameter. The parameter '**$0**' is a special case, see the section 'Parameters Set By The Shell'.

The parameters *****, **@** and **argv** are arrays containing all the positional parameters; thus '**$argv[***n***]**', etc., is equivalent to simply '**$***n*'. Note that the options **KSH_ARRAYS** or **KSH_ZERO_SUBSCRIPT** apply to these arrays as well, so with either of those options set, '**${argv[0]}**' is equivalent to '**$1**' and so on.

Positional parameters may be changed after the shell or function starts by using the **set** builtin, by assigning to the **argv** array, or by direct assignment of the form '*n***=***value*' where *n* is the number of the positional parameter to be changed. This also creates (with empty values) any of the positions from 1 to *n* that do not already have values. Note that, because the positional parameters form an array, an array assignment of the form '*n***=(***value ...***)**' is allowed, and has the effect of shifting all the values at positions greater than *n* by as many positions as necessary to accommodate the new values.

LOCAL PARAMETERS

Shell function executions delimit scopes for shell parameters. (Parameters are dynamically scoped.) The **typeset** builtin, and its alternative forms **declare**, **integer**, **local** and **readonly** (but not **export**), can be used to declare a parameter as being local to the innermost scope.

When a parameter is read or assigned to, the innermost existing parameter of that name is used. (That is, the local parameter hides any less−local parameter.) However, assigning to a non−existent parameter, or declaring a new parameter with **export**, causes it to be created in the *outer*most scope.

Local parameters disappear when their scope ends. **unset** can be used to delete a parameter while it is still in scope; any outer parameter of the same name remains hidden.

Special parameters may also be made local; they retain their special attributes unless either the existing or the newly−created parameter has the **−h** (hide) attribute. This may have unexpected effects: there is no default value, so if there is no assignment at the point the variable is made local, it will be set to an empty value (or zero in the case of integers). The following:

typeset PATH=/new/directory:$PATH

is valid for temporarily allowing the shell or programmes called from it to find the programs in **/new/directory** inside a function.

Note that the restriction in older versions of zsh that local parameters were never exported has been removed.

PARAMETERS SET BY THE SHELL

In the parameter lists that follow, the mark '<S>' indicates that the parameter is special. '<Z>' indicates that the parameter does not exist when the shell initializes in **sh** or **ksh** emulation mode.

The following parameters are automatically set by the shell:

! <S> The process ID of the last command started in the background with **&**, put into the background with the **bg** builtin, or spawned with **coproc**.

<S> The number of positional parameters in decimal. Note that some confusion may occur with the syntax **$#***param* which substitutes the length of *param*. Use **${#}** to resolve ambiguities. In particular, the sequence '**$#−**...' in an arithmetic expression is interpreted as the length of the parameter **−**, q.v.

ARGC <S> <Z>
Same as **#**.

$ <S> The process ID of this shell. Note that this indicates the original shell started by invoking **zsh**; all processes forked from the shells without executing a new program, such as subshells started by **(...)**, substitute the same value.

– <S> Flags supplied to the shell on invocation or by the **set** or **setopt** commands.

***** <S> An array containing the positional parameters.

argv <S> <Z>
Same as *****. Assigning to **argv** changes the local positional parameters, but **argv** is *not* itself a local parameter. Deleting **argv** with **unset** in any function deletes it everywhere, although only the innermost positional parameter array is deleted (so ***** and @ in other scopes are not affected).

@ <S> Same as **argv[@]**, even when **argv** is not set.

? <S> The exit status returned by the last command.

0 <S> The name used to invoke the current shell, or as set by the **–c** command line option upon invocation. If the **FUNCTION_ARGZERO** option is set, **$0** is set upon entry to a shell function to the name of the function, and upon entry to a sourced script to the name of the script, and reset to its previous value when the function or script returns.

status <S> <Z>
Same as **?**.

pipestatus <S> <Z>
An array containing the exit statuses returned by all commands in the last pipeline.

_ <S> The last argument of the previous command. Also, this parameter is set in the environment of every command executed to the full pathname of the command.

CPUTYPE
The machine type (microprocessor class or machine model), as determined at run time.

EGID <S>
The effective group ID of the shell process. If you have sufficient privileges, you may change the effective group ID of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command with a different effective group ID by '**(EGID=***gid***; command)**'

If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

EUID <S>
The effective user ID of the shell process. If you have sufficient privileges, you may change the effective user ID of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command with a different effective user ID by '**(EUID=***uid***; command)**'

If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

ERRNO <S>
The value of errno (see *errno*(3)) as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes. It is also useful with the **zsh/system** module which allows the number to be turned into a name or message.

FUNCNEST <S>
Integer. If greater than or equal to zero, the maximum nesting depth of shell functions. When it is exceeded, an error is raised at the point where a function is called. The default value is determined when the shell is configured, but is typically 500. Increasing the value increases the danger of a runaway function recursion causing the shell to crash. Setting a negative value turns off the check.

GID <S>
The real group ID of the shell process. If you have sufficient privileges, you may change the group ID of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different group ID by '**(GID=***gid***; command)**'

If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

HISTCMD
The current history event number in an interactive shell, in other words the event number for the command that caused **$HISTCMD** to be read. If the current history event modifies the history, **HISTCMD** changes to the new maximum history event number.

HOST The current hostname.

LINENO <S>
The line number of the current line within the current script, sourced file, or shell function being executed, whichever was started most recently. Note that in the case of shell functions the line number refers to the function as it appeared in the original definition, not necessarily as displayed by the **functions** builtin.

LOGNAME
If the corresponding variable is not set in the environment of the shell, it is initialized to the login name corresponding to the current login session. This parameter is exported by default but this can be disabled using the **typeset** builtin. The value is set to the string returned by the *getlogin*(3) system call if that is available.

MACHTYPE
The machine type (microprocessor class or machine model), as determined at compile time.

OLDPWD
The previous working directory. This is set when the shell initializes and whenever the directory changes.

OPTARG <S>
The value of the last option argument processed by the **getopts** command.

OPTIND <S>
The index of the last option argument processed by the **getopts** command.

OSTYPE
The operating system, as determined at compile time.

PPID <S>
The process ID of the parent of the shell. As for **$$**, the value indicates the parent of the original shell and does not change in subshells.

PWD The present working directory. This is set when the shell initializes and whenever the directory changes.

RANDOM <S>
A pseudo−random integer from 0 to 32767, newly generated each time this parameter is referenced. The random number generator can be seeded by assigning a numeric value to **RANDOM**.

The values of **RANDOM** form an intentionally−repeatable pseudo−random sequence; subshells that reference **RANDOM** will result in identical pseudo−random values unless the value of **RANDOM** is referenced or seeded in the parent shell in between subshell invocations.

SECONDS <S>
The number of seconds since shell invocation. If this parameter is assigned a value, then the value returned upon reference will be the value that was assigned plus the number of seconds since the assignment.

Unlike other special parameters, the type of the **SECONDS** parameter can be changed using the **typeset** command. Only integer and one of the floating point types are allowed. For example, '**typeset −F SECONDS**' causes the value to be reported as a floating point number. The value is available to microsecond accuracy, although the shell may show more or fewer digits depending on the use of **typeset**. See the documentation for the builtin **typeset** in *zshbuiltins*(1) for more details.

SHLVL <S>
Incremented by one each time a new shell is started.

signals An array containing the names of the signals. Note that with the standard zsh numbering of array indices, where the first element has index 1, the signals are offset by 1 from the signal number used by the operating system. For example, on typical Unix−like systems **HUP** is signal number 1, but is referred to as **$signals[2]**. This is because of **EXIT** at position 1 in the array, which is used internally by zsh but is not known to the operating system.

TRY_BLOCK_ERROR <S>
In an **always** block, indicates whether the preceding list of code caused an error. The value is 1 to indicate an error, 0 otherwise. It may be reset, clearing the error condition. See *Complex Commands* in *zshmisc*(1)

TRY_BLOCK_INTERRUPT <S>
This variable works in a similar way to **TRY_BLOCK_ERROR**, but represents the status of an interrupt from the signal SIGINT, which typically comes from the keyboard when the user types **ˆC**. If set to 0, any such interrupt will be reset; otherwise, the interrupt is propagated after the **always** block.

Note that it is possible that an interrupt arrives during the execution of the **always** block; this interrupt is also propagated.

TTY The name of the tty associated with the shell, if any.

TTYIDLE <S>
The idle time of the tty associated with the shell in seconds or −1 if there is no such tty.

UID <S>
The real user ID of the shell process. If you have sufficient privileges, you may change the user ID of the shell by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different user ID by '**(UID=***uid***; command)**'

If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

USERNAME <S>
The username corresponding to the real user ID of the shell process. If you have sufficient privileges, you may change the username (and also the user ID and group ID) of the shell by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different username (and user ID and group ID) by '**(USERNAME=***username***; command)**'

VENDOR
The vendor, as determined at compile time.

zsh_eval_context <S> <Z> (**ZSH_EVAL_CONTEXT** <S>)
An array (colon−separated list) indicating the context of shell code that is being run. Each time a piece of shell code that is stored within the shell is executed a string is temporarily appended to the array to indicate the type of operation that is being performed. Read in order the array gives an indication of the stack of operations being performed with the most immediate context last.

Note that the variable does not give information on syntactic context such as pipelines or subshells. Use **$ZSH_SUBSHELL** to detect subshells.

The context is one of the following:

cmdarg
Code specified by the **−c** option to the command line that invoked the shell.

cmdsubst
Command substitution using the **`...`** or **$(...)** construct.

equalsubst
File substitution using the **=(...)** construct.

eval Code executed by the **eval** builtin.

evalautofunc
Code executed with the **KSH_AUTOLOAD** mechanism in order to define an autoloaded function.

fc Code from the shell history executed by the **−e** option to the **fc** builtin.

file Lines of code being read directly from a file, for example by the **source** builtin.

filecode
Lines of code being read from a **.zwc** file instead of directly from the source file.

globqual
Code executed by the **e** or **+** glob qualifier.

globsort
Code executed to order files by the **o** glob qualifier.

insubst File substitution using the **<(...)** construct.

loadautofunc
Code read directly from a file to define an autoloaded function.

outsubst
File substitution using the **>(...)** construct.

sched Code executed by the **sched** builtin.

shfunc A shell function.

stty Code passed to **stty** by the **STTY** environment variable. Normally this is passed directly to the system's **stty** command, so this value is unlikely to be seen in practice.

style Code executed as part of a style retrieved by the **zstyle** builtin from the **zsh/zutil** module.

toplevel
The highest execution level of a script or interactive shell.

trap Code executed as a trap defined by the **trap** builtin. Traps defined as functions have the context **shfunc**. As traps are asynchronous they may have a different hierarchy from other code.

zpty Code executed by the **zpty** builtin from the **zsh/zpty** module.

zregexparse−guard
Code executed as a guard by the **zregexparse** command from the **zsh/zutil** module.

zregexparse−action
Code executed as an action by the **zregexparse** command from the **zsh/zutil** module.

ZSH_ARGZERO
If zsh was invoked to run a script, this is the name of the script. Otherwise, it is the name used to invoke the current shell. This is the same as the value of **$0** when the **POSIX_ARGZERO** option is set, but is always available.

ZSH_EXECUTION_STRING
If the shell was started with the option **−c**, this contains the argument passed to the option. Otherwise it is not set.

ZSH_NAME
Expands to the basename of the command used to invoke this instance of zsh.

ZSH_PATCHLEVEL
The output of '**git describe −−tags −−long**' for the zsh repository used to build the shell. This is most useful in order to keep track of versions of the shell during development between releases; hence most users should not use it and should instead rely on **$ZSH_VERSION**.

zsh_scheduled_events
See the section 'The zsh/sched Module' in *zshmodules*(1).

ZSH_SCRIPT
If zsh was invoked to run a script, this is the name of the script, otherwise it is unset.

ZSH_SUBSHELL
Readonly integer. Initially zero, incremented each time the shell forks to create a subshell for executing code. Hence '**(print $ZSH_SUBSHELL)**' and '**print $(print $ZSH_SUBSHELL)**' output 1, while '**((print $ZSH_SUBSHELL))**' outputs 2.

ZSH_VERSION
The version number of the release of zsh.

PARAMETERS USED BY THE SHELL

The following parameters are used by the shell. Again, '<S>' indicates that the parameter is special and '<Z>' indicates that the parameter does not exist when the shell initializes in **sh** or **ksh** emulation mode.

In cases where there are two parameters with an upper− and lowercase form of the same name, such as **path** and **PATH**, the lowercase form is an array and the uppercase form is a scalar with the elements of the array joined together by colons. These are similar to tied parameters created via '**typeset −T**'. The normal use for the colon−separated form is for exporting to the environment, while the array form is easier to manipulate within the shell. Note that unsetting either of the pair will unset the other; they retain their special properties when recreated, and recreating one of the pair will recreate the other.

ARGV0
If exported, its value is used as the **argv[0]** of external commands. Usually used in constructs like '**ARGV0=emacs nethack**'.

BAUD The rate in bits per second at which data reaches the terminal. The line editor will use this value in order to compensate for a slow terminal by delaying updates to the display until necessary. If the parameter is unset or the value is zero the compensation mechanism is turned off. The parameter is not set by default.

This parameter may be profitably set in some circumstances, e.g. for slow modems dialing into a communications server, or on a slow wide area network. It should be set to the baud rate of the slowest part of the link for best performance.

cdpath <S> <Z> (**CDPATH** <S>)
An array (colon−separated list) of directories specifying the search path for the **cd** command.

COLUMNS <S>
The number of columns for this terminal session. Used for printing select lists and for the line editor.

CORRECT_IGNORE
If set, is treated as a pattern during spelling correction. Any potential correction that matches the pattern is ignored. For example, if the value is '**_***' then completion functions (which, by convention, have names beginning with '_') will never be offered as spelling corrections. The pattern does not apply to the correction of file names, as applied by the **CORRECT_ALL** option (so with the example just given files beginning with '_' in the current directory would still be completed).

CORRECT_IGNORE_FILE
If set, is treated as a pattern during spelling correction of file names. Any file name that matches the pattern is never offered as a correction. For example, if the value is '**.***' then dot file names will never be offered as spelling corrections. This is useful with the **CORRECT_ALL** option.

DIRSTACKSIZE
The maximum size of the directory stack, by default there is no limit. If the stack gets larger than this, it will be truncated automatically. This is useful with the **AUTO_PUSHD** option.

ENV If the **ENV** environment variable is set when zsh is invoked as **sh** or **ksh**, **$ENV** is sourced after the profile scripts. The value of **ENV** is subjected to parameter expansion, command substitution,

and arithmetic expansion before being interpreted as a pathname. Note that **ENV** is *not* used unless the shell is interactive and zsh is emulating **sh** or **ksh**.

FCEDIT
The default editor for the **fc** builtin. If **FCEDIT** is not set, the parameter **EDITOR** is used; if that is not set either, a builtin default, usually **vi**, is used.

fignore <S> <Z> (**FIGNORE** <S>)
An array (colon separated list) containing the suffixes of files to be ignored during filename completion. However, if completion only generates files with suffixes in this list, then these files are completed anyway.

fpath <S> <Z> (**FPATH** <S>)
An array (colon separated list) of directories specifying the search path for function definitions. This path is searched when a function with the **−u** attribute is referenced. If an executable file is found, then it is read and executed in the current environment.

histchars <S>
Three characters used by the shell's history and lexical analysis mechanism. The first character signals the start of a history expansion (default '**!**'). The second character signals the start of a quick history substitution (default '**^**'). The third character is the comment character (default '**#**').

The characters must be in the ASCII character set; any attempt to set **histchars** to characters with a locale−dependent meaning will be rejected with an error message.

HISTCHARS <S> <Z>
Same as **histchars**. (Deprecated.)

HISTFILE
The file to save the history in when an interactive shell exits. If unset, the history is not saved.

HISTORY_IGNORE
If set, is treated as a pattern at the time history files are written. Any potential history entry that matches the pattern is skipped. For example, if the value is '**fc ***' then commands that invoke the interactive history editor are never written to the history file.

Note that **HISTORY_IGNORE** defines a single pattern: to specify alternatives use the '(*first*|*second*|...)' syntax.

Compare the **HIST_NO_STORE** option or the **zshaddhistory** hook, either of which would prevent such commands from being added to the interactive history at all. If you wish to use **HISTORY_IGNORE** to stop history being added in the first place, you can define the following hook:

```
zshaddhistory() {
  emulate −L zsh
  ## uncomment if HISTORY_IGNORE
  ## should use EXTENDED_GLOB syntax
  # setopt extendedglob
  [[ $1 != ${~HISTORY_IGNORE} ]]
}
```

HISTSIZE <S>
The maximum number of events stored in the internal history list. If you use the **HIST_EXPIRE_DUPS_FIRST** option, setting this value larger than the **SAVEHIST** size will give you the difference as a cushion for saving duplicated history events.

If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

HOME <S>
The default argument for the **cd** command. This is not set automatically by the shell in **sh**, **ksh** or **csh** emulation, but it is typically present in the environment anyway, and if it becomes set it has its usual special behaviour.

IFS <S>
Internal field separators (by default space, tab, newline and NUL), that are used to separate words which result from command or parameter expansion and words read by the **read** builtin. Any characters from the set space, tab and newline that appear in the IFS are called *IFS white space*. One or more IFS white space characters or one non−IFS white space character together with any adjacent IFS white space character delimit a field. If an IFS white space character appears twice consecutively in the IFS, this character is treated as if it were not an IFS white space character.

If the parameter is unset, the default is used. Note this has a different effect from setting the parameter to an empty string.

KEYBOARD_HACK
This variable defines a character to be removed from the end of the command line before interpreting it (interactive shells only). It is intended to fix the problem with keys placed annoyingly close to return and replaces the **SUNKEYBOARDHACK** option which did this for backquotes only. Should the chosen character be one of singlequote, doublequote or backquote, there must also be an odd number of them on the command line for the last one to be removed.

For backward compatibility, if the **SUNKEYBOARDHACK** option is explicitly set, the value of **KEYBOARD_HACK** reverts to backquote. If the option is explicitly unset, this variable is set to empty.

KEYTIMEOUT
The time the shell waits, in hundredths of seconds, for another key to be pressed when reading bound multi−character sequences.

LANG <S>
This variable determines the locale category for any category not specifically selected via a variable starting with '**LC_**'.

LC_ALL <S>
This variable overrides the value of the '**LANG**' variable and the value of any of the other variables starting with '**LC_**'.

LC_COLLATE <S>
This variable determines the locale category for character collation information within ranges in glob brackets and for sorting.

LC_CTYPE <S>
This variable determines the locale category for character handling functions. If the **MULTIBYTE** option is in effect this variable or **LANG** should contain a value that reflects the character set in use, even if it is a single−byte character set, unless only the 7−bit subset (ASCII) is used. For example, if the character set is ISO−8859−1, a suitable value might be **en_US.iso88591** (certain Linux distributions) or **en_US.ISO8859−1** (MacOS).

LC_MESSAGES <S>
This variable determines the language in which messages should be written. Note that zsh does not use message catalogs.

LC_NUMERIC <S>
This variable affects the decimal point character and thousands separator character for the formatted input/output functions and string conversion functions. Note that zsh ignores this setting when parsing floating point mathematical expressions.

LC_TIME <S>
This variable determines the locale category for date and time formatting in prompt escape sequences.

LINES <S>
The number of lines for this terminal session. Used for printing select lists and for the line editor.

LISTMAX
In the line editor, the number of matches to list without asking first. If the value is negative, the list will be shown if it spans at most as many lines as given by the absolute value. If set to zero, the shell asks only if the top of the listing would scroll off the screen.

LOGCHECK
The interval in seconds between checks for login/logout activity using the **watch** parameter.

MAIL If this parameter is set and **mailpath** is not set, the shell looks for mail in the specified file.

MAILCHECK
The interval in seconds between checks for new mail.

mailpath <S> <Z> (**MAILPATH** <S>)
An array (colon−separated list) of filenames to check for new mail. Each filename can be followed by a '**?**' and a message that will be printed. The message will undergo parameter expansion, command substitution and arithmetic expansion with the variable **$_** defined as the name of the file that has changed. The default message is '**You have new mail**'. If an element is a directory instead of a file the shell will recursively check every file in every subdirectory of the element.

manpath <S> <Z> (**MANPATH** <S> <Z>)
An array (colon−separated list) whose value is not used by the shell. The **manpath** array can be useful, however, since setting it also sets **MANPATH**, and vice versa.

match
mbegin
mend Arrays set by the shell when the **b** globbing flag is used in pattern matches. See the subsection *Globbing flags* in the documentation for *Filename Generation* in *zshexpn*(1).

MATCH
MBEGIN
MEND Set by the shell when the **m** globbing flag is used in pattern matches. See the subsection *Globbing flags* in the documentation for *Filename Generation* in *zshexpn*(1).

module_path <S> <Z> (**MODULE_PATH** <S>)
An array (colon−separated list) of directories that **zmodload** searches for dynamically loadable modules. This is initialized to a standard pathname, usually '**/usr/local/lib/zsh/$ZSH_VERSION**'. (The '**/usr/local/lib**' part varies from installation to installation.) For security reasons, any value set in the environment when the shell is started will be ignored.

These parameters only exist if the installation supports dynamic module loading.

NULLCMD <S>
The command name to assume if a redirection is specified with no command. Defaults to **cat**. For **sh/ksh** behavior, change this to **:**. For **csh**−like behavior, unset this parameter; the shell will print an error message if null commands are entered.

path <S> <Z> (**PATH** <S>)
An array (colon−separated list) of directories to search for commands. When this parameter is set, each directory is scanned and all files found are put in a hash table.

POSTEDIT <S>
This string is output whenever the line editor exits. It usually contains termcap strings to reset the terminal.

PROMPT <S> <Z>
PROMPT2 <S> <Z>
PROMPT3 <S> <Z>
PROMPT4 <S> <Z>
Same as **PS1**, **PS2**, **PS3** and **PS4**, respectively.

prompt <S> <Z>
Same as **PS1**.

PROMPT_EOL_MARK
When the **PROMPT_CR** and **PROMPT_SP** options are set, the **PROMPT_EOL_MARK** parameter can be used to customize how the end of partial lines are shown. This parameter undergoes prompt expansion, with the **PROMPT_PERCENT** option set. If not set, the default behavior is equivalent to the value '**%B%S%#%s%b**'.

PS1 <S>
The primary prompt string, printed before a command is read. It undergoes a special form of expansion before being displayed; see EXPANSION OF PROMPT SEQUENCES in *zshmisc*(1). The default is '**%m%#** '.

PS2 <S>
The secondary prompt, printed when the shell needs more information to complete a command. It is expanded in the same way as **PS1**. The default is '**%_>** ', which displays any shell constructs or quotation marks which are currently being processed.

PS3 <S>
Selection prompt used within a **select** loop. It is expanded in the same way as **PS1**. The default is '**?#** '.

PS4 <S>
The execution trace prompt. Default is '**+%N:%i>** ', which displays the name of the current shell structure and the line number within it. In sh or ksh emulation, the default is '**+** '.

psvar <S> <Z> (**PSVAR** <S>)
An array (colon−separated list) whose elements can be used in **PROMPT** strings. Setting **psvar** also sets **PSVAR**, and vice versa.

READNULLCMD <S>
The command name to assume if a single input redirection is specified with no command. Defaults to **more**.

REPORTMEMORY
If nonnegative, commands whose maximum resident set size (roughly speaking, main memory usage) in kilobytes is greater than this value have timing statistics reported. The format used to output statistics is the value of the **TIMEFMT** parameter, which is the same as for the **REPORTTIME** variable and the **time** builtin; note that by default this does not output memory usage. Appending **" max RSS %M"** to the value of **TIMEFMT** causes it to output the value that triggered the report. If **REPORTTIME** is also in use, at most a single report is printed for both triggers. This feature requires the **getrusage()** system call, commonly supported by modern Unix−like systems.

REPORTTIME
If nonnegative, commands whose combined user and system execution times (measured in seconds) are greater than this value have timing statistics printed for them. Output is suppressed for commands executed within the line editor, including completion; commands explicitly marked with the **time** keyword still cause the summary to be printed in this case.

REPLY
This parameter is reserved by convention to pass string values between shell scripts and shell builtins in situations where a function call or redirection are impossible or undesirable. The **read** builtin and the **select** complex command may set **REPLY**, and filename generation both sets and examines its value when evaluating certain expressions. Some modules also employ **REPLY** for similar purposes.

reply As **REPLY**, but for array values rather than strings.

RPROMPT <S>
RPS1 <S>
: This prompt is displayed on the right−hand side of the screen when the primary prompt is being displayed on the left. This does not work if the **SINGLE_LINE_ZLE** option is set. It is expanded in the same way as **PS1**.

RPROMPT2 <S>
RPS2 <S>
: This prompt is displayed on the right−hand side of the screen when the secondary prompt is being displayed on the left. This does not work if the **SINGLE_LINE_ZLE** option is set. It is expanded in the same way as **PS2**.

SAVEHIST
: The maximum number of history events to save in the history file.

  If this is made local, it is not implicitly set to 0, but may be explicitly set locally.

SPROMPT <S>
: The prompt used for spelling correction. The sequence '**%R**' expands to the string which presumably needs spelling correction, and '**%r**' expands to the proposed correction. All other prompt escapes are also allowed.

  The actions available at the prompt are **[nyae]**:
  n ('no') (default)
  : Discard the correction and run the command.
  y ('yes')
  : Make the correction and run the command.
  a ('abort')
  : Discard the entire command line without running it.
  e ('edit')
  : Resume editing the command line.

STTY If this parameter is set in a command's environment, the shell runs the **stty** command with the value of this parameter as arguments in order to set up the terminal before executing the command. The modes apply only to the command, and are reset when it finishes or is suspended. If the command is suspended and continued later with the **fg** or **wait** builtins it will see the modes specified by **STTY**, as if it were not suspended. This (intentionally) does not apply if the command is continued via '**kill −CONT**'. **STTY** is ignored if the command is run in the background, or if it is in the environment of the shell but not explicitly assigned to in the input line. This avoids running stty at every external command by accidentally exporting it. Also note that **STTY** should not be used for window size specifications; these will not be local to the command.

TERM <S>
: The type of terminal in use. This is used when looking up termcap sequences. An assignment to **TERM** causes zsh to re−initialize the terminal, even if the value does not change (e.g., '**TERM=$TERM**'). It is necessary to make such an assignment upon any change to the terminal definition database or terminal type in order for the new settings to take effect.

TERMINFO <S>
: A reference to your terminfo database, used by the 'terminfo' library when the system has it; see *terminfo*(5). If set, this causes the shell to reinitialise the terminal, making the workaround '**TERM=$TERM**' unnecessary.

TERMINFO_DIRS <S>
: A colon−seprarated list of terminfo databases, used by the 'terminfo' library when the system has it; see *terminfo*(5). This variable is only used by certain terminal libraries, in particular ncurses; see *terminfo*(5) to check support on your system. If set, this causes the shell to reinitialise the terminal, making the workaround '**TERM=$TERM**' unnecessary. Note that unlike other colon−separated arrays this is not tied to a zsh array.

TIMEFMT

The format of process time reports with the **time** keyword. The default is '**%J %U user %S system %P cpu %*E total**'. Recognizes the following escape sequences, although not all may be available on all systems, and some that are available may not be useful:

%%	A '**%**'.
%U	CPU seconds spent in user mode.
%S	CPU seconds spent in kernel mode.
%E	Elapsed time in seconds.
%P	The CPU percentage, computed as 100*(**%U**+**%S**)/**%E**.
%W	Number of times the process was swapped.
%X	The average amount in (shared) text space used in kilobytes.
%D	The average amount in (unshared) data/stack space used in kilobytes.
%K	The total space used (**%X**+**%D**) in kilobytes.
%M	The maximum memory the process had in use at any time in kilobytes.
%F	The number of major page faults (page needed to be brought from disk).
%R	The number of minor page faults.
%I	The number of input operations.
%O	The number of output operations.
%r	The number of socket messages received.
%s	The number of socket messages sent.
%k	The number of signals received.
%w	Number of voluntary context switches (waits).
%c	Number of involuntary context switches.
%J	The name of this job.

A star may be inserted between the percent sign and flags printing time (e.g., '**%*E**'); this causes the time to be printed in '*hh***:***mm***:***ss***.***ttt*' format (hours and minutes are only printed if they are not zero). Alternatively, '**m**' or '**u**' may be used (e.g., '**%mE**') to produce time output in milliseconds or microseconds, respectively.

TMOUT

If this parameter is nonzero, the shell will receive an **ALRM** signal if a command is not entered within the specified number of seconds after issuing a prompt. If there is a trap on **SIGALRM**, it will be executed and a new alarm is scheduled using the value of the **TMOUT** parameter after executing the trap. If no trap is set, and the idle time of the terminal is not less than the value of the **TMOUT** parameter, zsh terminates. Otherwise a new alarm is scheduled to **TMOUT** seconds after the last keypress.

TMPPREFIX

A pathname prefix which the shell will use for all temporary files. Note that this should include an initial part for the file name as well as any directory names. The default is '**/tmp/zsh**'.

TMPSUFFIX

A filename suffix which the shell will use for temporary files created by process substitutions (e.g., '**=(***list***)***'). Note that the value should include a leading dot '***.***' if intended to be interpreted as a file extension. The default is not to append any suffix, thus this parameter should be assigned only when needed and then unset again.*

watch <*S*> <*Z*> *(***WATCH** <*S*>*)*

An array (colon−separated list) of login/logout events to report.

If it contains the single word '**all**', then all login/logout events are reported. If it contains the single word '**notme**', then all events are reported as with '**all**' except **$USERNAME**.

An entry in this list may consist of a username, an '@' followed by a remote hostname, and a '**%**' followed by a line (tty). Any of these may be a pattern (be sure to quote this during the assignment to **watch** so that it does not immediately perform file generation); the setting of the **EXTENDED_GLOB** option is respected. Any or all of these components may be present in an

entry; if a login/logout event matches all of them, it is reported.

For example, with the **EXTENDED_GLOB** option set, the following:

watch=('^(pws|barts)')

causes reports for activity associated with any user other than **pws** or **barts**.

WATCHFMT

The format of login/logout reports if the **watch** parameter is set. Default is '**%n has %a %l from %m**'. Recognizes the following escape sequences:

%n The name of the user that logged in/out.

%a The observed action, i.e. "logged on" or "logged off".

%l The line (tty) the user is logged in on.

%M The full hostname of the remote host.

%m The hostname up to the first '**.**'. If only the IP address is available or the utmp field contains the name of an X−windows display, the whole name is printed.

NOTE: The '**%m**' and '**%M**' escapes will work only if there is a host name field in the utmp on your machine. Otherwise they are treated as ordinary strings.

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

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

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

%t
%@ The time, in 12−hour, am/pm format.

%T The time, in 24−hour format.

%w The date in '*day−dd*' format.

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

%D The date in '*yy−mm−dd*' format.

%D{*string***}**
The date formatted as *string* using the **strftime** function, with zsh extensions as described by EXPANSION OF PROMPT SEQUENCES in *zshmisc*(1).

%(*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. Both the separator and the right parenthesis may be escaped with a backslash. Ternary expressions may be nested.

The test character *x* may be any one of '**l**', '**n**', '**m**' or '**M**', which indicate a 'true' result if the corresponding escape sequence would return a non−empty value; or it may be '**a**', which indicates a 'true' result if the watched user has logged in, or 'false' if he has logged out. Other characters evaluate to neither true nor false; the entire expression is omitted in this case.

If the result is 'true', then the *true−text* is formatted according to the rules above and printed, and the *false−text* is skipped. If 'false', the *true−text* is skipped and the *false−text* is formatted and printed. Either or both of the branches may be empty, but both separators must be present in any case.

WORDCHARS <S>
A list of non−alphanumeric characters considered part of a word by the line editor.

ZBEEP
If set, this gives a string of characters, which can use all the same codes as the **bindkey** command as described in the zsh/zle module entry in *zshmodules*(1), that will be output to the terminal instead of beeping. This may have a visible instead of an audible effect; for example, the string '**\e[?5h\e[?5l**' on a vt100 or xterm will have the effect of flashing reverse video on and off (if you usually use reverse video, you should use the string '**\e[?5l\e[?5h**' instead). This takes precedence over the **NOBEEP** option.

ZDOTDIR
The directory to search for shell startup files (.zshrc, etc), if not **$HOME**.

zle_bracketed_paste
Many terminal emulators have a feature that allows applications to identify when text is pasted into the terminal rather than being typed normally. For ZLE, this means that special characters such as tabs and newlines can be inserted instead of invoking editor commands. Furthermore, pasted text forms a single undo event and if the region is active, pasted text will replace the region.

This two−element array contains the terminal escape sequences for enabling and disabling the feature. These escape sequences are used to enable bracketed paste when ZLE is active and disable it at other times. Unsetting the parameter has the effect of ensuring that bracketed paste remains disabled.

zle_highlight
An array describing contexts in which ZLE should highlight the input text. See *Character Highlighting* in *zshzle*(1).

ZLE_LINE_ABORTED
This parameter is set by the line editor when an error occurs. It contains the line that was being edited at the point of the error. '**print −zr −− $ZLE_LINE_ABORTED**' can be used to recover the line. Only the most recent line of this kind is remembered.

ZLE_REMOVE_SUFFIX_CHARS
ZLE_SPACE_SUFFIX_CHARS
These parameters are used by the line editor. In certain circumstances suffixes (typically space or slash) added by the completion system will be removed automatically, either because the next editing command was not an insertable character, or because the character was marked as requiring the suffix to be removed.

These variables can contain the sets of characters that will cause the suffix to be removed. If **ZLE_REMOVE_SUFFIX_CHARS** is set, those characters will cause the suffix to be removed; if **ZLE_SPACE_SUFFIX_CHARS** is set, those characters will cause the suffix to be removed and replaced by a space.

If **ZLE_REMOVE_SUFFIX_CHARS** is not set, the default behaviour is equivalent to:

ZLE_REMOVE_SUFFIX_CHARS=$' \t\n;&|'

If **ZLE_REMOVE_SUFFIX_CHARS** is set but is empty, no characters have this behaviour. **ZLE_SPACE_SUFFIX_CHARS** takes precedence, so that the following:

ZLE_SPACE_SUFFIX_CHARS=$'&|'

causes the characters '**&**' and '**|**' to remove the suffix but to replace it with a space.

To illustrate the difference, suppose that the option **AUTO_REMOVE_SLASH** is in effect and the directory **DIR** has just been completed, with an appended **/**, following which the user types '**&**'. The default result is '**DIR&**'. With **ZLE_REMOVE_SUFFIX_CHARS** set but without including '**&**' the result is '**DIR/&**'. With **ZLE_SPACE_SUFFIX_CHARS** set to include '**&**' the result is '**DIR &**'.

Note that certain completions may provide their own suffix removal or replacement behaviour which overrides the values described here. See the completion system documentation in *zshcompsys*(1).

ZLE_RPROMPT_INDENT <S>

If set, used to give the indentation between the right hand side of the right prompt in the line editor as given by **RPS1** or **RPROMPT** and the right hand side of the screen. If not set, the value 1 is used.

Typically this will be used to set the value to 0 so that the prompt appears flush with the right hand side of the screen. This is not the default as many terminals do not handle this correctly, in particular when the prompt appears at the extreme bottom right of the screen. Recent virtual terminals are more likely to handle this case correctly. Some experimentation is necessary.