

NAME

ksh, rksh – KornShell, a standard/restricted command and programming language

SYNOPSIS

```
ksh [ ±abcefhiklmnpstuvxBCDEGH ] [ ±o option ] ... [ - ] [ arg ... ]
rksh [ ±abcefhiklmnpstuvxBCDEGH ] [ ±o option ] ... [ - ] [ arg ... ]
```

DESCRIPTION

Ksh is a command and programming language that executes commands read from a terminal or a file. *Rksh* is a restricted version of the command interpreter *ksh*; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See *Invocation* below for the meaning of arguments to the shell.

Definitions.

A *metacharacter* is one of the following characters:

```
; & ( ) | < > new-line space tab
```

A *blank* is a **tab** or a **space**. An *identifier* is a sequence of letters, digits, or underscores starting with a letter or underscore. Identifiers are used as components of *variable* names. A *vname* is a sequence of one or more identifiers separated by a **.** and optionally preceded by a **.**. Vnames are used as function and variable names. A *word* is a sequence of *characters* from the character set defined by the current locale, excluding non-quoted *metacharacters*.

A *command* is a sequence of characters in the syntax of the shell language. The shell reads each command and carries out the desired action either directly or by invoking separate utilities. A built-in command is a command that is carried out by the shell itself without creating a separate process. Some commands are built in purely for convenience and are not documented here. Built-ins that cause side effects in the shell environment and built-ins that are found before performing a path search (see *Execution* below) are documented here. For historical reasons, some of these built-ins behave differently than other built-ins and are called *special built-ins*.

Commands.

A *simple-command* is a list of variable assignments (see *Variable Assignments* below) or a sequence of *blank* separated words which may be preceded by a list of variable assignments (see *Environment* below). The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see *exec(2)*). The *value* of a *simple-command* is its exit status; 0-255 if it terminates normally; 256+*signum* if it terminates abnormally (the name of the signal corresponding to the exit status can be obtained via the **-I** option of the **kill** built-in utility).

A *pipeline* is a sequence of one or more *commands* separated by **|**. The standard output of each command but the last is connected by a *socketpair(2)* or (if the **posix** shell option is on) by a *pipe(2)* to the standard input of the next command. Each command except the last is run asynchronously in a subshell (see *Subshells* below). If the **monitor** or **pipefail** option is on, or the pipeline is preceded by the *reserved word* **time**, then the shell waits for all component commands in the pipeline to terminate; otherwise, the shell only waits for the last component command. The exit status of a pipeline is the exit status of its last component command, unless the **pipefail** option is enabled. Each pipeline can be preceded by the *reserved word* **!** which causes the exit status of the pipeline to become 0 if the exit status of the last command is non-zero, and 1 if the exit status of the last command is 0.

A *list* is a sequence of one or more pipelines separated by **;**, **&**, **|&**, **&&**, or **||**, and optionally terminated by **;**, **&**, or **|&**. Of these five symbols, **;**, **&**, and **|&** have equal precedence, which is lower than that of **&&** and **||**. The symbols **&&** and **||** also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding pipeline; an ampersand (**&**) causes asynchronous execution of the preceding pipeline (i.e., the shell does *not* wait for that pipeline to finish). The symbol **|&** causes asynchronous execution of the preceding pipeline with a two-way pipe established to the parent shell; the standard input and output of the spawned pipeline can be written to and read from by the parent shell by applying the redirection operators **<&** and **>&** with arg **p** to commands and by using **-p** option of the built-in commands **read** and **print** described later. The symbol **&&** (**||**) causes the *list* following it to be executed only if the preceding pipeline

returns a zero (non-zero) value. One or more new-lines may appear in a *list* instead of a semicolon, to delimit a command. The first *item* of the first *pipeline* of a *list* that is a simple command not beginning with a redirection, and not occurring within a **while**, **until**, or **if** *list*, can be preceded by a semicolon. This semicolon is ignored unless the **showme** option is enabled as described with the **set** built-in below.

A *command* is either a *simple-command* or a *compound-command*, which is one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command.

for *vname* [**in** *word* ...] ;**do** *list* ;**done**

Each time a **for** command is executed, *vname* is set to the next *word* taken from the **in** *word* list. If **in** *word* ... is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set starting from **1** (see *Parameter Expansion* below). Execution ends when there are no more words in the list.

for (([*expr1*] ; [*expr2*] ; [*expr3*])) ;**do** *list* ;**done**

The arithmetic expression *expr1* is evaluated first (see *Arithmetic Evaluation* below). 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.

select *vname* [**in** *word* ...] ;**do** *list* ;**done**

A **select** command prints on standard error (file descriptor 2) the set of *words*, each preceded by a number. If **in** *word* ... is omitted, then the positional parameters starting from **1** are used instead (see *Parameter Expansion* below). The **PS3** prompt is printed and a line is read from the standard input. If this line consists of the number of one of the listed *words*, then the value of the variable *vname* 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 variable *vname* is set to the empty string. The contents of the line read from standard input is saved in the variable **REPLY**. The *list* is executed for each selection until a **break** or *end-of-file* is encountered. If the **REPLY** variable is set to the empty string by the execution of *list*, then the selection list is printed before displaying the **PS3** prompt for the next selection.

case *word* **in** [([*pattern* [| *pattern*] ...) *list* ;;] ... **esac**

A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for pathname expansion (see *Pathname Expansion* below). The **;;** operator causes execution of **case** to terminate. If **;&** is used in place of **;;** the next subsequent list, if any, is executed.

if *list* ;**then** *list* [;**elif** *list* ;**then** *list*] ... [;**else** *list*] ;**fi**

The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing each successive **elif** *list*, the **else** *list* is executed. If the **if** *list* has non-zero exit status and there is no **else** *list*, then the **if** command returns a zero exit status.

while *list* ;**do** *list* ;**done**

until *list* ;**do** *list* ;**done**

A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

while *inputredirection* ;**do** *list* ;**done**

Filescan loop. This is defined by a lone input redirection following **while** (see *Input/Output* below). It is faster than using the **read** built-in command in a regular **while** loop. The shell reads lines from the file or stream opened by *inputredirection* until the end is reached or the loop is broken. For each line read, the command *list* is executed with the line's contents assigned to the **REPLY** variable and the line's fields split into the positional parameters (see *Field Splitting* and *Positional Parameters* below). Within the *list*, standard input is redirected to **/dev/null**. If the **posix**

compatibility shell option is on, this loop type is disabled and *inputredirection* is processed like a lone redirection in any other context.

((expression))

The *expression* is evaluated using the rules for arithmetic evaluation described below. If the value of the arithmetic expression is non-zero, the exit status is 0, otherwise the exit status is 1.

(list)

Execute *list* in a subshell (see *Subshells* below). Note, that if two adjacent open parentheses are needed for nesting, a space must be inserted to avoid evaluation as an arithmetic command as described above.

{ list;}

list is simply executed. Note that unlike the metacharacters (and), { and } are *reserved words* and must occur at the beginning of a line or after a ; in order to be recognized.

[[expression]]

Evaluates *expression* and returns a zero exit status when *expression* is true. See *Conditional Expressions* below, for a description of *expression*.

function varname { list ;} [redirection ...]

varname () *compound-command* [*redirection* ...]

Define a function which is referenced by *varname*. A function whose *varname* contains a . is called a discipline function and the portion of the *varname* preceding the last . must refer to an existing variable. The body of the function is the *list* of commands between { and }. A function defined with the **function** *varname* syntax can also be used as an argument to the . special built-in command to get the equivalent behavior as if the *varname*() syntax were used to define it. (See *Functions* below.)

namespace identifier { list ;}

Defines or uses the name space *identifier* and runs the commands in *list* in this name space. (See *Name Spaces* below.)

time [pipeline]

If *pipeline* is omitted the user and system time for the current shell and completed child processes is printed on standard error. Otherwise, *pipeline* is executed and the elapsed time as well as the user and system time are printed on standard error. The **TIMEFORMAT** variable may be set to a format string that specifies how the timing information should be displayed. See **Shell Variables** below for a description of the **TIMEFORMAT** variable.

The following reserved words are recognized as reserved only when they are the first word of a command and are not quoted:

if then else elif fi case esac for while until do done { } function select time [[]] !

Variable Assignments.

One or more variable assignments can start a simple command or can be arguments to the **typeset**, **enum**, **export**, or **readonly** special built-in commands as well as to other declaration commands created as types. The syntax for an *assignment* is of the form:

varname=*word*

varname[*word*]=*word*

No space is permitted between *varname* and the = or between = and *word*.

varname=(*assign_list*)

No space is permitted between *varname* and the =. The variable *varname* is unset before the assignment. An *assign_list* can be one of the following:

word ...

Indexed array assignment.

[word]=word . . .

Associative array assignment. If preceded by **typeset -a** this will create an indexed array instead.

assignment . . .

Compound variable assignment. This creates a compound variable *varname* with subvariables of the form *varname.name*, where *name* is the name portion of *assignment*. The value of *varname* will contain all the assignment elements. Additional assignments made to subvariables of *varname* will also be displayed as part of the value of *varname*. If no *assignments* are specified, *varname* will be a compound variable allowing subsequence child elements to be defined.

typeset [options] assignment . . .

Nested variable assignment. Multiple assignments can be specified by separating each of them with a **;**. The previous value is unset before the assignment. Other declaration commands such as **readonly**, **enum**, and other declaration commands can be used in place of **typeset**.

.filename

Include the assignment commands contained in *filename*.

In addition, a **+=** can be used in place of the **=** to signify adding to or appending to the previous value. When **+=** is applied to an arithmetic type, *word* is evaluated as an arithmetic expression and added to the current value. When applied to a string variable, the value defined by *word* is appended to the value. For compound assignments, the previous value is not unset and the new values are appended to the current ones provided that the types are compatible.

The right hand side of a variable assignment undergoes all the expansion listed below except word splitting, brace expansion, and pathname expansion. When the left hand side is an assignment to a compound variable and the right hand side is the name of a compound variable, the compound variable on the right will be copied or appended to the compound variable on the left.

Comments.

A word beginning with **#** causes that word and all the following characters up to a new-line to be ignored.

Aliasing.

The first word of each command is replaced by the text of an **alias** if an **alias** for this word has been defined. An **alias** name consists of any number of characters excluding metacharacters, quoting characters, file expansion characters, parameter expansion and command substitution characters, the characters **/** and **=**. The replacement string can contain any valid shell script including the metacharacters listed above. The first word of each command in the replaced text, other than any that are in the process of being replaced, will be tested for aliases. If the last character of the alias value is a *blank* then the word following the alias will also be checked for alias substitution. Aliases can be used to redefine built-in commands but cannot be used to redefine the reserved words listed above. Aliases can be created and listed with the **alias** command and can be removed with the **unalias** command.

Aliasing is performed when scripts are read, not while they are executed. Therefore, for an alias to take effect, the **alias** definition command has to be executed before the command which references the alias is read.

The following aliases are automatically preset when the shell is invoked as an interactive shell. Preset aliases can be unset or redefined.

```
history='hist -l'
r='hist -s'
```

Tilde Expansion.

After alias substitution is performed, each word is checked to see if it begins with an unquoted **~**. For tilde expansion, *word* also refers to the *word* portion of parameter expansion (see *Parameter Expansion* below). If a *word* is preceded by a tilde, then it is checked up to a **/** to see if it matches a user name in the password database (see *getpwnam(3)*). If a match is found, the **~** and the matched login name are replaced by the login directory of the matched user. If no match is found, the original text is left unchanged. A **~** by itself, or in front of a **/**, is replaced by **\$HOME**, unless the **HOME** variable is unset, in which case the current user's

home directory as configured in the operating system is used. A `~` followed by a `+` or `-` is replaced by `$PWD` or `$OLDPWD` respectively.

In addition, when expanding a variable assignment (see *Variable Assignments* above), tilde expansion is attempted when the value of the assignment begins with a `~`, and when a `~` appears after a `:`. A `:` also terminates a user name following a `~`.

The tilde expansion mechanism may be extended or modified by defining one of the discipline functions `.sh.tilde.set` or `.sh.tilde.get` (see *Functions* and *Discipline Functions* below). If either exists, then upon encountering a tilde word to expand, that function is called with the tilde word assigned to either `.sh.value` (for the `.sh.tilde.set` function) or `.sh.tilde` (for the `.sh.tilde.get` function). Performing tilde expansion within a discipline function will not recursively call that function, but default tilde expansion remains active, so literal tildes should still be quoted where required. Either function may assign a replacement string to `.sh.value`. If this value is non-empty and does not start with a `~`, it replaces the default tilde expansion when the function terminates. Otherwise, the tilde expansion is left unchanged.

Subshells.

A *subshell* is a separate execution environment that is a complete duplicate of the current shell environment, except for two things: all traps are reset to default except those for signals that are being ignored, and subshells cannot be interactive (i.e., they have no command prompt). Changes made within a subshell do not affect the parent environment and are lost when the subshell exits.

Particular care should be taken not to confuse a subshell with a newly invoked shell that is merely a child process of the current shell, and which (unlike a subshell) starts from scratch in terms of variables and functions and may be interactive. Beware of shell tutorials on the Internet that confuse these.

Subshells cannot be created or invoked using any command. Instead, the following are automatically run in a subshell:

- any command or group of commands enclosed in parentheses;
- command substitutions of the first and third form (see *Command Substitution* below);
- process substitutions (see *Process Substitution* below);
- all elements of a pipeline except the last;
- any command executed asynchronously (i.e., in a background process).

Creating processes is expensive, so as a performance optimization, a subshell of a non-interactive shell may share the process of its parent environment. Such a subshell is known as a virtual subshell. Subshells are virtual unless or until something (such as asynchronous execution, or an attempt to set a process limit using the `ulimit` built-in command, or other implementation- or system-defined requirements) makes it necessary to *fork(2)* it into a separate process. Barring any bugs in the shell, virtual subshells should be indistinguishable from real subshells except by their execution speed and their process ID. See the description of the `.sh.pid` variable below for more information.

Command Substitution.

The standard output from a command list enclosed in parentheses preceded by a dollar sign (`$(list)`), or in a brace group preceded by a dollar sign (`${list;}`), or in a pair of grave accents (`` ``) may be used as part or all of a word; trailing new-lines are removed. In the second case, the `{` and `}` are treated as a reserved words so that `{` must be followed by a *blank* and `}` must appear at the beginning of the line or follow a `;`. In the third (obsolete) form, the string between the quotes is processed for special quoting characters before the command is executed (see *Quoting* below). The command substitution `$(cat file)` can be replaced by the equivalent but faster `$(<file)`. The command substitution `$(n<#)` will expand to the current byte offset for file descriptor *n*. Except for the second form, the command list is run in a subshell so that no side effects are possible. For the second form, the final `}` will be recognized as a reserved word after any token.

Arithmetic Expansion.

An arithmetic expression enclosed in double parentheses preceded by a dollar sign (`$(())`) is replaced by the value of the arithmetic expression within the double parentheses.

Process Substitution.

Each command argument of the form `<(list)` or `>(list)` will run process *list* asynchronously connected to some file in `/dev/fd` if this directory exists, or else a fifo in a temporary directory. The name of this file will become the argument to the command. If the form with `>` is selected then writing on this file will provide input for *list*. If `<` is used, then the file passed as an argument will contain the output of the *list* process. For example,

```
paste <(cut -f1 file1) <(cut -f3 file2) | tee >(process1) >(process2)
```

cuts fields 1 and 3 from the files *file1* and *file2* respectively, *pastes* the results together, and sends it to the processes *process1* and *process2*, as well as putting it onto the standard output. Note that the file, which is passed as an argument to the command, is a UNIX *pipe(2)* so programs that expect to *lseek(2)* on the file will not work.

Process substitution of the form `<(list)` can also be used with the `<` redirection operator which causes the output of *list* to be standard input or the input for whatever file descriptor is specified.

Parameter Expansion.

A *parameter* is a *variable*, one or more digits, or any of the characters `*`, `@`, `#`, `?`, `-`, `$`, and `!`. A *variable* is denoted by a *vname*. To create a variable whose *vname* contains a `.`, a variable whose *vname* consists of everything before the last `.` must already exist. A *variable* has a *value* and zero or more *attributes*. *Variables* can be assigned *values* and *attributes* by using the **typeset** special built-in command. The attributes supported by the shell are described later with the **typeset** special built-in command.

The shell supports both indexed and associative arrays. An element of an array variable is referenced by a *subscript*. A *subscript* for an indexed array is denoted by an *arithmetic expression* (see *Arithmetic Evaluation* below) between a `[` and a `]`. To assign values to an indexed array, use `vname=(value ...)` or `set -A vname value ...`. The value of all non-negative subscripts must be in the range of 0 through 4,194,303. A negative subscript is treated as an offset from the maximum current index +1 so that `-1` refers to the last element. Indexed arrays can be declared with the `-a` option to **typeset**. Indexed arrays need not be declared. Any reference to a variable with a valid subscript is legal and an array will be created if necessary.

An associative array is created with the `-A` option to **typeset**. A *subscript* for an associative array is denoted by a string enclosed between `[` and `]`.

Referencing any array without a subscript is equivalent to referencing the array with subscript 0.

The *value* of a *variable* may be assigned by writing:

```
vname=value [ vname=value ] ...
```

or

```
vname[subscript]=value [ vname[subscript]=value ] ...
```

Note that no space is allowed before or after the `=`.

Attributes assigned by the *typeset* special built-in command apply to all elements of the array. An array element can be a simple variable, a compound variable or an array variable. An element of an indexed array can be either an indexed array or an associative array. An element of an associative array can also be either. To refer to an array element that is part of an array element, concatenate the subscript in brackets. For example, to refer to the *foobar* element of an associative array that is defined as the third element of the indexed array, use `${vname[3][foobar]}`

A *nameref* is a variable that is a reference to another variable. A *nameref* is created with the `-n` attribute of **typeset** or with the equivalent **nameref** command. The value of the variable at the time of that command becomes the variable that will be referenced whenever the *nameref* variable is used. The name of a *nameref* cannot contain a `.`. When a variable or function name contains a `.`, and the portion of the name up to the first `.` matches the name of a *nameref*, the variable referred to is obtained by replacing the *nameref* portion with the name of the variable referenced by the *nameref*. If a *nameref* is used as the index of a **for** loop, a name reference is established for each item in the list.

A *nameref* provides a convenient way to refer to the variable inside a function whose name is passed as an

argument to a function. For example, if the name of a variable is passed as the first argument to a function, the command **typeset -n var=\$1** (a.k.a. **nameref var=\$1**) inside the function causes references and assignments to **var** to be references and assignments to the variable whose name has been passed to the function. Note that, for this to work, the positional parameter must be assigned directly to the nameref as part of the declaration command, as in the example above; only that idiom can allow one function to access a local variable of another. For instance, **typeset -n var; var=\$1** won't cross that barrier, nor will **typeset foo=\$1; typeset -n var=foo**.

If any of the floating point attributes, **-E**, **-F**, or **-X**, or the integer attribute, **-i**, is set for *vname*, then the *value* is subject to arithmetic evaluation as described below.

Positional parameters, parameters denoted by a number, may be assigned values with the **set** special built-in command. Parameter **\$0** is set from argument zero when the shell is invoked.

The character **\$** is used to introduce substitutable *parameters*.

\${parameter}

The shell reads all the characters from **\${** to the matching **}** as part of the same word even if it contains braces or metacharacters. The value, if any, of the parameter is substituted. The braces are required when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name, when the variable name contains a **.**. The braces are also required when a variable is subscripted unless it is part of an Arithmetic Expression or a Conditional Expression. If *parameter* is one or more digits then it is a positional parameter. A positional parameter of more than one digit must be enclosed in braces. If *parameter* is ***** or **@**, then all the positional parameters, starting with **\$1**, are substituted (separated by a field separator character). If an array *vname* with last subscript ***** **@**, or for indexed arrays of the form *sub1* .. *sub2*. is used, then the value for each of the elements between *sub1* and *sub2* inclusive (or all elements for ***** and **@**) is substituted, separated by the first character of the value of **IFS**.

\${#parameter}

If *parameter* is ***** or **@**, the number of positional parameters is substituted. Otherwise, the length of the value of the *parameter* is substituted.

\${#vname[*]}

\${#vname[@]}

The number of elements in the array *vname* is substituted.

\${@vname}

Expands to the type name (see *Type Variables* below) or attributes of the variable referred to by *vname*.

\${!vname}

Expands to the name of the variable referred to by *vname*. This will be *vname* except when *vname* is a name reference.

\${!vname[subscript]}

Expands to name of the subscript unless *subscript* is *****, **@**. or of the form *sub1* .. *sub2*. When *subscript* is *****, the list of array subscripts for *vname* is generated. For a variable that is not an array, the value is *0* if the variable is set. Otherwise it is the empty string. When *subscript* is **@**, same as above, except that when used in double quotes, each array subscript yields a separate argument. When *subscript* is of the form *sub1* .. *sub2* it expands to the list of subscripts between *sub1* and *sub2* inclusive using the same quoting rules as **@**.

\${!prefix@}

\${!prefix*}

These both expand to the names of the variables whose names begin with *prefix*. The expansions otherwise work like **\$@** and **\$***, respectively (see under *Quoting* below).

`${parameter:-word}`

If *parameter* is set and has a non-empty value, then substitute its value; otherwise substitute *word*.

`${parameter:=word}`

If *parameter* is not set or has the empty string value, then set it to *word*; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

`${parameter:?word}`

If *parameter* is set and has a non-empty value, then substitute its value; otherwise, print *word* and exit from the shell (if not interactive). If *word* is omitted then a standard message is printed.

`${parameter:+word}`

If *parameter* is set and has a non-empty value then substitute *word*; otherwise substitute the empty string.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, `pwd` is executed only if `d` is not set or has the empty string value:

```
print ${d:-$(pwd)}
```

If the colon (`:`) is omitted from the above expressions, then the shell only checks whether *parameter* is set or not.

`${parameter:offset:length}`

`${parameter:offset}`

Expands to the portion of the value of *parameter* starting at the character (counting from `0`) determined by expanding *offset* as an arithmetic expression and consisting of the number of characters determined by the arithmetic expression defined by *length*. In the second form, the remainder of the value is used. If A negative *offset* counts backwards from the end of *parameter*. Note that one or more *blanks* is required in front of a minus sign to prevent the shell from interpreting the operator as `:-`. If *parameter* is `*` or `@`, or is an array name indexed by `*` or `@`, then *offset* and *length* refer to the array index and number of elements respectively. A negative *offset* is taken relative to one greater than the highest subscript for indexed arrays. The order for associative arrays is unspecified.

`${parameter#pattern}`

`${parameter##pattern}`

If the shell *pattern* matches the beginning of the value of *parameter*, then the value of this expansion is the value of the *parameter* with the matched portion deleted; otherwise the value of this *parameter* is substituted. In the first form the smallest matching pattern is deleted and in the second form the largest matching pattern is deleted. When *parameter* is `@`, `*`, or an array variable with subscript `@` or `*`, the substring operation is applied to each element in turn.

`${parameter%pattern}`

`${parameter%%pattern}`

If the shell *pattern* matches the end of the value of *parameter*, then the value of this expansion is the value of the *parameter* with the matched part deleted; otherwise substitute the value of *parameter*. In the first form the smallest matching pattern is deleted and in the second form the largest matching pattern is deleted. When *parameter* is `@`, `*`, or an array variable with subscript `@` or `*`, the substring operation is applied to each element in turn.

`${parameter/pattern/string}`

`${parameter//pattern/string}`

`${parameter/#pattern/string}`

`${parameter!%pattern/string}`

Expands *parameter* and replaces the longest match of *pattern* with the given *string*. Each occurrence of `\n` in *string* is replaced by the portion of *parameter* that matches the *n*-th subpattern. In the first form, only the first occurrence of *pattern* is replaced. In the second form, each match for *pattern* is replaced by the given *string*. The third form restricts the pattern match to the beginning of the string while the fourth form restricts the pattern match to the end of the string. In the first

and second forms, an empty *pattern* never matches. In the third and fourth forms, an empty *pattern* matches the beginning or the end of the string, respectively. When *string* is empty, the *pattern* will be deleted and the */* in front of *string* may be omitted. When *parameter* is *@*, ***, or an array variable with subscript *@* or ***, the substitution operation is applied to each element in turn. In this case, the *string* portion of *word* will be re-evaluated for each element.

Shell Variables.

The following parameters are automatically set by the shell:

- # The number of positional parameters in decimal.
- Options supplied to the shell on invocation or by the **set** command.
- ? The exit status returned by the last executed command. Its meaning depends on the command or function that defines it, but there are conventions that other commands often depend on: zero typically means 'success' or 'true', one typically means 'non-success' or 'false', and a value greater than one typically indicates some kind of error. Only the 8 least significant bits of **\$?** (values 0 to 255) are preserved when the exit status is passed on to a parent process, but within the same (sub)shell environment, it is a signed integer value with a range of possible values as shown by the commands **getconf INT_MIN** and **getconf INT_MAX**. Shell functions that run in the current environment may return status values in this range.
- \$ The process ID of the main shell process. Note that this value will not change in a subshell, even if the subshell runs in a different process. See also **.sh.pid**.
- Initially, the value of **_** is an absolute pathname of the shell or script being executed as passed in the *environment*. Subsequently it is assigned the last argument of the previous command. This parameter is not set for commands which are asynchronous. This parameter is also used to hold the name of the matching **MAIL** file when checking for mail. While defining a compound variable or a type, **_** is initialized as a reference to the compound variable or type. When a discipline function is invoked, **_** is initialized as a reference to the variable associated with the call to this function. Finally when **_** is used as the name of the first variable of a type definition, the new type is derived from the type of the first variable. (See *Type Variables* below.)
- ! The process ID of the last background command invoked or the most recent job put in the background with the **bg** built-in command.

.sh.command

When processing a **DEBUG** trap, this variable contains the current command line that is about to run. The value is in the same format as the output generated by the **xtrace** option (minus the preceding **PS4** prompt).

.sh.edchar

This variable contains the value of the keyboard character (or sequence of characters if the first character is an ESC, ASCII **033**) that has been entered when processing a **KEYBD** trap (see *Key Bindings* below). If the value is changed as part of the trap action, then the new value replaces the key (or key sequence) that caused the trap.

.sh.edcol

The character position of the cursor at the time of the most recent **KEYBD** trap.

.sh.edmode

Upon executing a **KEYBD** trap action, the value of this variable is set to the ESC control character if the shell is in **vi** input mode (See *Vi Editing Mode* below), or to the empty string value otherwise.

.sh.edtext

The characters in the input buffer at the time of the most recent **KEYBD** trap. The variable is unset when not processing a **KEYBD** trap.

- .sh.file** The pathname of the file that contains the current command.
- .sh.fun** The name of the current function that is being executed.
- .sh.level**
Set to the current call depth of functions and dot scripts. Normally, this variable is read-only, but while executing a **DEBUG** trap, its value may be changed to switch the current function scope to that of the specified level for the duration of the trap run, making it possible to access a parent scope for debugging purposes. When trap execution ends, the variable and the scope are restored. It is an error to assign a value lower than 0 (the global scope) or higher than the current call depth.
- .sh.lineno**
Set during a **DEBUG** trap to the line number for the caller of each function.
- .sh.match**
Whenever a match is found in a pattern matching operation using either the `[[` compound command (see *Conditional Expressions* below) or the expansions `${parameter#pattern}`, `${parameter%pattern}`, `${parameter/pattern/string}`, or one of their variants (see *Parameter Expansion* above), the match and its subpattern matches are stored in this indexed array, overwriting its previous values. The **0**-th element stores the complete match and the *i*-th element stores the *i*-th submatch. For `//`, the array is two-dimensional, with the first subscript indicating the most recent match and subpattern match, and the second subscript indicating which match with **0** representing the first match. If no match is found, **.sh.match** is not reset or modified. Note that even matching operations performed on the **.sh.match** variable itself will overwrite it upon finding a match.
- .sh.math**
Used for defining arithmetic functions (see *Arithmetic Evaluation* below) and stores the list of user defined arithmetic functions.
- .sh.name**
Set to the name of the variable at the time that a discipline function is invoked.
- .sh.subscript**
Set to the name subscript of the variable at the time that a discipline function is invoked.
- .sh.subshell**
The current depth for subshells and command substitution.
- .sh.pid** Set to the process ID of the current shell process. Unlike `$$`, this is updated in a subshell when it forks into a new process. Note that a virtual subshell may have to fork mid-execution due to various system- and implementation-dependent requirements, so the value should not be counted on to remain the same from one command to the next. If a persistent process ID is required for a subshell, it must be ensured it is running in its own process first. Any attempt to set a process limit using the `ulimit` built-in command, such as `ulimit -t unlimited 2>/dev/null`, is a reliable way to make a subshell fork if it hasn't already.
- .sh.ppid**
Set to the process ID of the parent of the current shell process. Unlike `$PPID`, this is updated in a subshell when it forks into a new process. The same note as for **.sh.pid** applies.
- .sh.value**
Set to the value of the variable at the time that the `set` or `append` discipline function is invoked. When a user defined arithmetic function is invoked, the value of **.sh.value** is saved and **.sh.value** is set to long double precision floating point. **.sh.value** is restored when the function returns.

.sh.version

Set to a value that identifies the version of this shell.

COLUMNS

Width of the terminal window in character positions. Updated automatically at initialization and on receiving a **SIGWINCH** signal. The shell uses the value to define the width of the edit window for the shell edit modes and for printing **select** lists.

KSH_VERSION

A name reference to **.sh.version**.

LINENO

The current line number within the script or function being executed.

LINES Height of the terminal window in lines. Updated automatically at initialization and on receiving a **SIGWINCH** signal. The shell uses the value to determine the column length for printing **select** lists: they are printed vertically until about two thirds of **LINES** lines are filled.

OLDPWD

The previous working directory set by the **cd** command.

OPTARG

The value of the last option argument processed by the **getopts** built-in command.

OPTIND

The index of the last option argument processed by the **getopts** built-in command.

PPID The process ID of the parent of the main shell process. Note that this value will not change in a subshell, even if the subshell runs in a different process. See also **.sh.ppid**.

PWD The present working directory set by the **cd** command.

RANDOM

Each time this variable is referenced, a pseudorandom integer is generated, uniformly distributed between 0 and 32767 (the 16-bit unsigned integer range) except that the same number is never repeated twice in a row. The sequence of pseudorandom numbers is reproducible and can be initialized to a fixed starting point by assigning a numeric seed value to **RANDOM**. Each time a new shell or subshell environment is entered (see *Subshells* above), the sequence is automatically reset to a different point.

REPLY This variable is set by the **select** statement and by the **read** built-in command when no arguments are supplied.

SECONDS

Each time this variable is referenced, the number of seconds since shell invocation is returned. If this variable 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.

SHLVL An integer variable that is incremented and exported each time the shell is invoked. If **SHLVL** is not in the environment when the shell is invoked, it is set to 1.

The following variables are used by the shell:

CDPATH

The search path for the **cd** command.

EDITOR

If the **VISUAL** variable is not set, the value of this variable will be checked for certain patterns and the corresponding editing option will be turned on as described with **VISUAL** below.

ENV If this variable is set, then parameter expansion, command substitution, and arithmetic expansion are performed on the value to generate the pathname of the script that will be executed when the shell is invoked interactively (see *Invocation* below). This file is

typically used for **alias** and **function** definitions. The default value is **\$HOME/.kshrc**. On systems that support a system wide **/etc/ksh.kshrc** initialization file, if the filename generated by the expansion of **ENV** begins with **./** or **./.** the system wide initialization file will not be executed.

FCEDIT

Obsolete name for the default editor name for the **hist** command. **FCEDIT** is not used when **HISTEDIT** is set.

FIGNORE

A pattern that defines the set of filenames that will be ignored when performing filename matching.

FPATH The search path for function definitions. The directories in this path are searched for a file with the same name as the function or command when a function with the **-u** attribute is referenced and when a command is not found. If an executable file with the name of that command is found, then it is read and executed in the current environment. Unlike **PATH**, the current directory must be represented explicitly by **.** rather than by adjacent **:** characters or a beginning or ending **:**.

histchars

This variable can be used to specify up to three ASCII characters that control history expansion (see *History Expansion* below). The first (default: **!**) signals the start of a history expansion. The second (default: **^**) is used for short-form substitutions. The third (default: **#**), when found as the first character of a word, causes history expansion to be skipped for the rest of the words on the line. Multi-byte characters (e.g. UTF-8) are not supported and produce undefined results.

HISTCMD

Number of the current command in the history file.

HISTEDIT

Name for the default editor name for the **hist** command.

HISTFILE

If this variable is set when the shell is invoked, then the value is the pathname of the file that will be used to store the command history (see *Command Re-entry* below).

HISTSIZE

If this variable is set when the shell is invoked, then the number of previously entered commands that are accessible by this shell will be greater than or equal to this number. The default is 512.

HOME The default argument (home directory) for the **cd** command.

IFS Internal field separators, normally **space**, **tab**, and **new-line** that are used to separate the results of command substitution or parameter expansion and to separate fields with the built-in command **read**. The first character of the **IFS** variable is used to separate arguments for the **"\$*"** expansion (see *Quoting* below). Each single occurrence of an **IFS** character in the string to be split that is not in the *isspace* character class, and any adjacent characters in **IFS** that are in the *isspace* character class, delimit a field. One or more characters in **IFS** that belong to the *isspace* character class delimit a field. In addition, if the same *isspace* character appears consecutively inside **IFS** and the **posix** shell option is not on, this character is treated as if it were not in the *isspace* class - for example, if **IFS** consists of two **tab** characters, then two adjacent **tab** characters delimit an empty field.

JOBMAX

This variable defines the maximum number running background jobs that can run at a time. When this limit is reached, the shell will wait for a job to complete before starting a new job.

- LANG** This variable determines the locale category for any category not specifically selected with a variable starting with **LC_** or **LANG**.
- LC_ALL**
This variable overrides the value of the **LANG** variable and any other **LC_** variable.
- LC_COLLATE**
This variable determines the locale category for character collation information.
- LC_CTYPE**
This variable determines the locale category for character handling functions. It determines the character classes for pattern matching (see *Pathname Expansion* below).
- LC_NUMERIC**
This variable determines the locale category for the decimal point character.
- MAIL** If this variable is set to the name of a mail file *and* the **MAILPATH** variable is not set, then the shell informs the user of arrival of mail in the specified file.
- MAILCHECK**
This variable specifies how often (in seconds) the shell will check for changes in the modification time of any of the files specified by the **MAILPATH** or **MAIL** variables. The default value is 600 seconds. When the time has elapsed the shell will check before issuing the next prompt.
- MAILPATH**
A colon (:) separated list of file names. If this variable is set, then the shell informs the user of any modifications to the specified files that have occurred within the last **MAILCHECK** seconds. Each file name 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 mail in \$_*.
- PATH** The search path for commands (see *Execution* below). The user may not change **PATH** if executing under **rksh** (except in **.profile**).
- PS1** Every time a new command line is started on an interactive shell, the value of this variable is expanded to resolve backslash escaping, parameter expansion, command substitution, and arithmetic expansion. The result defines the primary prompt string for that command line. The default is “\$ ”. The character ! in the primary prompt string is replaced by the *command* number (see *Command Re-entry* below). Two successive occurrences of ! will produce a single ! when the prompt string is printed. Note that any terminal escape sequences used in the **PS1** prompt thus need every instance of ! in them to be changed to !!.
- PS2** Secondary prompt string, by default “> ”.
- PS3** Selection prompt string used within a **select** loop, by default “#? ”.
- PS4** The value of this variable is expanded for parameter evaluation, command substitution, and arithmetic expansion and precedes each line of an execution trace. By default, **PS4** is “+ ”. In addition when **PS4** is unset, the execution trace prompt is also “+ ”.
- SHELL** The pathname of the *shell* is kept in the environment. At invocation, if the basename of this variable is **rsh**, **rksh**, or **krsh**, then the shell becomes restricted.
- TIMEFORMAT**
The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the **time** reserved word should be displayed. The % character introduces a format sequence that is expanded to a time value or other information. The format sequences and their meanings are as follows.

<code>%%</code>	A literal %.
<code>%[p][I]R</code>	The elapsed time in seconds.
<code>%[p][I]U</code>	The number of CPU seconds spent in user mode.
<code>%[p][I]S</code>	The number of CPU seconds spent in system mode.
<code>%P</code>	The CPU percentage, computed as (U + S) / R.

The brackets denote optional portions. The optional *p* is a digit specifying the *precision*, the number of fractional digits after a decimal point. A value of 0 causes no decimal point or fraction to be output. At most three places after the decimal point can be displayed; values of *p* greater than 3 are treated as 3. If *p* is not specified, the value 3 is used.

The optional **I** specifies a longer format, including hours if greater than zero, minutes, and seconds of the form *HHhMMmSS.FFs*. The value of *p* determines whether or not the fraction is included. Seconds are zero-padded unless the **posix** shell option is on.

All other characters are output without change and a trailing newline is added. If the variable is unset, the default value, `$'\nreal\t%2IR\nuser\t%2IU\nsys\t%2IS'`, is used. If the value is empty, no timing information is displayed.

TMOUT

Terminal read timeout. If set to a value greater than zero, the **read** built-in command and the **select** compound command time out after **TMOUT** seconds when input is from a terminal. An interactive shell will issue a warning and allow for an extra 60 second timeout grace period before terminating if a line is not entered within the prescribed number of seconds while reading from a terminal. (Note that the shell can be compiled with a maximum bound for this value which cannot be exceeded.)

VISUAL

The value of this variable is scanned when the shell is invoked and whenever its value is changed; if it is found to match certain patterns, the corresponding line editor (see *In-line Editing Options* below) is activated. If it matches the pattern `*[Vv][Ii]*`, the **vi** option is turned on; else if it matches the pattern `*gmacs*`, the **gmacs** option is turned on; else if it matches the pattern `*macs*`, the **emacs** option is turned on. If none of the patterns match, **emacs** is turned on by default upon initializing an interactive shell. If the value is changed by assignment and none of the patterns match, no options are changed. The value of **VISUAL** overrides the value of **EDITOR**.

The shell gives default values to **PATH**, **PS1**, **PS2**, **PS3**, **PS4**, **MAILCHECK**, **FCEDIT**, **TMOUT** and **IFS**, while **HOME**, **SHELL**, **ENV**, **histchars**, and **MAIL** are not set at all by the shell (although **HOME** is set by *login*(1)). On some systems **MAIL** and **SHELL** are also set by *login*(1).

Field Splitting.

After parameter expansion and command substitution, the results of substitutions are scanned for the field separator characters (those found in **IFS**) and split into distinct fields where such characters are found. Explicit empty fields ("" or '') are retained. Implicit empty fields (those resulting from *parameters* that are unset or have empty string values or from command substitutions yielding the empty string, and that are not quoted with "") are removed.

Brace Expansion.

If the **braceexpand** (**-B**) option is set then each *word*, as well as any fields resulting from *field splitting* (see above), are checked to see if they contain one or more of the brace patterns `{*,*}`, `{I1..I2}`, `{n1..n2}`, `{n1..n2%fmt}`, `{n1..n2..n3}`, or `{n1..n2..n3%fmt}`, where *** represents any character, *I1*, *I2* are letters and *n1*, *n2*, *n3* are signed numbers and *fmt* is a format specified as used by **printf**. In each case, fields are created by prepending the characters before the { and appending the characters after the } to each of the strings generated by the characters between the { and }. The resulting fields are checked to see if they have any brace patterns.

In the first form, a field is created for each string between { and ,, between , and ,, and between , and }. The string represented by *** can contain embedded matching { and } without quoting. Otherwise, each { and }

with * must be quoted.

In the second form, *l1* and *l2* must both be either upper case or both be lower case characters in the C locale. In this case a field is created for each character from *l1* through *l2*.

In the remaining forms, a field is created for each number starting at *n1* and continuing until it reaches *n2* incrementing *n1* by *n3*. The values of *n1*, *n2* and *n3* are limited to the standard integer range as output by **getconf INT_MIN** and **getconf INT_MAX**; the behavior is undefined if this range is exceeded. The cases where *n3* is not specified behave as if *n3* where **1** if *n1* ≤ *n2* and **-1** otherwise. If forms which specify *%fmt* any format flags, widths and precisions can be specified and *fmt* can end in any of the specifiers **cdiouxX**. For example, **{a,z}{1..5..3%02d}{b..c}x** expands to the 8 fields, **a01bx**, **a01cx**, **a04bx**, **a04cx**, **z01bx**, **z01cx**, **z04bx** and **z04cx**.

Pathname Expansion.

This is also known as *globbing* or sometimes *filename generation*. Pathname expansion is disabled if the **-f** a.k.a. **--noglob** shell option is on. Otherwise, if certain special characters are found in a *word* or in a field resulting from *field splitting* (see above), then the word or field is regarded as a *pattern*. Each literal *word* is scanned for the characters *, ?, [, and (, but fields resulting from *field splitting* are scanned only for the characters *, ?, and [for compatibility reasons (in which case the (character is not special and any pattern syntax described below that involves parentheses does not apply).

Each file name component that contains a recognized pattern character is replaced with a lexicographically sorted set of names that matches the pattern from that directory. If no file name is found that matches the pattern, then that component of the filename is left unchanged unless the pattern is prefixed with **~(N)**, in which case it is removed as described below. The special traversal names **.** and **..** are never matched. If **FIGNORE** is set, then each file name component that matches the pattern defined by the value of **FIGNORE** is ignored when generating the matching filenames. If **FIGNORE** is not set, the character **.** at the start of each file name component will be ignored unless the first character of the pattern corresponding to this component is the character **.** itself. Note that, for uses of pattern matching other than pathname expansion, the **/** and **.** are not treated specially.

- * Matches any string, including the empty string. When used for filename expansion, if the **globstar** option is on, an isolated pattern of two adjacent *s will match all files and zero or more directories and subdirectories. If followed by a **/** then only directories and subdirectories will match.
- ? Matches any single character.
- [...] Matches any one of the enclosed characters. A pair of characters separated by **-** matches any character lexically between the pair, inclusive. If the first character following the opening [is a **!** or **^**, then any character not enclosed is matched. A **-** can be included in the character set by putting it as the first or last character.
Within [and], character classes can be specified with the syntax **[:class:]**, where *class* is one of the following classes defined in the ANSI C standard (note that **word** is equivalent to **alnum** plus the character **_**):
alnum alpha blank cntrl digit graph lower print punct space upper word xdigit
Within [and], an equivalence class can be specified with the syntax **[=c=]** which matches all characters with the same primary collation weight (as defined by the current locale) as the character *c*. Within [and], **[.symbol.]** matches the collating symbol *symbol*.

A *pattern-list* is a list of one or more patterns separated from each other with a **&** or **|**. A **&** signifies that all patterns must be matched whereas **|** requires that only one pattern be matched. Composite patterns can be formed with one or more of the following subpatterns:

- ?(*pattern-list*)
Optionally matches any one of the given patterns.
- *(*pattern-list*)
Matches zero or more occurrences of the given patterns.

- +(*pattern-list*)**
Matches one or more occurrences of the given patterns.
- {*n*}(*pattern-list*)**
Matches *n* occurrences of the given patterns.
- {*m,n*}(*pattern-list*)**
Matches from *m* to *n* occurrences of the given patterns. If *m* is omitted, **0** will be used. If *n* is omitted, at least *m* occurrences will be matched.
- @(*pattern-list*)**
Matches exactly one of the given patterns.
- !(*pattern-list*)**
Matches anything except one of the given patterns.

By default, each pattern or subpattern will match the longest string possible consistent with generating the longest overall match. If more than one match is possible, the one starting closest to the beginning of the string will be chosen. However, for each of the above compound patterns, a **-** can be inserted in front of the **(** to cause the shortest match to the specified *pattern-list* to be used.

When *pattern-list* is contained within parentheses, the backslash character **** is treated specially even when inside a character class. All ANSI C character escapes are recognized and match the specified character. In addition, the following escape sequences are recognized:

- \d** Matches any character in the **digit** class.
- \D** Matches any character not in the **digit** class.
- \s** Matches any character in the **space** class.
- \S** Matches any character not in the **space** class.
- \w** Matches any character in the **word** class.
- \W** Matches any character not in the **word** class.

A pattern of the form **%(*pattern-pair*(s))** is a subpattern that can be used to match nested character expressions. Each *pattern-pair* is a two-character sequence that cannot contain **&** or **|**. The first *pattern-pair* specifies the starting and ending characters for the match. Each subsequent *pattern-pair* represents the beginning and ending characters of a nested group that will be skipped over when counting starting and ending character matches. The behavior is unspecified when the first character of a *pattern-pair* is alphanumeric, except for the following:

- D** Causes the ending character to terminate the search for this pattern without finding a match.
- E** Causes the ending character to be interpreted as an escape character.
- L** Causes the ending character to be interpreted as a quote character, causing all characters to be ignored when looking for a match.
- Q** Causes the ending character to be interpreted as a quote character, causing all characters other than any escape character to be ignored when looking for a match.

Thus, **%({}Q"EA)**, matches characters starting at **{** until the matching **}** is found, not counting any **{** or **}** that is inside a double-quoted string or preceded by the escape character ****. Without the **{}**, this pattern matches any C language string.

Each subpattern in a composite pattern is numbered, starting at 1, by the location of the **(** within the pattern. The sequence **\n**, where *n* is a single digit and **\n** comes after the *n*th subpattern, matches the same string as the subpattern itself.

Finally, a pattern can contain subpatterns of the form **~(*options*:*pattern-list*)**, where either *options* or **:*pattern-list*** can be omitted. Unlike the other compound patterns, these subpatterns are not counted in the numbered subpatterns. **:*pattern-list*** must be omitted for options **E**, **F**, **G**, **N**, **P**, **V**, and **X** below. If *options* is present, it can consist of one or more of the following:

- +** Enable the following options. This is the default.
- Disable the following options.
- E** The remainder of the pattern uses extended regular expression syntax like the **-E** option of the *grep*(1) command.

F	The remainder of the pattern uses the fixed pattern syntax of the -F option of the <i>grep(1)</i> command.
G	The remainder of the pattern uses basic regular expression syntax like the <i>grep(1)</i> command without options.
K	The remainder of the pattern uses shell pattern syntax. This is the default.
N	When it is the first letter and is used with pathname expansion, and no matches occur, the file pattern expands to the empty string instead of remaining unexpanded. Otherwise, it is ignored.
X	The remainder of the pattern uses augmented regular expression syntax like the -X option of the AT&T AST version of the <i>grep(1)</i> command.
P	The remainder of the pattern uses <i>perl(1)</i> regular expression syntax. Not all perl regular expression syntax is currently implemented.
V	The remainder of the pattern uses System V regular expression syntax.
i	Always treat the match as case-insensitive, regardless of the globcasedetect shell option.
g	File the longest match (greedy). This is the default.
l	Left-anchor the pattern. This is the default for K style patterns.
r	Right-anchor the pattern. This is the default for K style patterns.

If both *options* and *:pattern-list* are specified, then the options apply only to *pattern-list*. Otherwise, these options remain in effect until they are disabled by a subsequent *~(...)* or at the end of the subpattern containing *~(...)*.

Quoting.

Each of the *metacharacters* listed earlier (see *Definitions* above) has a special meaning to the shell and causes termination of a word unless quoted. A character may be *quoted* (i.e., made to stand for itself) by preceding it with a ****. The pair **\new-line** is removed. All characters enclosed between a pair of single quote marks (') that is not preceded by a **\$** are quoted. A single quote cannot appear within the single quotes. A single quoted string preceded by an unquoted **\$** is processed as a C99 string except for the following:

\0	Causes the remainder of the string to be ignored.
\E	Equivalent to the escape character (ASCII 033),
\e	Equivalent to the escape character (ASCII 033),
\cx	Expands to the character control- <i>x</i> .
\C[.name.]	Expands to the collating element <i>name</i> .

Inside double quote marks (""), parameter and command substitution occur and **** quotes the characters ****, **`**, **"**, and **\$**. A **\$** in front of a double quoted string will be ignored in the "C" or "POSIX" locale, and may cause the string to be replaced by a locale specific string otherwise. The meaning of **\$*** and **\$@** is identical when not quoted or when used as a variable assignment value or as a file name. However, when used as a command argument, **"\$*"** is equivalent to **"\$1\$d\$2d..."**, where *d* is the first character of the **IFS** variable, whereas **"\$@"** is equivalent to **"\$1" "\$2" ...**. Inside grave quote marks (`), **** quotes the characters ****, **`**, and **\$**. If the grave quotes occur within double quotes, then **** also quotes the character **`**.

The special meaning of reserved words or aliases can be removed by quoting any character of the reserved word. The recognition of function names or built-in command names listed below cannot be altered by quoting them.

Arithmetic Evaluation.

The shell performs arithmetic evaluation for arithmetic expansion, to evaluate an arithmetic command, to evaluate an indexed array subscript, and to evaluate arguments to the built-in commands **shift** and **let** as well as arguments to numeric format specifiers given to **print -f** and **printf**. Evaluations are performed using double precision floating point arithmetic or long double precision floating point for systems that provide this data type. Floating point constants follow the ANSI C programming language floating point conventions. The case-insensitive floating point constants **NaN** and **Inf** can be used to represent "not a number" and infinity respectively, unless the **posix** shell option is on. Integer constants follow the ANSI C programming language integer constant conventions although only single byte character constants are

recognized and character casts are not recognized.

In addition, integer constants can be of the form *base#n* or (for negative numbers) *-base#n*, where *base* is a decimal number between two and sixty-four representing the arithmetic base, and *n* is an unsigned integer in that base. The digits above 9 are represented by the lower case letters, the upper case letters, @, and _, respectively. For bases less than or equal to 36, upper and lower case letters can be used interchangeably.

An arithmetic expression uses the same syntax, precedence, and associativity of expression as the C language. All the C language operators that apply to floating point quantities can be used. In addition, the operator ****** can be used for exponentiation. It has higher precedence than multiplication and is left associative. In addition, when the value of an arithmetic variable or subexpression can be represented as a long integer, all C language integer arithmetic operations can be performed. Variables can be referenced by name within an arithmetic expression without using the parameter expansion syntax. When a variable is referenced, its value is evaluated as an arithmetic expression.

Any of the following math library functions that are in the C math library can be used within an arithmetic expression:

```
abs acos acosh asin asinh atan atan2 atanh cbrt ceil copysign cos cosh erf
erfc exp exp10 exp2 expm1 fabs fdim finite float floor fma fmax fmin fmod
fpclass fpclassify hypot ilogb int isfinite isgreater isgreaterequal isinf isinfinite is-
less islessequal islessgreater isnan isnormal issubnormal isunordered iszero j0 j1
jn ldexp lgamma log log10 log1p log2 logb nearbyint nextafter nexttoward pow
remainder rint round scalb scalbn signbit sin sinh sqrt tan tanh tgamma trunc
y0 y1 yn
```

In addition, arithmetic functions can be defined as shell functions with a variant of the **function name** syntax,

```
function .sh.math.name ident ... { list ;}
```

where *name* is the function name used in the arithmetic expression and each identifier, *ident* is a name reference to the long double precision floating point argument. The value of **.sh.value** when the function returns is the value of this function. User defined functions can take up to 3 arguments and override C math library functions.

An internal representation of a *variable* as a double precision floating point can be specified with the **-E** [*n*], **-F** [*n*], or **-X** [*n*] option of the **typeset** special built-in command. The **-E** option causes the expansion of the value to be represented using scientific notation when it is expanded. The optional option argument *n* defines the number of significant figures. The **-F** option causes the expansion to be represented as a floating decimal number when it is expanded. The **-X** option causes the expansion to be represented using the **%a** format defined by ISO C-99. The optional option argument *n* defines the number of places after the decimal (or radix) point in this case.

An internal integer representation of a *variable* can be specified with the **-i** [*n*] option of the **typeset** special built-in command. The optional option argument *n* specifies an arithmetic base to be used when expanding the variable. If you do not specify an arithmetic base, base 10 will be used.

Arithmetic evaluation is performed on the value of each assignment to a variable with the **-E**, **-F**, **-X**, or **-i** attribute. Assigning a floating point number to a variable whose type is an integer causes the fractional part to be truncated.

Prompting.

When used interactively, the shell prompts with the value of **PS1** after expanding it for parameter expansion, command substitution, and arithmetic expansion, before reading a command. In addition, each single **!** in the prompt is replaced by the command number. A **!!** is required to place **!** in the prompt. If at any time a new-line is typed and further input is needed to complete a command, then the secondary prompt (i.e., the value of **PS2**) is issued.

Conditional Expressions.

A *conditional expression* is used with the `[[` compound command to test attributes of files and to compare strings. Field splitting and pathname expansion are not performed on the words between `[[` and `]]`. Each expression can be constructed from one or more of the following unary or binary expressions:

- string* Same as `-n string` below.
- `-a file` Same as `-e` below. This is obsolete.
- `-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 an ordinary file.
- `-g file` True if *file* exists and it has its setgid bit set.
- `-k file` True if *file* exists and it has its sticky bit set.
- `-n string`
True if length of *string* is non-zero.
- `-o ?option`
True if option named *option* is a valid option name.
- `-o option`
True if option named *option* is on.
- `-p file` True if *file* exists and is a fifo special file or a 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 fildes`
True if file descriptor number *fildes* is open and associated with a terminal device.
- `-u file` True if *file* exists and it has its setuid bit set.
- `-v name`
True if variable *name* is a valid variable name and 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 true if 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.
- `-h file` True if *file* exists and is a symbolic link.
- `-N file` True if *file* exists and the modification time is greater than the last access time.
- `-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.
- `-R name`
True if variable *name* is a name reference.
- `-S file` True if *file* exists and is a socket.
- `file1 -nt file2`
True if *file1* exists and *file2* does not, or *file1* is newer than *file2*.
- `file1 -ot file2`
True if *file2* exists and *file1* does not, or *file1* is older than *file2*.
- `file1 -ef file2`
True if *file1* and *file2* exist and refer to the same file.
- `string == pattern`
True if *string* matches *pattern*. Any part of *pattern* can be quoted to cause it to be matched as a string. With a successful match to a pattern, the `.sh.match` array variable will contain the match and subpattern matches.
- `string = pattern`
Same as `==` above, but is obsolete.

string **!=** *pattern*

True if *string* does not match *pattern*. When the *string* matches the *pattern* the **.sh.match** array variable will contain the match and subpattern matches.

string **=~** *ere*

True if *string* matches the pattern **~(E)ere** where *ere* is an extended regular expression.

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.

The following obsolete arithmetic comparisons are also permitted:

exp1 **-eq** *exp2*

True if *exp1* is equal to *exp2*.

exp1 **-ne** *exp2*

True if *exp1* is not equal to *exp2*.

exp1 **-lt** *exp2*

True if *exp1* is less than *exp2*.

exp1 **-gt** *exp2*

True if *exp1* is greater than *exp2*.

exp1 **-le** *exp2*

True if *exp1* is less than or equal to *exp2*.

exp1 **-ge** *exp2*

True if *exp1* is greater than or equal to *exp2*.

In each of the above expressions, if *file* is of the form **/dev/fd/*n***, where *n* is an integer, then the test is applied to the open file whose descriptor number is *n*. The **posix** shell option disables this special handling.

A compound expression can be constructed from these primitives by using any of the following, listed in decreasing order of precedence:

(*expression*)

True if *expression* is true. Used to group expressions.

! *expression*

True if *expression* is false.

expression1 **&&** *expression2*

True if *expression1* and *expression2* are both true.

expression1 **||** *expression2*

True if either *expression1* or *expression2* is true.

Input/Output.

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may precede or follow a *command* and are *not* passed on to the invoked command. Command substitution, parameter expansion, and arithmetic expansion occur before *word* or *digit* is used except as noted below. Pathname expansion occurs only if the shell is interactive and the pattern matches a single file. Field splitting is not performed.

In each of the following redirections, if *file* is of the form **/dev/sctp/host/port**, **/dev/tcp/host/port**, or **/dev/udp/host/port**, where *host* is a hostname or host address, and *port* is a service given by name or an integer port number, then the redirection attempts to make a **tcp**, **sctp** or **udp** connection to the corresponding socket.

No intervening space is allowed between the characters of redirection operators.

<*word* Use file *word* as standard input (file descriptor 0).

>*word* Use file *word* as standard output (file descriptor 1). If the file does not exist then it is created. If the file exists, and the **noclobber** option is on, this causes an error; otherwise, it is truncated to zero length.

> <i>word</i>	Same as >, except that it overrides the noclobber option.
>; <i>word</i>	Write output to a temporary file. If the command completes successfully rename it to <i>word</i> , otherwise, delete the temporary file. >; <i>word</i> cannot be used with the <i>exec</i> and <i>redirect</i> built-ins.
>> <i>word</i>	Use file <i>word</i> as standard output. If the file exists, then output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<> <i>word</i>	Open file <i>word</i> for reading and writing as standard output. If the posix option is active, it defaults to standard input instead.
<>; <i>word</i>	The same as <> <i>word</i> except that if the command completes successfully, <i>word</i> is truncated to the offset at command completion. <>; <i>word</i> cannot be used with the <i>exec</i> and <i>redirect</i> built-ins.
<<[-] <i>word</i>	The shell input is read up to a line that is the same as <i>word</i> after any quoting has been removed, or to an end-of-file. No parameter expansion, command substitution, arithmetic expansion or pathname expansion is performed on <i>word</i> . The resulting document, called a <i>here-document</i> , becomes the standard input. If any character of <i>word</i> is quoted, then no interpretation is placed upon the characters of the document; otherwise, parameter expansion, command substitution, and arithmetic expansion occur, \new-line is ignored, and \ must be used to quote the characters \ , \$, ^ . If - is appended to <<, then all leading tabs are stripped from <i>word</i> and from the document. If # is appended to <<, then leading spaces and tabs will be stripped off the first line of the document and up to an equivalent indentation will be stripped from the remaining lines and from <i>word</i> . A tab stop is assumed to occur at every 8 columns for the purposes of determining the indentation.
<<< <i>word</i>	A short form of here document in which <i>word</i> becomes the contents of the here-document after any parameter expansion, command substitution, and arithmetic expansion occur.
<& <i>digit</i>	The standard input is duplicated from file descriptor <i>digit</i> (see <i>dup(2)</i>).
>& <i>digit</i>	The standard output is duplicated from file descriptor <i>digit</i> .
<& <i>digit</i> -	The file descriptor given by <i>digit</i> is moved to standard input.
>& <i>digit</i> -	The file descriptor given by <i>digit</i> is moved to standard output.
<&-	The standard input is closed.
>&-	The standard output is closed.
<& p	The input from the co-process is moved to standard input.
>& p	The output to the co-process is moved to standard output.
<#(<i>expr</i>)	Evaluate arithmetic expression <i>expr</i> and position file descriptor 0 to the resulting value bytes from the start of the file. The variables CUR and EOF evaluate to the current offset and end-of-file offset respectively when evaluating <i>expr</i> .
>#(<i>offset</i>)	The same as <# except applies to file descriptor 1.
<# <i>pattern</i>	Seeks forward to the beginning of the next line containing <i>pattern</i> .
<## <i>pattern</i>	The same as <# except that the portion of the file that is skipped is copied to standard output.

If one of the above is preceded by a digit, with no intervening space, then the file descriptor number referred to is that specified by the digit (instead of the default 0 or 1). If one of the above, other than >&- and the ># and <# forms, is preceded by {*varname*} with no intervening space, then a file descriptor number > 9 will be selected by the shell and stored in the variable *varname*, so it can be read from or written to with redirections like <& *\$varname* or >& *\$varname*. If >&- or the any of the ># and <# forms is preceded by {*varname*} the value of *varname* defines the file descriptor to close or position. For example:

```
... 2>&1
```

means file descriptor 2 is to be opened for writing as a duplicate of file descriptor 1 and

```
exec {n}<file
```

means open file named **file** for reading and store the file descriptor number in variable **n**.

A special shorthand redirection operator **&>word** is available; it is equivalent to **>word 2>&1**. It cannot be preceded by any digit or variable name. This shorthand is disabled if the **posix** shell option is active.

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 (i.e. *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*.

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.

Environment.

The *environment* (see *environ(7)*) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The names must be *identifiers* and the values are character strings. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a variable for each name found, giving it the corresponding value and attributes and marking it *export*. Executed commands inherit the environment. If the user modifies the values of these variables or creates new ones, using the **export** or **typeset -x** commands, they become part of the environment. The environment seen by any executed command is thus composed of any name-value pairs originally inherited by the shell, whose values may be modified by the current shell, plus any additions which must be noted in **export** or **typeset -x** commands.

Exported variables pass their attributes through the environment so that a newly invoked ksh that is a child or exec'd process of the current shell will automatically import them, unless the **posix** shell option is on. As of ksh 93u+m/1.0, this never includes the **readonly** attribute. Passing attributes through the environment is deprecated and this feature will be removed in ksh 93u+m/1.1.

The environment for any *simple-command* or function may be augmented by prefixing it with one or more variable assignments. A variable assignment argument is a word of the form *identifier=value*. Thus:

```
TERM=450 cmd args and  
(export TERM; TERM=450; cmd args)
```

are equivalent (as far as the above execution of *cmd* is concerned except for special built-in commands listed below – those that are marked with †).

If the obsolete **-k** option is set, *all* variable assignment arguments are placed in the environment, even if they occur after the command name. The following first prints **a=b c** and then **c**:

```
echo a=b c  
set -k  
echo a=b c
```

This feature is intended for use with scripts written for early versions of the shell and its use in new scripts is strongly discouraged. It is likely to disappear someday.

Functions.

For historical reasons, there are two ways to define functions, the *name()* syntax and the **function name** syntax, described in the *Commands* section above. 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 *Execution* below.)

Functions defined by the **function name** syntax and called by name execute in the same process as the

caller and share all files and present working directory with the caller. All changes to shell options are local to the function call. The **errexit** option is turned off in the function's scope. The same applies to the **xtrace** option, unless the **functrace** shell option is set or the function was tagged using **typeset -f -t**. The state of all other shell options is inherited from the caller. Traps caught by the caller are reset to their default action inside the function. A trap condition that is not caught or ignored by the function causes the function to terminate and the condition to be passed on to the caller. A trap on **EXIT** set inside a function is executed in the environment of the caller after the function completes. Ordinarily, variables are shared between the calling program and the function. However, the **typeset** special built-in command used within a function defines local variables whose scope includes the current function. They can be passed to functions that they call in the variable assignment list that precedes the call or as arguments passed as name references. Errors within functions return control to the caller.

Functions defined with the *name()* syntax and functions defined with the **function name** syntax that are invoked with the **.** special built-in are executed in the caller's environment and share all variables, options and traps with the caller. Errors within these function executions cause the script that contains them to abort.

The special built-in command **return** is used to return from function calls.

Function names can be listed with the **-f** or **+f** option of the **typeset** special built-in command. The text of functions, when available, will also be listed with **-f**. Functions can be undefined with the **-f** option of the **unset** special built-in command.

Ordinarily, functions are unset when the shell executes a shell script. Functions that need to be defined across separate invocations of the shell should be placed in a directory and the **FPATH** variable should contain the name of this directory. They may also be specified in the **ENV** file.

Discipline Functions.

Each variable can have zero or more discipline functions associated with it. The shell initially understands the discipline names **get**, **set**, **append**, and **unset** but can be added when defining new types. On most systems others can be added at run time via the C programming interface extension provided by the **builtin** built-in utility. If the **get** discipline is defined for a variable, it is invoked whenever the given variable is referenced. If the variable **.sh.value** is assigned a value inside the discipline function, the referenced variable will evaluate to this value instead. If the **set** discipline is defined for a variable, it is invoked whenever the given variable is assigned a value. If the **append** discipline is defined for a variable, it is invoked whenever a value is appended to the given variable. The variable **.sh.value** is given the value of the variable before invoking the discipline, and the variable will be assigned the value of **.sh.value** after the discipline completes. If **.sh.value** is unset inside the discipline, then that value is unchanged. If the **unset** discipline is defined for a variable, it is invoked whenever the given variable is unset.

The variable **.sh.name** contains the name of the variable for which the discipline function is called, **.sh.subscript** is the subscript of the variable, and **.sh.value** will contain the value being assigned inside the **set** discipline function. The variable **_** is a reference to the variable including the subscript if any. For the **set** discipline, changing **.sh.value** will change the value that gets assigned. Finally, the expansion **\${var.name}**, when *name* is the name of a discipline, and there is no variable of this name, is equivalent to the command substitution **\${ var.name ;}**.

Name Spaces.

Commands and functions that are executed as part of the *list* of a **namespace** command that modify variables or create new ones, create a new variable whose name is the name of the name space as given by *identifier* preceded by **.**. When a variable whose name is *name* is referenced, it is first searched for using **.identifier.name**. Similarly, a function defined by a command in the **namespace list** is created using the name space name preceded by **.**.

When the *list* of a **namespace** command contains a **namespace** command, the names of variables and functions that are created consist of the variable or function name preceded by the list of *identifiers* each preceded by **.**.

Outside of a name space, a variable or function created inside a name space can be referenced by preceding

it with the name space name.

By default, variables starting with **.sh** are in the **sh** name space.

Type Variables.

Typed variables provide a way to create data structure and objects. A type can be defined either by a shared library, by the **enum** built-in command described below, or by using the new **-T** option of the **typeset** built-in command. With the **-T** option of **typeset**, the type name, specified as an option argument to **-T**, is set with a compound variable assignment that defines the type. Function definitions can appear inside the compound variable assignment and these become discipline functions for this type and can be invoked or redefined by each instance of the type. The function name **create** is treated specially. It is invoked for each instance of the type that is created but is not inherited and cannot be redefined for each instance.

When a type is defined a special built-in command of that name is added. These built-ins are declaration commands and follow the same expansion rules as the built-in commands described below that are marked with a ‡ symbol. These commands can subsequently be used inside further type definitions. The man page for these commands can be generated by using the **--man** option or any of the other **--** options described with **getopts**. The **-r**, **-a**, **-A**, **-h**, and **-S** options of **typeset** are permitted with each of these new built-ins.

An instance of a type is created by invoking the type name followed by one or more instance names. Each instance of the type is initialized with a copy of the subvariables except for subvariables that are defined with the **-S** option. Variables defined with the **-S** are shared by all instances of the type. Each instance can change the value of any subvariable and can also define new discipline functions of the same names as those defined by the type definition as well as any standard discipline names. No additional subvariables can be defined for any instance.

When defining a type, if the value of a subvariable is not set and the **-r** attribute is specified, it causes the subvariable to be a required subvariable. Whenever an instance of a type is created, all required subvariables must be specified. These subvariables become read-only in each instance.

When **unset** is invoked on a subvariable within a type, and the **-r** attribute has not been specified for this field, the value is reset to the default value associative with the type. Invoking **unset** on a type instance not contained within another type deletes all subvariables and the variable itself.

A type definition can be derived from another type definition by defining the first subvariable name as **_** and defining its type as the base type. Any remaining definitions will be additions and modifications that apply to the new type. If the new type name is the same as that of the base type, the type will be replaced and the original type will no longer be accessible.

The **typeset** command with the **-T** and no option argument or operands will write all the type definitions to standard output in a form that can be read in to create all they types.

Jobs.

If the **monitor** option of the **set** command is turned on, 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 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 you are running a job and wish to do something else, you may hit the key **^Z** (control-Z) which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Stopped', 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 stop if it tries to read from the terminal. 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 stop when they try to produce output like they do when they try to read input.

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`
Any job whose command line begins with *string*.
- `%?string`
Any 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, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. The **notify** option of the **set** command causes the shell to print these job change messages as soon as they occur. If this happens while you are typing a command with one of the built-in line editors active, the job change message appears above your input without disturbing your command entry.

When the **monitor** option 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 stopped, you will be warned that ‘You have stopped(running) jobs.’ You may use the **jobs** command to see what they are. If you immediately try to exit again, the shell will not warn you a second time, and the stopped jobs will be terminated. When a login shell receives a HUP signal, it sends a HUP signal to each job that has not been disowned with the **disown** built-in command described below.

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. Otherwise, signals have the values inherited by the shell from its parent (but see also the **trap** built-in command below).

Execution.

Each time a command is read, the above expansions and substitutions are carried out. If the command name matches one of the *Special Built-in Commands* listed below, it is executed within the current shell process. Next, the command name is checked to see if it matches a user defined function. If it does, the positional parameters are saved and then reset to the arguments of the *function* call. A function is also executed in the current shell process. When the *function* completes or issues a **return**, the positional parameter list is restored. For functions defined with the **function name** syntax, any trap set on **EXIT** within the function is executed. The exit value of a *function* is the value of the last command executed. If a command name is not a *special built-in command* or a user defined *function*, but it is one of the built-in commands listed below, it is executed in the current shell process.

The shell variables **PATH** followed by the variable **FPATH** defines the list of directories to search for the command name. Alternative directory names are separated by a colon (:). The default path is the value that was output by **getconf PATH** at the time ksh was compiled. The current directory can be specified by two or more adjacent colons, or by a colon at the beginning or end of the path list. If the command name contains a */*, then the search path is not used. Otherwise, each directory in the list of directories defined by **PATH** and **FPATH** is checked in order. If the directory being searched is contained in **FPATH** and contains a file whose name matches the command being searched, then this file is loaded into the current shell environment as if it were the argument to the `.` command except that only preset aliases are expanded, and a function of the given name is executed as described above.

If this directory is not in **FPATH** the shell first determines whether there is a built-in version of a command corresponding to a given pathname and if so it is invoked in the current process. If no built-in is found, the

shell checks for a file named **.paths** in this directory. If found and there is a line of the form **FPATH=***path* where *path* names an existing directory then that directory is searched immediately after the current directory as if it were found in the **FPATH** variable. If *path* does not begin with **/**, it is checked for relative to the directory being searched.

The **.paths** file is then checked for a line of the form **PLUGIN_LIB=***libname* [**:** *libname*] Each library named by *libname* will be searched for as if it were an option argument to **builtin -f**, and if it contains a built-in of the specified name this will be executed instead of a command by this name. Any built-in loaded from a library found this way will be associated with the directory containing the **.paths** file so it will only execute if not found in an earlier directory.

Finally, the directory will be checked for a file of the given name. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A separate shell is spawned to read it. All non-exported variables are removed in this case. If the **.paths** contains a line of the form **name=***value* in the first or second line, then the environment variable *name* is modified by prepending the directory specified by *value* to the directory list. If *value* is not an absolute directory, then it specifies a directory relative to the directory that the executable was found. If the environment variable *name* does not already exist it will be added to the environment list for the specified command. A parenthesized command is executed in a subshell without removing non-exported variables.

Command Re-entry.

The text of the last **HISTSIZE** (default 512) commands entered from a terminal device is saved in a *history* file. The file **\$HOME/.sh_history** is used if the **HISTFILE** variable is not set or if the file it names is not writable. A shell can access the commands of all *interactive* shells which use the same named **HISTFILE**. The built-in command **hist** is used to list or edit a portion of this file. The portion of the file to be edited or listed can be selected by number or by giving the first character or characters of the command. A single command or range of commands can be specified. If you do not specify an editor program as an argument to **hist** then the value of the variable **HISTEDIT** is used. If **HISTEDIT** is unset, the obsolete variable **FCEDIT** is used. If **FCEDIT** is not defined, then **/bin/ed** is used. The edited command(s) is printed and re-executed upon leaving the editor unless you quit without writing. The **-s** option (and in obsolete versions, the editor name **-**) is used to skip the editing phase and to re-execute the command. In this case a substitution parameter of the form **old=new** can be used to modify the command before execution. For example, with the preset alias **r**, which is aliased to **'hist -s'**, typing **'r bad=good c'** will re-execute the most recent command which starts with the letter **c**, replacing the first occurrence of the string **bad** with the string **good**.

History Expansion.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix typos in the previous command. The history expansion facility is an alternative to history control via the **fc** or **hist** built-in command. To enable it, turn on the **-H** or **histexpand** option using the **set** command (see *Built-in Commands* below).

History expansions begin with the character **!**. They may begin anywhere in the input. The **!** is passed unchanged with its special meaning disabled when preceded by **** or **\${**, enclosed in single quotes, or followed by a space, tab, newline, **=** or **(**. History expansions do not nest. They are parsed separately before the shell parser is invoked, so they can override shell grammar rules.

By default, the expanded version of any line that contains a history expansion is printed, added to the history, and then immediately executed. History expansions are never added to the history themselves, regardless of whether they succeed or fail due to an error. Normally, this means that a command line with an erroneous history expansion is lost and needs to be retyped from scratch, but if the **histredit** shell option is turned on and a line editor is active (see *In-line Editing Options* below), the erroneous line is pre-filled into the next prompt's input buffer for correcting. The **histverify** option causes the same to be done for the results of successful history expansions, allowing verification and editing before execution.

A history expansion may have an *event specification*, which indicates the event from which words are to be taken, a *word designator*, which selects particular words from the chosen event, and/or a *modifier*, which manipulates the selected words.

An event specification can be:

<i>n</i>	A number, referring to a particular event.
<i>-n</i>	An offset, referring to the event <i>n</i> before the current event.
#	The current event.
!	The previous event (equivalent to -1).
<i>s</i>	The most recent event whose first word begins with the string <i>s</i> .
?s?	The most recent event which contains the string <i>s</i> . The second ? can be omitted if it is immediately followed by a newline.

For example, consider this bit of someone's history list as might be output by the **hist -l** command:

```

9  nroff -man wumpus.man
10 cp wumpus.man wumpus.man.old
11 vi wumpus.man
12 diff wumpus.man.old wumpus.man
```

The commands are shown with their event numbers. The current event, which we haven't typed in yet, is event 13. **!11** and **!-2** refer to event 11. **!!** refers to the previous event, 12. **!!** can be abbreviated **!** if it is followed by **:** (see below). **!n** refers to event 9, which begins with **n**. **!old?** also refers to event 12, which contains **old**. Without word designators or modifiers, history references simply expand to the entire event, so we might type **!cp** to redo the copy command or **!!more** if the **diff** output scrolled off the top of the screen.

To select words from an event, the event specification can be followed by a **:** and a designator for the desired words. The words of an input line are numbered from 0, the first word (usually the command name) being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	The first word (command name).
<i>n</i>	The <i>n</i> th argument.
^	The first argument, equivalent to 1 .
\$	The last argument.
%	The word matched by the most recent ?s? search.
x-y	A range of words.
-y	Equivalent to 0-y .
*	Equivalent to ^-\$, but returns nothing if the event contains only 1 word.
x*	Equivalent to x-\$.
x-	Equivalent to x* , but omitting the last word (\$).

Selected words are inserted into the command line separated by single blanks. For example, the **diff** command in the previous example might have been typed as **diff !!:1.old !!:1** (using **:1** to select the first argument from the previous event) or **diff !-2:2 !-2:1** to select and swap the arguments from the **cp** command. If we didn't care about the order of the **diff**, we might have said **diff !-2:1-2** or simply **diff !-2:***. The **cp** command might have been written **cp wumpus.man !#:1.old**, using **#** to refer to the current event. **!n:-hurkle.man** would reuse the first two words from the **nroff** command to say **nroff -man hurkle.man**.

The **:** separating the event specification from the word designator can be omitted if the argument selector begins with a **^**, **\$**, *****, **%** or **-**. For example, our **diff** command might have been **diff !!^old !!^** or, equivalently, **diff !!\$old !!\$**. However, if **!!** is abbreviated **!**, an argument selector beginning with **-** will be interpreted as an event specification.

The word(s) in a history reference can be edited by following them with one or more modifiers, each preceded by a colon (**:**):

h	Remove a trailing pathname component, leaving the head.
t	Remove all leading pathname components, leaving the tail.
r	Remove a filename extension .xxx , leaving the root name.
e	Remove all but the extension.

s///r/	Substitute <i>l</i> for <i>r</i> . <i>l</i> is simply a string like <i>r</i> , not a regular expression as in the eponymous <i>ed(1)</i> command. Any character may be used as the delimiter in place of <i>/</i> ; a <code>\</code> can be used to quote the delimiter inside <i>l</i> and <i>r</i> . The character <code>&</code> in the <i>r</i> is replaced by <i>l</i> ; <code>\</code> also quotes <code>&</code> . If <i>l</i> is empty, the <i>l</i> from the previous substitution is used, or if there is none, the <i>s</i> from the most recent <code>?s?</code> search. The trailing delimiter may be omitted if it is immediately followed by a newline.
&	Repeat the previous substitution.
g	Global substitution, for example <code>:g/foo/bar/</code> or <code>:g&</code> . Applies the <code>s</code> or <code>&</code> modifier to the entire command line.
a	Same as <code>g</code> .
p	Print the new command line but do not execute it.
q	Quote the expanded words, preventing further expansions.
x	Like <code>q</code> , but break into words at blanks, tabs and newlines.

Modifiers are applied to only the first modifiable word (unless `g` or `a` is used). It is an error for no word to be modifiable.

For example, the `diff` command might have been written as `diff wumpus.man.old !#:r`, using `:r` to remove `.old` from the first argument on the same line (`!#`). We might follow `mail -s "I forgot my password" rot` with `!:s/rot/root` to correct the spelling of `rot`.

History expansions also occur when an input line begins with `^`. When it is the first character on an input line, it is an abbreviation of `!:s^`. Thus we might have said `^rot^root` to make the spelling correction in the previous example. This is the only history expansion that does not explicitly begin with `!`.

If a word on a command line begins with the history comment character `#`, history expansion is ignored for the rest of that line. This usually causes the shell parser (which uses the same character to signal a comment) to treat the rest of the line as a comment as well, but as history expansion is parsed separately from the shell grammar and with different rules, this cannot be guaranteed in all cases. If the history comment character is changed, the shell grammar comment character does not change along with it.

The three characters used to signal history expansion can be changed using the `histchars` shell variable; see *Shell Variables* above.

In-line Editing Options.

Normally, each command line entered from a terminal device is simply typed followed by a **new-line** ('RETURN' or 'LINE FEED'). If either the `emacs`, `gmacs`, or `vi` option is active, the user can edit the command line. To be in either of these edit modes, `set` the corresponding option. An editing option is automatically selected each time the `VISUAL` or `EDITOR` variable is assigned a value matching any of these editor names; for details, see *Shell Variables* above under `VISUAL`.

The editing features require that the user's terminal accept 'RETURN' as carriage return without line feed and that a space (' ') must overwrite the current character on the screen.

Unless the `multiline` option is on, the editing modes implement a concept where the user is looking through a window at the current line. The window width is the value of `COLUMNS` if it is defined, otherwise 80. If the window width is too small to display the prompt and leave at least 8 columns to enter input, the prompt is truncated from the left. If the line is longer than the window width minus two, a mark is displayed at the end of the window to notify the user. As the cursor moves and reaches the window boundaries, the window will be centered about the cursor. The mark is a `>` (`<`, `*`) if the line extends on the right (left, both) side(s) of the window.

The search commands in each edit mode provide access to the history file. Only strings are matched, not patterns, although a leading `^` in the string restricts the match to begin at the first character in the line.

Each of the edit modes has an operation to list the files or commands that match a partially entered word. When applied to the first word on the line, or the first word after a `;`, `|`, `&`, or `(`, and the word does not begin with `~` or contain a `/`, the list of aliases, functions, and executable commands defined by the `PATH` variable that could match the partial word is displayed. Otherwise, the list of files that match the given word is displayed. If the partially entered word does not contain any file expansion characters, a `*` is appended before

generating these lists. After displaying the generated list, the input line is redrawn. These operations are called command name listing and file name listing, respectively. There are additional operations, referred to as command name completion and file name completion, which compute the list of matching commands or files, but instead of printing the list, replace the current word with a complete or partial match. For file name completion, if the match is unique, a */* is appended if the file is a directory and a space is appended if the file is not a directory. Otherwise, the longest common prefix for all the matching files replaces the word. For command name completion, only the portion of the file names after the last */* are used to find the longest command prefix. If only a single name matches this prefix, then the word is replaced with the command name followed by a space. When using a *tab* for completion that does not yield a unique match, a subsequent *tab* will provide a numbered list of matching alternatives. A specific selection can be made by entering the selection number followed by a *tab*. Neither completion nor listing operations are attempted before the first character in a line.

Key Bindings.

The **KEYBD** trap can be used to intercept keys as they are typed and change the characters that are actually seen by the shell. This trap is executed after each character (or sequence of characters when the first character is ESC) is entered while reading from a terminal. The variable **.sh.edchar** contains the character or character sequence which generated the trap. Changing the value of **.sh.edchar** in the trap action causes the shell to behave as if the new value were entered from the keyboard rather than the original value.

The variable **.sh.edcol** is set to the input column number of the cursor at the time of the input. The variable **.sh.edmode** is set to ESC when in **vi** input mode (see below) and set to the empty string otherwise. Prepending **#{.sh.editmode}** to a value assigned to **.sh.edchar** will cause the shell to change to control mode if it is not already in this mode.

This trap is not invoked for characters entered as arguments to editing directives, or while reading input for a character search.

Emacs Editing Mode.

This mode is entered by enabling either the **emacs** or **gmacs** option. The only difference between these two modes is the way they handle **^T**. To edit, the user moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. All the editing commands are control characters or escape sequences. The notation for control characters is caret (^) followed by the character. For example, **^F** is the notation for control F. This is entered by depressing 'f' while holding down the 'CTRL' (control) key. The 'SHIFT' key is *not* depressed. (The notation **^?** indicates the DEL (delete) key.)

The notation for escape sequences is **M-** followed by a character. For example, **M-f** (pronounced Meta f) is entered by depressing ESC (ASCII **033**) followed by 'f'. (**M-F** would be the notation for ESC followed by 'SHIFT' (capital) 'F'.)

All edit commands operate from any place on the line (not just at the beginning). Neither the 'RETURN' nor the 'LINE FEED' key is entered after edit commands except when noted.

The **M-[** multi-character commands below are DEC VT220 escape sequences generated by special keys on standard PC keyboards, such as the arrow keys. You could type them directly but they are meant to recognize the keys in question, which are indicated in parentheses.

^F	Move cursor forward (right) one character.
M-[C	(Right arrow) Same as ^F .
M-f	Move cursor forward one word. (The emacs editor's idea of a word is a string of characters consisting of only letters, digits and underscores.)
M-[1;3C	(Alt-Right arrow) Same as M-f .
M-[1;5C	(Ctrl-Right arrow) Same as M-f .
M-[1;9C	(iTerm2 Alt-Right arrow) Same as M-f .
^B	Move cursor backward (left) one character.
M-[D	(Left arrow) Same as ^B .
M-b	Move cursor backward one word.

M-[1;3D	(Alt-Left arrow) Same as M-b .
M-[1;5D	(Ctrl-Left arrow) Same as M-b .
M-[1;9D	(iTerm2 Alt-Left arrow) Same as M-b .
^A	Move cursor to start of line.
M-[H	(Home) Same as ^A .
M-[1~	Same as ^A .
M-[7~	Same as ^A .
^E	Move cursor to end of line.
M-[F	(End) Same as ^E .
M-[4~	Same as ^E .
M-[8~	Same as ^E .
M-[Y	Same as ^E .
M-OA	(Up Arrow) Same as M-[A .
M-OB	(Down Arrow) Same as M-[B .
M-OC	(Right Arrow) Same as M-[C .
M-OD	(Left Arrow) Same as M-[D .
M-O5C	(Ctrl-Right Arrow) Same as M-f .
M-O5D	(Ctrl-Left Arrow) Same as M-b .
]char	Move cursor forward to character <i>char</i> on current line.
M-^]char	Move cursor backward to character <i>char</i> on current line.
^X^X	Interchange the cursor and mark.
<i>erase</i>	(User defined erase character as defined by the <i>stty(1)</i> command, usually ^H .) Delete previous character.
<i>lnext</i>	(User defined literal next character as defined by the <i>stty(1)</i> command, or ^V if not defined.) Removes the next character's editing features (if any).
^D	Delete current character.
M-[3~	(Forward delete) Same as ^D .
M-d	Delete current word.
M-[3;5~	(Ctrl-Delete) Same as M-d .
M-^H	(Meta-backspace) Delete previous word.
M-h	Delete previous word.
M-^?	(Meta-DEL) Delete previous word (if your interrupt character is ^? (DEL, the default) then this command will not work).
^T	Transpose current character with previous character and advance the cursor in <i>emacs</i> mode. Transpose two previous characters in <i>gmacs</i> mode.
^C	Capitalize current character.
M-c	Capitalize current word.
M-l	Change the current word to lower case.
^K	Delete from the cursor to the end of the line. If preceded by a numerical parameter whose value is less than the current cursor position, then delete from given position up to the cursor. If preceded by a numerical parameter whose value is greater than the current cursor position, then delete from cursor up to given cursor position.
^W	Kill from the cursor to the mark.
M-p	Push the region from the cursor to the mark on the stack.
<i>kill</i>	(User defined kill character as defined by the <i>stty(1)</i> command, usually ^U .) Kill the entire current line. If two <i>kill</i> characters are entered in succession, all kill characters from then on cause a line feed (useful when using paper terminals). A subsequent pair of <i>kill</i> characters undoes this change.
^Y	Restore last item removed from line. (Yank item back to the line.)
^X^E	Return the command hist -e \${VISUAL:-\${EDITOR:-vi}} in the input buffer to call a full editor — vi by default — on the current command line.
^L	Line feed and print current line.

M-^L	Clear the screen.
^@	(Null character) Set mark.
M-space	(Meta space) Set mark.
^J	(New line) Execute the current line.
^M	(Return) Execute the current line.
eof	End-of-file character, normally ^D , is processed as an End-of-file only if the current line is empty.
^P	Fetch previous command. Each time ^P is entered the previous command back in time is accessed. Moves back one line when not on the first line of a multi-line command.
M-[A	(Up arrow) If the cursor is at the end of the line, it is equivalent to ^R with <i>string</i> set to the contents of the current line. Otherwise, it is equivalent to ^P .
M-<	Fetch the least recent (oldest) history line.
M->	Fetch the most recent (youngest) history line.
^N	Fetch next command line. Each time ^N is entered the next command line forward in time is accessed.
M-[B	(Down arrow) Equivalent to ^N .
^Rstring	Reverse search history for a previous command line containing <i>string</i> . If a parameter of zero is given, the search is forward. <i>String</i> is terminated by a 'RETURN' or 'NEW LINE'. If string is preceded by a ^, the matched line must begin with <i>string</i> . If <i>string</i> is omitted, then the next command line containing the most recent <i>string</i> is accessed. In this case a parameter of zero reverses the direction of the search.
^G	Exit reverse search mode.
^O	Operate – Execute the current line and fetch the next line relative to current line from the history file.
M-digits	(Escape) Define numeric parameter, the digits are taken as a parameter to the next command. The commands that accept a parameter are ^F , ^B , <i>erase</i> , ^C , ^D , ^K , ^R , ^P , ^N , ^] , M- , M-^] , M-<u>_</u> , M-= , M-b , M-c , M-d , M-f , M-h , M-l , M-^H , and the arrow keys and forward-delete key.
M-letter	Soft-key – Your alias list is searched for an alias by the name <u>letter</u> and if an alias of this name is defined, its value will be inserted on the input queue. The <i>letter</i> must not be one of the above meta-functions.
M-[letter	Soft-key – Your alias list is searched for an alias by the name <u>letter</u> and if an alias of this name is defined, its value will be inserted on the input queue. This can be used to program function keys on many terminals.
M-	The last word of the previous command is inserted on the line. If preceded by a numeric parameter, the value of this parameter determines which word to insert rather than the last word.
M-<u>_</u>	Same as M- .
M-*	Attempt pathname expansion on the current word. An asterisk is appended if the word doesn't match any file or contain any special pattern characters.
M-ESC	Command or file name completion as described above.
^I tab	Attempts command or file name completion as described above. If a partial completion occurs, repeating this will behave as if M-= were entered. If no match is found or entered after <i>space</i> , a <i>tab</i> is inserted.
M-=	If not preceded by a numeric parameter, it generates the list of matching commands or file names as described above. Otherwise, the word under the cursor is replaced by the item corresponding to the value of the numeric parameter from the most recently generated command or file list. If the cursor is not on a word, it is inserted instead.
^U	Multiply parameter of next command by 4.
\	If the backslashctrl shell option is on (which is the default setting), this escapes the next character. Editing characters, the user's erase, kill and interrupt (normally ^C) characters may be entered in a command line or in a search string if preceded by a \. The \ removes the next character's editing features (if any). See also <i>lnext</i> which is not subject to any shell option.
M-^V	Display version of the shell.

- M-[2~** (Insert) Escape the next character.
- M-#** If the line does not begin with a #, a # is inserted at the beginning of the line and after each new-line, and the line is entered. This causes a comment to be inserted in the history file. If the line begins with a #, the # is deleted and one # after each new-line is also deleted.

Vi Editing Mode.

There are two typing modes. Initially, when you enter a command you are in the *input* mode. To edit, the user enters *control* mode by typing ESC (**033**) and moves the cursor to the point needing correction and then inserts or deletes characters or words as needed. Most control commands accept an optional repeat *count* prior to the command.

The notation for control characters used below is ^ followed by a character. For instance, ^H is entered by holding down the Control key and pressing H. ^[(Control+[) is equivalent to the ESC key. The notation for escape sequences is ^[followed by one or more characters.

The ^[[(ESC [) multi-character commands below are DEC VT220 escape sequences generated by special keys on standard PC keyboards, such as the arrow keys, which are indicated in parentheses. When in *input* mode, these keys will switch you to *control* mode before performing the associated action. These sequences can use preceding repeat count parameters, but only when the ^[and the subsequent [are entered into the input buffer at the same time, such as when pressing one of those keys.

Input Edit Commands

By default the editor is in input mode.

- erase* (User defined erase character as defined by the *stty*(1) command, usually ^H or #.) Delete previous character.
- ^W Delete the previous blank separated word.
- eof* As the first character of the line causes the shell to terminate unless the **ignoreeof** option is set. Otherwise this character is ignored.
- lnext* (User defined literal next character as defined by the *stty*(1) or ^V if not defined.) Removes the next character's editing features (if any).
- \ If the **backslashctrl** shell option is on (which is the default setting), this escapes the next *erase* or *kill* character.
- ^I *tab* Attempts command or file name completion as described above and returns to input mode. If a partial completion occurs, repeating this will behave as if = were entered from control mode. If no match is found or entered after *space*, a *tab* is inserted.

Motion Edit Commands

These commands will move the cursor.

- [*count*]I Cursor forward (right) one character.
- [*count*]^[C (Right arrow) Same as I.
- [*count*]w Cursor forward one alphanumeric word.
- [*count*]W Cursor to the beginning of the next word that follows a blank.
- [*count*]e Cursor to end of word.
- [*count*]E Cursor to end of the current blank delimited word.
- [*count*]h Cursor backward (left) one character.
- [*count*]^[D (Left arrow) Same as h.
- [*count*]b Cursor backward one word.
- [*count*]B Cursor to preceding blank separated word.
- [*count*]| Cursor to column *count*.
- [*count*]fc Find the next character *c* in the current line.
- [*count*]Fc Find the previous character *c* in the current line.
- [*count*]tc Equivalent to f followed by h.
- [*count*]Tc Equivalent to F followed by I.

[count];	Repeats <i>count</i> times, the last single character find command, f , F , t , or T .
[count],	Reverses the last single character find command <i>count</i> times.
0	Cursor to start of line.
^[H	(Home) Same as 0 .
^[I~	Same as 0 .
^[7~	Same as 0 .
^[1;3D	(Alt-Left arrow) Same as b .
^[1;5D	(Ctrl-Left arrow) Same as b .
^[1;9D	(iTerm2 Alt-Left arrow) Same as b .
^[1;3C	(Alt-Right arrow) Same as w .
^[1;5C	(Ctrl-Right arrow) Same as w .
^[1;9C	(iTerm2 Alt-Right arrow) Same as w .
^[2~	(Insert) Same as i .
^[3;5~	(Ctrl-Delete) Same as dw .
^[OA	(Up Arrow) Same as ^[A .
^[OB	(Down Arrow) Same as ^[B .
^[OC	(Right Arrow) Same as ^[C .
^[OD	(Left Arrow) Same as ^[D .
^[O5C	(Ctrl-Right Arrow) Same as w .
^[O5D	(Ctrl-Left Arrow) Same as b .
^	Cursor to first non-blank character in line.
\$	Cursor to end of line.
^[F	(End) Same as \$.
^[4~	Same as \$.
^[8~	Same as \$.
^[Y	Same as \$.
^G	Exit reverse search mode.
%	Moves to balancing (,) , { , } , [, or] . If cursor is not on one of the above characters, the remainder of the line is searched for the first occurrence of one of the above characters first.

Search Edit Commands

These commands access your command history.

[count]k	Fetch previous command. Each time k is entered the previous command back in time is accessed.
[count]-	Equivalent to k .
[count]^[A	(Up arrow) If cursor is at the end of the line it is equivalent to / with <i>string</i> set to the contents of the current line. Otherwise, it is equivalent to k .
[count]j	Fetch next command. Each time j is entered the next command forward in time is accessed.
[count]+	Equivalent to j .
[count]^[B	(Down arrow) Equivalent to j .
[count]G	The command number <i>count</i> is fetched. The default is the least recent history command.
/string	Search backward through history for a previous command containing <i>string</i> . <i>String</i> is terminated by a 'RETURN' or 'NEW LINE'. If <i>string</i> is preceded by a ^ , the matched line must begin with <i>string</i> . If <i>string</i> is empty, the previous string will be used.
?string	Same as / except that search will be in the forward direction.
n	Search for next match of the last pattern to / or ? commands.
N	Search for next match of the last pattern to / or ? , but in reverse direction.

Text Modification Edit Commands

These commands will modify the line.

- a** Enter input mode and enter text after the current character.
- A** Append text to the end of the line. Equivalent to **\$a**.
- [count]c***motion*
c*[count]motion* Delete current character through the character that *motion* would move the cursor to and enter input mode. If *motion* is **c**, the entire line will be deleted and input mode entered.
- C** Delete the current character through the end of line and enter input mode. Equivalent to **c\$**.
- S** Equivalent to **cc**.
- [count]s** Replace characters under the cursor in input mode.
- D** Delete the current character through the end of line. Equivalent to **d\$**.
- [count]d***motion*
d*[count]motion* Delete current character through the character that *motion* would move to. If *motion* is **d**, the entire line will be deleted.
- i** Enter input mode and insert text before the current character.
- I** Insert text before the first non-blank character. Equivalent to **^i**.
- [count]P** Place the previous text modification before the cursor.
- [count]p** Place the previous text modification after the cursor.
- R** Enter input mode and replace characters on the screen with characters you type overlay fashion.
- [count]rc** Replace the *count* character(s) starting at the current cursor position with *c*, and advance the cursor.
- [count]x** Delete current character.
- [count]^[[3~**
 (Forward delete) Same as **x**.
- [count]X** Delete preceding character.
- [count].** Repeat the previous text modification command.
- [count]~** Invert the case of the *count* character(s) starting at the current cursor position and advance the cursor.
- [count]_** Causes the *count* word of the previous command to be appended and input mode entered. The last word is used if *count* is omitted.
- *** Causes an ***** to be appended to the current word and pathname expansion attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.
- ** Command or file name completion as described above.

Other Edit Commands

Miscellaneous commands.

- [count]y***motion*
y*[count]motion* Yank current character through character that *motion* would move the cursor to and puts them into the delete buffer. The text and cursor are unchanged.
- yy** Yanks the entire line.
- Y** Yanks from current position to end of line. Equivalent to **y\$**.
- u** Undo the last text modifying command.
- U** Undo all the text modifying commands performed on the line.
- [count]v** Returns the command **hist -e \${VISUAL:-\${EDITOR:-vi}}** *count* in the input buffer to call a full editor — **vi** by default — on a history entry. If *count* is omitted, then the current line is used.

^L	Line feed and print current line. Has effect only in control mode.
^J	(New line) Execute the current line, regardless of mode.
^M	(Return) Execute the current line, regardless of mode.
#	If the first character of the command is a #, then this command deletes this # and each # that follows a newline. Otherwise, sends the line after inserting a # in front of each line in the command. Useful for causing the current line to be inserted in the history as a comment and uncommenting previously commented commands in the history file.
[count]=	If <i>count</i> is not specified, it generates the list of matching commands or file names as described above. Otherwise, the word under the cursor is replaced by the <i>count</i> item from the most recently generated command or file list. If the cursor is not on a word, it is inserted instead.
@letter	Your alias list is searched for an alias by the name <i>_letter</i> and if an alias of this name is defined, its value will be inserted on the input queue for processing.
^V	Display version of the shell.

Built-in Commands.

The *simple-commands* listed below are built in to the shell and are executed in the same process as the shell. The effects of any added Input/Output redirections are local to the command, except for the **exec** and **redirect** commands. Unless otherwise indicated, the output is written on standard output (file descriptor 1) and the exit status, when there is no syntax error, is zero. Except for **:**, **true**, **false**, and **echo**, all built-in commands accept **--** to indicate end of options, and are self-documenting.

The self-documenting commands interpret the option **--man** as a request to display that command's own manual page, **--help** as a request to display the *OPTIONS* section from their manual page, and **-?** as a request to print a brief usage message. All these are processed as error messages, so they are written on standard error (file descriptor 2) and to pipe them into a pager such as *more(1)* you need to add a **2>&1** redirection before the |. The display of boldface text depends on whether standard error is on a terminal, so is disabled when using a pager. Exporting the **ERROR_OPTIONS** environment variable with a value containing *emphasis* will force this on; a value containing *noemphasis* forces it off. The **test/[** command needs an additional **--** argument to recognize self-documentation options, e.g. **test --man --**. The **exec** and **redirect** commands, as they make redirections permanent, should use self-documentation options in a subshell when redirecting, for example: **(redirect --man) 2>&1**. There are advanced output options as well; see **getopts --man** for more information.

Commands that are preceded by a † symbol below are *special built-in commands* and are treated specially in the following ways:

1. Variable assignment lists preceding the command remain in effect when the command completes.
2. I/O redirections are processed after variable assignments.
3. Errors cause a script that contains them to abort.
4. They are not valid function names.

Commands that are preceded by a ‡ symbol below are *declaration commands*. Any following words that are in the format of a variable assignment are expanded with the same rules as a variable assignment. This means that tilde expansion is performed after the = sign, array assignments of the form *varname=(assignment_list)* are supported, and field splitting and pathname expansion are not performed.

† : [*arg* ...]

The command only expands parameters.

† .*name* [*arg* ...]

If *name* is a function defined with the **function name** reserved word syntax, the function is executed in the current environment (as if it had been defined with the *name()* syntax). Otherwise if *name* refers to a file, the file is read in its entirety and the commands are executed in the current shell environment. The search path specified by **PATH** is used to find the directory containing the file. If any arguments *arg* are given, they become the positional parameters while processing the . command and the original positional parameters are restored upon completion. Otherwise the

positional parameters are unchanged. The exit status is the exit status of the last command executed.

[*expression*]

The [*command*] is the same as **test**, with the exception that an additional closing] argument is required. See **test** below.

alias [**-ptx**] [*name*[=*value*]] ...

alias with no arguments prints the list of aliases in the form *name=value* on standard output. The **-p** option causes the word **alias** to be inserted before each one. When one or more arguments are given, an *alias* is defined for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution. With the **-t** option, **alias** acts like **hash**; see there. Without the **-t** option, for each *name* in the argument list for which no *value* is given, the name and value of the alias is printed. The obsolete **-x** option has no effect in most contexts, although if it's used with **-t** it will suppress all output. The exit status is non-zero if a *name* is given, but no value, and no alias has been defined for the *name*.

autoload *name* ...

Marks each *name* undefined so that the **FPATH** variable will be searched to find the function definition when the function is referenced. The same as **typeset -fu**.

bg [*job* ...]

This command is only on systems that support job control. Puts each specified *job* into the background. The current job is put in the background if *job* is not specified. See *Jobs* for a description of the format of *job*.

† **break** [*n*]

Exit from the enclosing **for**, **while**, **until**, or **select** loop, if any. If *n* is specified, then break *n* levels.

builtin [**-ds**] [**-f***file*] [*name* ...]

If *name* is not specified, and no **-f** option is specified, the built-ins are printed on standard output. The **-s** option prints only the special built-ins. Otherwise, each *name* represents the pathname whose basename is the name of the built-in. The entry point function name is determined by prepending **b_** to the built-in name. A built-in specified by a pathname will only be executed when that pathname would be found during the path search. Built-ins found in libraries loaded via the **.paths** file will associate with the pathname of the directory containing the **.paths** file.

The ISO C/C++ prototype is **b_mycommand(int argc, char *argv[], void *context)** for the builtin command *mycommand* where *argv* is array an of *argc* elements and context is an optional pointer to a **Shell_t** structure as described in **<ast/shell.h>**.

Special built-ins cannot be bound to a pathname or deleted. The **-d** option deletes each of the given built-ins. On systems that support dynamic loading, the **-f** option names a shared library containing the code for built-ins. The shared library prefix and/or suffix, which depend on the system, can be omitted. Once a library is loaded, its symbols become available for subsequent invocations of **builtin**. Multiple libraries can be specified with separate invocations of the **builtin** command. Libraries are searched in the reverse order in which they are specified. When a library is loaded, it looks for a function in the library whose name is **lib_init()** and invokes this function with an argument of **0**.

cd [**-L**] [**-eP**] [*arg*]

cd [**-L**] [**-eP**] *old new*

This command can be in either of two forms. In the first form it changes the current directory to *arg*. If *arg* is **-** the directory is changed to the previous directory. The shell variable **HOME** is the default *arg*. The variable **PWD** is set to the current directory. The shell variable **CDPATH** defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (:). The default path is the empty string (specifying the current directory). Note that the current directory may be specified by a dot (.) or by an empty path name, either of which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path

list. If *arg* begins with a / then the search path is not used. Otherwise, each directory in the path is searched for *arg*.

The second form of **cd** substitutes the string *new* for the string *old* in the current directory name, **PWD**, and tries to change to this new directory.

By default, symbolic link names are treated literally when finding the directory name. This is equivalent to the **-L** option. The **-P** option causes symbolic links to be resolved when determining the directory. The last instance of **-L** or **-P** on the command line determines which method is used.

If **-e** and **-P** are both in effect and the correct **PWD** could not be determined after successfully changing the directory, **cd** will return with exit status one and produce no output. If any other error occurs while both flags are active, the exit status is greater than one.

The **cd** command may not be executed by **rksh**.

command [**-pvxV**] *name* [*arg* ...]

With the **-v** option, **command** is equivalent to the built-in **whence** command described below. The **-V** option causes **command** to act like **whence -v**.

Without the **-v** or **-V** options, **command** executes *name* with the arguments given by *arg*. Functions and aliases will not be searched for when finding *name*. If *name* refers to a special built-in, as marked with † in this manual, **command** disables the special properties described above for that mark, executing the command as a regular built-in. (For example, using **command set -o option-name** prevents a script from terminating when an invalid option name is given.)

The **-p** option causes the operating system's standard utilities path (as output by **getconf PATH**) to be searched rather than the one defined by the value of **PATH**.

The **-x** option searches for *name* as an external command, bypassing built-ins. If the arguments contain at least one word that expands to multiple arguments, for example ***.txt** or **"\$@"**, then the **-x** option also allows executing external commands with argument lists that are longer than the operating system allows. This functionality is similar to **xargs(1)** but is easier to use. The shell does this by invoking the external command multiple times if needed, dividing the expanded argument list over the invocations. Any arguments that come before the first word that expands to multiple arguments, as well as any that follow the last such word, are repeated for each invocation. This allows each invocation to use the same command options, as well as the same trailing destination arguments for commands like **cp(1)** or **mv(1)**. When all invocations are completed, **command -x** exits with the status of the invocation that had the highest exit status. (Note that **command -x** may still fail with an "argument list too long" error if a single argument exceeds the maximum length of the argument list, or if a long arguments list contains no word that expands to multiple arguments.)

‡ **compound** *vname*[=*value*] ...

Causes each *vname* to be a compound variable. The same as **typeset -C**.

† **continue** [*n*]

Resume the next iteration of the enclosing **for**, **while**, **until**, or **select** loop. If *n* is specified, then resume at the *n*-th enclosing loop.

disown [*job*...]

Causes the shell not to send a HUP signal to each given *job*, or all active jobs if *job* is omitted, when a login shell terminates.

echo [*arg* ...]

When the first *arg* does not begin with a -, and none of the arguments contain a \, then **echo** prints each of its arguments separated by a space and terminated by a new-line. Otherwise, the behavior of **echo** is system dependent and **print** or **printf** described below should be used. See **echo(1)** for usage and description.

‡ **enum** [**-i**] *type*[*=(value ...)*] ...

Creates, for each *type* specified, an enumeration type declaration command named *type*. Variables of the created type can only store any one of the *values* given. For example, **enum bool=(false true)** creates a Boolean variable type of which variables may be declared like **bool x=true y=false**. If *=(value ...)* is omitted, then *type* must be an indexed array variable with at least two elements and the values are taken from this array variable. If **-i** is specified the values are case-insensitive. Declaration commands are created as special builtins that cannot be removed or overridden by shell functions. Each created declaration command has a **--man** option that shows documentation on its type's behavior and possible values.

Within arithmetic expressions (see *Arithmetic Evaluation* above), enumeration type values translate to index numbers between 0 and the number of defined values minus 1. It is an error for an arithmetic expression to assign a value outside of that range. Decimal fractions are ignored. Taking the **bool** type from the example above, if a variable of this type is used in an arithmetic expression, **false** translates to 0 and **true** to 1. Enumeration values may also be used directly in an arithmetic expression that refers to a variable of an enumeration type. To continue our example, for a **bool** variable **v**, **((v==true))** is the same as **((v==1))** and if a variable named **true** exists, it is ignored.

† **eval** [*arg ...*]

The arguments are read as input to the shell and the resulting command(s) executed.

† **exec** [**-c**] [**-a name**] [*arg ...*]

If *arg* is given, the command specified by the arguments is searched on **PATH** as an external command and executed in place of the current shell without creating a new process. The value of the **SHLVL** environment variable is decreased by one, unless the shell replaced is a subshell. The **-c** option causes the environment to be cleared before applying variable assignments associated with the **exec** invocation. The **-a** option causes *name* rather than the first *arg*, to become **argv[0]** for the new process. If *arg* is not given and only I/O redirections are given, then this command persistently modifies file descriptors as in **redirect**.

† **exit** [*n*]

Causes the shell to exit with the exit status specified by *n*. The value will be the least significant 8 bits of *n* (if specified) or of the exit status of the last command executed. An end-of-file will also cause the shell to exit, except for an interactive shell that has the **ignoreeof** option turned on (see **set** below).

‡ **export** [**-p**] [*name*[*=value*]] ...

If *name* is not given, the names and values of each variable with the export attribute are printed with the values quoted in a manner that allows them to be re-input. The **export** command is the same as **typeset -x** except that if you use **export** within a function, no local variable is created. The **-p** option causes the word **export** to be inserted before each one. Otherwise, the given *names* are marked for automatic export to the *environment* of subsequently-executed commands.

false Does nothing, and exits 1. Used with **until** for infinite loops.

fc [**-e ename**] [**-N num**] [**-nlr**] [*first* [*last*]]

fc -s [*old=new*] [*command*]

The same as **hist**.

fg [*job...*]

This command is only on systems that support job control. Each *job* specified is brought to the foreground and waited for in the specified order. Otherwise, the current job is brought into the foreground. See *Jobs* for a description of the format of *job*.

‡ **float** *vname*[*=value*] ...

Declares each *vname* to be a long floating point number. The same as **typeset -IE**.

functions [**-Stux**] [*name* ...]

Lists functions. The same as **typeset -f**.

getconf [*name* [*pathname*]]

Prints the current value of the configuration parameter given by *name*. The configuration parameters are defined by the IEEE POSIX 1003.1 and IEEE POSIX 1003.2 standards. (See *pathconf*(2) and *sysconf*(3).) The *pathname* argument is required for parameters whose value depends on the location in the file system. If no arguments are given, **getconf** prints the names and values of the current configuration parameters. The *pathname* / is used for each of the parameters that requires *pathname*.

getopts [**-a name**] *optstring* *vname* [*arg* ...]

Checks *arg* for legal options. If *arg* is omitted, the positional parameters are used. An option argument begins with a + or a -. An option not beginning with + or - or the argument -- ends the options. Options beginning with + are only recognized when *optstring* begins with a +. *optstring* contains the letters that **getopts** recognizes. If a letter is followed by a :, that option is expected to have an argument. The options can be separated from the argument by blanks. The option **-?** causes **getopts** to generate a usage message on standard error. The **-a** argument can be used to specify the name to use for the usage message, which defaults to **\$0**.

getopts places the next option letter it finds inside variable *vname* each time it is invoked. The option letter will be prepended with a + when *arg* begins with a +. The index of the next *arg* is stored in **OPTIND**. The option argument, if any, gets stored in **OPTARG**.

A leading : in *optstring* causes **getopts** to store the letter of an invalid option in **OPTARG**, and to set *vname* to ? for an unknown option and to : when a required option argument is missing. Otherwise, **getopts** prints an error message. The exit status is non-zero when there are no more options.

There is no way to specify any of the options :, +, -, ?, [, and]. The option # can only be specified as the first option.

hash [**-r**] [*utility* ...]

hash displays or modifies the hash table with the locations of recently used programs. If given no arguments, it lists all command/path associations (a.k.a. 'tracked aliases') in the hash table. Otherwise, **hash** performs a **PATH** search for each *utility* supplied and adds the result to the hash table. Any *utility* that is not found is silently ignored. The **-r** option empties the hash table. This can also be achieved by resetting **PATH**.

hist [**-e** *ename*] [**-N** *num*] [**-nlr**] [*first* [*last*]]

hist -s [*old=new*] [*command*]

In the first form, a range of commands from *first* to *last* is selected from the last **HISTSIZE** commands that were typed at the terminal. The arguments *first* and *last* may be specified as a number or as a string. A string is used to locate the most recent command starting with the given string. A negative number is used as an offset to the current command number. If the **-l** option is selected, the commands are listed on standard output. Otherwise, the editor program *ename* is invoked on a file containing these keyboard commands. If *ename* is not supplied, then the value of the variable **HISTEDIT** is used. If **HISTEDIT** is not set, then **FCEDIT** (default **/bin/ed**) is used as the editor. When editing is complete, the edited command(s) is executed if the changes have been saved. If *last* is not specified, then it will be set to *first*. If *first* is not specified, the default is the previous command for editing and -16 for listing. The option **-r** reverses the order of the commands and the option **-n** suppresses command numbers when listing. In the second form, *command* is interpreted as *first* described above and defaults to the last command executed. The resulting command is executed after the optional substitution *old=new* is performed. The option **-N** causes **hist** to start *num* commands back.

‡ **integer** *vname*[=*value*] ...

Declares each *vname* to be a long integer number. The same as **typeset -li**.

jobs [**-lnp**] [*job* ...]

Lists information about each given job; or all active jobs if *job* is omitted. The **-l** option lists process IDs in addition to the normal information. The **-n** option only displays jobs that have stopped or exited since last notified. The **-p** option causes only the process group to be listed. See *Jobs* for a description of the format of *job*.

kill [**-s** *signame*] *job* ...

kill [**-n** *signum*] *job* ...

kill -Ll [*sig* ...]

Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number with the **-n** option or by name with the **-s** option (as given in **<signal.h>**, stripped of the prefix “SIG”). For backward compatibility, the **n** and **s** can be omitted and the number or name placed immediately after the **-**. If the signal being sent is TERM (terminate) or HUP (hangup), and the **posix** shell option is off, then the job or process will be sent a CONT (continue) signal if it is stopped. The argument *job* can be the process ID of a process that is not a member of one of the active jobs. See *Jobs* for a description of the format of *job*. In the third form, **kill -l** or **kill -L**, if *sig* is not specified, the signal names are listed. The **-l** option lists only the signal names. The **-L** option lists each signal name and corresponding number. Otherwise, for each *sig* that is a name, the corresponding signal number is listed. For each *sig* that is a number, the signal name corresponding to the least significant 8 bits of *sig* is listed.

let *arg* ...

Each *arg* is a separate *arithmetic expression* to be evaluated. **let** only recognizes octal numbers starting with **0** when the **set** option **letoctal** is on. See *Arithmetic Evaluation* above for a description of arithmetic expression evaluation.

The exit status is 0 if the value of the last expression is non-zero, and 1 otherwise.

‡ **nameref** *vname*[=*refname*] ...

Declares each *vname* to be a variable name reference. The same as **typeset -n**.

print [**-C***R***enprsv**] [**-u** *unit*] [**-f** *format*] [*arg* ...]

With no options or with option **-** or **--**, each *arg* is printed on standard output. The **-f** option causes the arguments to be printed as described by **printf**. In this case, any **e**, **n**, **r**, **R** options are ignored. Otherwise, unless the **-C**, **-R**, **-r**, or **-v** are specified, the following escape conventions will be applied:

\a	The alert character (ASCII 07).
\b	The backspace character (ASCII 010).
\c	Causes print to end without processing more arguments and not adding a new-line.
\f	The formfeed character (ASCII 014).
\n	The newline character (ASCII 012).
\r	The carriage return character (ASCII 015).
\t	The tab character (ASCII 011).
\v	The vertical tab character (ASCII 013).
\E	The escape character (ASCII 033).
\	The backslash character \ .
\0x	The character defined by the 1, 2, or 3-digit octal string given by <i>x</i> .

The **-R** option will print all subsequent arguments and options other than **-n**. The **-e** causes the above escape conventions to be applied. This is the default behavior. It reverses the effect of an earlier **-r**. The **-p** option causes the arguments to be written onto the pipe of the process spawned with **|&** instead of standard output. The **-v** option treats each *arg* as a variable name and writes the value in the **printf %B** format. The **-C** option treats each *arg* as a variable name and writes the value in the **printf %#B** format. The **-s** option causes the arguments to be written onto the history file instead of standard output. The **-u** option can be used to specify a one digit file descriptor unit number *unit* on which the output will be placed. The default is 1. If the option **-n** is used, no **new-line** is added to the output.

printf [**-v** *vname*] *format* [*arg* ...]

The arguments *arg* are printed on standard output in accordance with the ANSI C formatting rules associated with the format string *format*. If the number of arguments exceeds the number of format specifications, the **format** string is reused to format remaining arguments. The following extensions can also be used:

- %b** A **%b** format can be used instead of **%s** to cause escape sequences in the corresponding *arg* to be expanded as described in **print**.
- %B** A **%B** option causes each of the arguments to be treated as variable names and the binary value of variable will be printed. The alternate flag **#** causes a compound variable to be output on a single line. This is most useful for compound variables and variables whose attribute is **-b**.
- %H** A **%H** format can be used instead of **%s** to cause characters in *arg* that are special in HTML and XML to be output as their entity name. The alternate flag **#** formats the output for use as a URI.
- %p** A **%p** format will convert the given number to hexadecimal.
- %P** A **%P** format can be used instead of **%s** to cause *arg* to be interpreted as an extended regular expression and be printed as a shell pattern.
- %q** A **%q** format can be used instead of **%s** to cause the resulting string to be quoted in a manner than can be reinput to the shell. When **q** is preceded by the alternative format specifier, **#**, the string is quoted in manner suitable as a field in a **.csv** format file.
- %(date-format)T**
A **%(date-format)T** format can be used to treat an argument as a date/time string and to format the date/time according to the *date-format*.
- %Q** A **%Q** format will convert the given number of seconds to readable time.
- %R** A **%R** format can be used instead of **%s** to cause *arg* to be interpreted as a shell pattern and to be printed as an extended regular expression.
- %Z** A **%Z** format will output a byte whose value is 0.
- %d** The precision field of the **%d** format can be followed by a **.** and the output base. In this case, the **#** flag character causes *base#* to be prepended. When an output base is specified without giving a precision (e.g. **%.2d**), the precision defaults to 1 instead of 0.
- #** The **#** flag, when used with the **%d** format without an output base, displays the output in powers of 1000 indicated by one of the following suffixes: **k M G T P E**, and when used with the **%i** format displays the output in powers of 1024 indicated by one of the following suffixes: **Ki Mi Gi Ti Pi Ei**.
- =** The **=** flag centers the output within the specified field width.
- L** The **L** flag, when used with the **%c** or **%s** formats, treats precision as character width instead of byte count.
- ,** The **,** flag, when used with the **%d** or **%f** formats, separates groups of digits with the grouping delimiter (**,** on groups of 3 in the C locale).

The **-v** option assigns the output directly to a variable instead of writing it to standard output. This is faster than capturing the output using a command substitution and avoids the latter's stripping of final linefeed characters (**\n**). The *vname* argument should be a valid variable name, optionally with one or more array subscripts in square brackets. Note that square brackets should be quoted to avoid pathname expansion.

On some systems, the external **printf(1)** command allows the *format* operand to begin with a **-** without any preceding **--** option terminator argument for compatibility with ancient scripts, provided no options are given. On such systems, **ksh**'s built-in **printf** may have been built to be as compatible with this as possible while still allowing its options to be usable. However, the POSIX standard requires adding the preceding **--** to keep such a *format* operand from being misinterpreted as options. The obsolete syntax is not portable and should be avoided in new scripts.

pwd [**-LP**]

Outputs the value of the current working directory. The **-L** option is the default; it prints the logical name of the current directory. If the **-P** option is given, all symbolic links are resolved from

the name. The last instance of **-L** or **-P** on the command line determines which method is used.

```
read [ -ACSaprsv ] [ -d delim ] [ -n n ] [ -N n ] [ -t timeout ] [ -u unit ] [ vname?prompt ] [ vname ... ]
```

The shell input mechanism. One line is read and is broken up into fields using the characters in **IFS** as separators. The escape character, ****, is used to remove any special meaning for the next character and for line continuation. The first field is assigned to the first *vname*, the second field to the second *vname*, etc., with leftover fields assigned to the last *vname*. If *vname* is omitted, then **REPLY** is used as the default *vname*. When *vname* has the binary attribute and **-n** or **-N** is specified, the bytes that are read are stored directly into the variable. If you append *prompt* to the first *vname*, then **read** will display *prompt* on standard error before reading if standard input is a terminal or pipe; the **?** should be quoted to protect it from pathname expansion. The exit status is 0 unless an end-of-file is encountered or **read** has timed out. The options for the **read** command have meaning as follows:

- A** Causes the variable *vname* to be unset and each field that is read to be stored in successive elements of the indexed array *vname*.
- C** Causes the variable *vname* to be read as a compound variable. Blanks will be ignored when finding the beginning open parenthesis.
- N** Causes *n* bytes to be read unless an end-of-file has been encountered or the read times out because of the **-t** option.
- S** Causes the line to be treated like a record in a **.csv** format file so that double quotes can be used to allow the delimiter character and the new-line character to appear within a field.
- a** Same as **-A**.
- d** Causes the read to continue to the first character of *delim* instead of the newline control character. Multibyte characters for *delim* are not supported.
- n** Causes at most *n* bytes to be read instead of a full line, but will return when reading from a slow device as soon as any characters have been read.
- p** Input is read from the current co-process spawned by the shell using **|&**. An end-of-file causes **read** to disconnect the co-process so that another can be created.
- r** Raw mode. The **** character is not treated specially.
- s** The input will be saved as a command in the history file.
- t** Used to specify a timeout in seconds when reading from a terminal or pipe.
- u** This option can be used to specify a one-digit file descriptor unit *unit* to read from. The file descriptor can be opened with the **exec** or **redirect** built-in command. If *unit* is **p**, input is read from the current co-process as with the **-p** option. The default value of *unit* is 0.
- v** The value of the first *vname* will be used as a default value when reading from a terminal device.

```
†‡ readonly [ -p ] [ vname[=value] ] ...
```

If *vname* is not given, the names and values of each variable with the read-only attribute is printed with the values quoted in a manner that allows them to be re-input. The **-p** option causes the word **readonly** to be inserted before each one. Otherwise, the given *vnames* are marked read-only and these names cannot be changed by subsequent assignment. Unlike **typeset -r**, **readonly** does not create a function-local scope and the given *vnames* are marked globally read-only by default. When defining a type, if the value of a read-only subvariable is not defined, the value is required when creating each instance.

redirect

This command only accepts input/output redirections. It can open and close files and modify file descriptors from **0** to **9** as specified by the input/output redirection list (see the *Input/Output* section above), with the difference that the effect persists past the execution of the **redirect** command. When invoking another program, file descriptors greater than **2** that were opened with this mechanism are only passed on if they are explicitly redirected to themselves as part of the invocation

(e.g. **4>&4**) or if the **posix** option is set.

† **return** [*n*]

Causes a shell function, dot script (see **.** and **source**), or profile script to return to the invoking shell environment with the exit status specified by *n*. This status value can use the full signed integer range as shown by the commands **getconf INT_MIN** and **getconf INT_MAX**. A value outside that range will produce a warning and an exit status of 128. If *n* is omitted, then the value of **\$_?** is assumed, i.e., the exit status of the last command executed is passed on. If **return** is invoked while not in a function, dot script, or profile script, then it behaves the same as **exit**.

† **set** [**±BCGHabefhkmnprstuvx**] [**±o** [*option*]] ... [**±A** *vname*] [*arg* ...]

The options for this command have meaning as follows:

- A** Array assignment. Unset the variable *vname* and assign values sequentially from the *arg* list. If **+A** is used, the variable *vname* is not unset first.
- B** Enable brace group expansion. On by default, except if ksh is invoked as **sh** or **rsh**.
- C** Prevents redirection **>** from truncating existing files. Files that are created are opened with the **O_EXCL** mode. Requires **>|** to truncate a file when turned on.
- G** Enables recursive pathname expansion. This adds the double-star pattern ****** to the pathname expansion (see *Pathname Expansion* above). By itself, it matches the recursive contents of the current directory, which is to say, all files and directories in the current directory and in all its subdirectories, sub-subdirectories, and so on. If the pathname pattern ends in ****/**, only directories and subdirectories are matched, including symbolic links that point to directories. A prefixed directory name is not included in the results unless that directory was itself found by a pattern. For example, **dir/**** matches the recursive contents of **dir** but not **dir** itself, whereas **di[r]**** matches both **dir** itself and the recursive contents of **dir**. Symbolic links to non-directories are not followed. Symbolic links to directories are followed if they are specified literally or match a pattern as described under *Pathname Expansion*, but not if they result from a double-star pattern.
- H** Enable **!**-style history expansion similar to *csh*(1). See *History Expansion* above.
- a** All variables that are assigned a value while this option is on are automatically exported, unless they have a dot in their name. Variables created in namespaces declared with the **namespace** keyword (see *Name Spaces* above) are only exported while their name space is active.
- b** Prints job completion messages as soon as a background job changes state rather than waiting for the next prompt. If one of the shell line editors is in use (see *In-line Editing Options* above), the completion message is inserted directly above the command line being typed.
- e** Unless contained in a **||** or **&&** command, or the command following an **if while** or **until** command or in the pipeline following **!**, if a command has a non-zero exit status, execute the **ERR** trap, if set, and exit. This mode is disabled while reading profiles.
- f** Disables pathname expansion.
- h** Obsolete; no effect.
- k** All variable assignment arguments are placed in the environment for a command, not just those that precede the command name.
- m** Background jobs will run in a separate process group and a line will print upon completion. The exit status of background jobs is reported in a completion message. A pipeline will not terminate until all component commands of the pipeline have terminated. On systems with job control, this option is turned on automatically for interactive shells.
- n** Read commands and check them for syntax errors, but do not execute them. Ignored for interactive shells.
- o** The following argument can be one of the following option names:

allexport

Same as **-a**.

backslashctrl

The backslash character **** escapes the next control character in the **emacs** built-in editor and the next *erase* or *kill* character in the **vi** built-in editor. On by default.

bgnice All background jobs are run at a lower priority. This is the default mode.

braceexpand

Same as **-B**.

emacs Activates the *emacs*-style command line editor. See *Emacs Editing Mode* above.

errexit Same as **-e**.

functrace

Causes the **-x** option's state and the **DEBUG** trap action to be inherited by functions defined using the **function** keyword (see *Functions* above) instead of being reset to default. Changes made to them within the function do not propagate back to the parent scope. Similarly, this option also causes the **DEBUG** trap action to be inherited by subshells.

globcasedetect

When this option is turned on, globbing (see *Pathname Expansion* above) and file name listing and completion (see *In-line Editing Options* above) automatically become case-insensitive on file systems where the difference between upper- and lowercase is ignored for file names. This is transparently determined for each directory, so a path pattern that spans multiple file systems can be part case-sensitive and part case-insensitive. In more precise terms, each slash-separated path name component pattern *p* is treated as $\tilde{(i;p)}$ if its parent directory exists on a case-insensitive file system. This option is only present on operating systems that support case-insensitive file systems.

globstar Same as **-G**.

gmacs Activates the *emacs*-style command line editor with modified **^T**. See *Emacs Editing Mode* above.

histexpand

Same as **-H**.

histredit

If a history expansion (see **-H**) fails, the command line is reloaded into the next prompt's edit buffer, allowing corrections.

histverify

The results of a history expansion (see **-H**) are not immediately executed. Instead, the expanded line is loaded into the next prompt's edit buffer, allowing further changes.

ignoreeof

An interactive shell will not exit on end-of-file. The command **exit** must be used.

keyword

Same as **-k**.

letoctal The **let** command allows octal numbers starting with **0**. On by default if ksh is invoked as **sh** or **rsh**.

markdirs

All directory names resulting from pathname expansion have a trailing **/** appended.

monitor Same as **-m**.

multiline

The built-in editors will use multiple lines on the screen for lines that are longer than the width of the screen. This may not work for all terminals. The

shell uses the system default **tput**(1) command to obtain the terminal escape codes for the necessary operations. Multi-line editing is disabled if this fails. On most systems, setting the **TERM** variable to your terminal's type and exporting it corrects this situation. The **multiline** option is ineffectual on systems whose **tput**(1) command supports neither **terminfo**(5) nor **termcap**(5) capability names.

noclobber

Same as **-C**.

noexec

Same as **-n**.

noglob

Same as **-f**.

nolog

Obsolete; has no effect.

notify

Same as **-b**.

nounset

Same as **-u**.

pipefail

The exit status of the entire pipeline will be that of the last component command that exited with a non-zero exit status, or zero if no command exited with a non-zero exit status. The shell will wait for all component commands of the pipeline to terminate, instead of only waiting for the last component command.

posix

Enables the POSIX standard mode for maximum compatibility with other compliant shells. At the moment that the **posix** option is turned on, it also turns on **letoctal** and turns off **-B/braceexpand**; the reverse is done when **posix** is turned back off. (These options can still be controlled independently in between.) Furthermore, the **posix** option is automatically turned on upon invocation if the shell is invoked as **sh** or **rsh**, or if **-o posix** or **--posix** is specified on the shell invocation command line, or when executing scripts without a **#!** path with this option active in the invoking shell. In that case, the invoked shell will not import type attributes for variables (such as integer or left/right justify) from the environment.

In addition, while on, the **posix** option

- disables exporting variable type attributes to the environment for other ksh processes to import;
- disallows brace expansion on the results of unquoted expansions (if the **-B/braceexpand** option is turned back on);
- disables the special handling of repeated *isspace* class characters in the **IFS** variable;
- causes **redirect** and **exec** to open file descriptors > 2 without the close-on-exec flag, so that they are left open when invoking another program;
- disables the **&>** redirection shorthand;
- disables fast filescan loops of type **while inputredirection ;do list ;done**;
- makes the **<>** redirection operator default to redirecting standard input if no file descriptor number precedes it;
- causes the shell to use a standard UNIX *pipe*(2) instead of a *socketpair*(2) to connect commands in a pipeline (when reading directly from a pipeline, the **<#pattern** and **<##pattern** redirection operators will not work and the **-n** option to the **read** built-in will not return early when reading from a slow device);
- disables the special floating point constants **Inf** and **NaN** in arithmetic evaluation so that, e.g., **\$((inf))** and **\$((nan))** refer to the variables by those names;
- enables the recognition of a leading zero as introducing an octal number in all arithmetic evaluation contexts, except in the **let** built-in while **letoctal** is off;

- disables zero-padding of seconds in the output of the **time** and **times** built-ins;
- stops the **.** command (but not **source**) from looking up functions defined with the **function** syntax;
- disables the recognition of unexpanded shell arithmetic expressions for the numerical conversion specifiers of the **printf** built-in command, causing them to print a warning for operands that are not valid decimal, **0x**-prefixed hexadecimal or **0**-prefixed octal numbers;
- stops the **kill** built-in command from automatically sending a **CONT** signal to stopped jobs or processes upon sending a **HUP** or **TERM** signal;
- disables the special handling of **/dev/fd/n** in file existence and access test operators in **test/** and **[[**, causing these to operate on the file **/dev/fd/n** in the file system;
- disables the recognition of unexpanded shell arithmetic expressions in the numerical comparison operators **-eq**, **-ne**, **-gt**, **-ge**, **-lt** and **-le** of the **test/** built-in command, causing them to accept only decimal numbers as operands;
- changes the **test/** built-in command to make its deprecated **expr1 -a expr2** and **expr1 -o expr2** operators work even if **expr1** equals **!"** or **(** (which means the nonstandard unary **-a file** and **-o option** operators cannot be directly negated using **!** or wrapped in parentheses); and
- disables a hack that makes **test -t** (**[-t]**) equivalent to **test -t 1** (**[-t 1]**).

privileged

Same as **-p**.

showme When enabled, simple commands or pipelines preceded by a semicolon (**;**) will be displayed as if the **xtrace** option were enabled but will not be executed. Otherwise, the leading **;** will be ignored.

trackall Same as **-h**.

verbose Same as **-v**.

vi Activates the *vi*-style command line editor, initially in input mode. See *Vi Editing Mode* above.

viraw Obsolete; has no effect.

xtrace Same as **-x**.

If no option name is supplied, then the current option settings are printed.

- p** Disables processing of the **\$HOME/.profile** file and uses the file **/etc/suid_profile** instead of the **ENV** file. This mode is on whenever the effective UID (GID) is not equal to the real UID (GID). Turning this off causes the effective UID and GID to be set to the real UID and GID.
- r** Enables the restricted shell. This option cannot be unset once set.
- s** Sort the positional parameters lexicographically.
- t** (Obsolete). Exit after reading and executing one command.
- u** Treat unset parameters as an error when substituting. **\$@** and **\$*** are exempt.
- v** Print shell input lines as they are read.
- x** Print commands and their arguments as they are executed.
- Do not change any of the options; useful in setting **\$1** to a value beginning with **-**. If no arguments follow this option then the positional parameters are unset.

As an obsolete feature, if the first *arg* is **-** then the **-x** and **-v** options are turned off and the next *arg* is treated as the first argument. Using **+** rather than **-** causes these options to be turned off. These options can also be used upon invocation of the shell. The current set of options may be found in **\$-**. Unless **-A** is specified, the remaining arguments are positional parameters and are assigned, in order, to **\$1 \$2 ...**. If no arguments are given, then the names and values of all variables are printed on the standard output.

† **shift** [*n*]

The positional parameters from **\$n+1** ... are renamed **\$1** ... , default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to **\$#**.

sleep [**-s**] *duration*

Suspends execution for the number of decimal seconds or fractions of a second given by *duration*. *duration* can be an integer, floating point value or ISO 8601 duration specifying the length of time to sleep. The option **-s** causes the sleep builtin to terminate when it receives any signal. If *duration* is not specified in conjunction with **-s**, **sleep** will wait for a signal indefinitely.

source *name* [*arg* ...]

Same as **.**, except it is not treated as a special built-in command.

stop *job* ...

Sends a **SIGSTOP** signal to one or more processes specified by *job*, suspending them until they receive **SIGCONT**. The same as **kill -s STOP**.

suspend

Sends a **SIGSTOP** signal to the main shell process, suspending the script or child shell session until it receives **SIGCONT** (for instance, when typing **fg** in the parent shell). Equivalent to **kill -s STOP "\$\$"**, except that it accepts no operands and refuses to suspend a login shell.

test *expression*

The **test** and **[** commands execute conditional expressions similar to those specified for the **[[** compound command under *Conditional Expressions* above, but with several important differences. The **=**, **==** and **!=** operators test for string (in)equality without pattern matching; **==** is nonstandard and unportable. The **&&** and **||** operators are not available. Instead, the **-a** and **-o** binary operators can be used, but they are fraught with pitfalls due to grammatical ambiguities and therefore deprecated in favor of invoking separate **test** commands. Most importantly, as **test** and **[** are simple regular commands, field splitting and pathname expansion *are* performed on all their arguments and all aspects of regular shell grammar (such as redirection) remain active. This is usually harmful, so care must be taken to quote arguments and expansions to avoid this. To avoid the many pitfalls arising from these issues, the **[[** compound command should be used instead. The primary purpose of the **test** and **[** commands is compatibility with other shells that lack **[[**.

The **test/[** command does not parse options except if there are two arguments and the second is **--**. To access the inline documentation with an option such as **--man**, you need one of the forms **test --man --** or **[--man --]**.

times Displays the accumulated user and system CPU times, one line with the times used by the shell and another with those used by all of the shell's child processes. No options are supported. Seconds are zero-padded unless the **posix** shell option is on.

† **trap** [**-p**] [*action*] [*sig*] ...

The **-p** option causes the trap action associated with each trap as specified by the arguments to be printed with appropriate quoting. Otherwise, *action* will be processed as if it were an argument to **eval** when the shell receives signal(s) *sig*. Each *sig* can be given as a number or as the name of the signal. Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If *action* is omitted and the first *sig* is a number, or if *action* is **-**, then the trap(s) for each *sig* are reset to their original values. If *action* is the empty string, then this signal is ignored by the shell and by the commands it invokes. If *sig* is **ERR** then *action* will be executed whenever a command has a non-zero exit status. If *sig* is **DEBUG** then *action* will be executed before each command. The variable **.sh.command** will contain the current command line when *action* is running, in the same format as the output generated by the **xtrace** option (minus the preceding **PS4** prompt). If the exit status of the trap is **2** the command will not be executed. If the exit status of the trap is **255** and inside a function or a dot script, the function or dot script will return. If *sig* is **0** or **EXIT** and the **trap** statement is executed inside the body of a function defined with the **function name** syntax, then the command *action* is executed after the function completes. If *sig* is **0** or **EXIT** for a **trap** set outside any function then

the command *action* is executed on exit from the shell. If *sig* is **KEYBD**, then *action* will be executed whenever a key is read while in **emacs**, **gmacs**, or **vi** mode. The **trap** command with no arguments prints a list of commands associated with each signal number.

An **exit** or **return** without an argument in a trap action will preserve the exit status of the command that invoked the trap.

true Does nothing, and exits 0. Used with **while** for infinite loops.

type [**-afpPqt**] *name* ...

The same as **whence -v**.

†† **typeset** [**±ACHSbfimnprstux**] [**±EFLRXZi**[*n*]] [**+-M** [*mapname*]] [**-T** [*tname*=(*assign_list*)]] [**-h** *str*] [**-a** [[*type*]]] [*vname*[=*value*]] ...

Sets attributes and values for shell variables and functions. When invoked inside a function defined with the **function** *name* syntax, a new instance of the variable *vname* is created, and the variable's value and type are restored when the function completes. The following list of attributes may be specified:

- A** Declares *vname* to be an associative array. Subscripts are strings rather than arithmetic expressions.
- C** Causes each *vname* to be a compound variable. If *value* names a compound variable, it is copied into *vname*. Otherwise, the empty compound value is assigned to *vname*.
- a** Declares *vname* to be an indexed array. This is the default. Subscripts are numerical and start at 0. To make it possible to use alphanumeric enumeration constants of a given type as subscripts, an option value of the form [*type*] can be specified (including the surrounding square brackets), which should be quoted to avoid pathname expansion), where *type* must be the name of an enumeration type created with the **enum** command.
- E** Declares *vname* to be a double precision floating point number. If *n* is non-zero, it defines the number of significant figures that are used when expanding *vname*. Otherwise, ten significant figures will be used.
- F** Declares *vname* to be a double precision floating point number. If *n* is non-zero, it defines the number of places after the decimal point that are used when expanding *vname*. Otherwise ten places after the decimal point will be used.
- H** This option provides UNIX to host-name file mapping on non-UNIX machines.
- L** Left justify and remove leading blanks from *value*. If *n* is non-zero, it defines the width of the field, otherwise it is determined by the width of the value of first assignment. When the variable is assigned to, it is filled on the right with blanks or truncated, if necessary, to fit into the field. The **-R** option is turned off.
- M** Use the character mapping *mapping* defined by *wctrans*(3). such as **tolower** and **toupper** when assigning a value to each of the specified operands. When *mapping* is specified and there are not operands, all variables that use this mapping are written to standard output. When *mapping* is omitted and there are no operands, all mapped variables are written to standard output.
- R** Right justify and fill with leading blanks. If *n* is non-zero, it defines the width of the field, otherwise it is determined by the width of the value of first assignment. The field is left filled with blanks or truncated from the end if the variable is reassigned. The **-L** option is turned off.
- S** When used within the *assign_list* of a type definition, it causes the specified subvariable to be shared by all instances of the type. When used inside a function defined with the **function** reserved word, the specified variables will have *function static* scope. Otherwise, the variable is unset prior to processing the assignment list.
- T** If followed by *tname*, it creates a type named by *tname* using the compound assignment *assign_list* to *tname*. Otherwise, it writes all the type definitions to standard output.
- X** Declares *vname* to be a double precision floating point number and expands using the **%a** format of ISO-C99. If *n* is non-zero, it defines the number of hex digits after the radix point that is used when expanding *vname*. The default is 10.

- Z** Right justify and fill with leading zeros if the first non-blank character is a digit and the **-L** option has not been set. Remove leading zeros if the **-L** option is also set. If *n* is non-zero, it defines the width of the field, otherwise it is determined by the width of the value of first assignment.
- f** The names refer to function names rather than variable names. No assignments can be made and the only other valid options are **-S**, **-t**, **-u** and **-x**. The **-S** can be used with discipline functions defined in a type to indicate that the function is static. For a static function, the same method will be used by all instances of that type no matter which instance references it. In addition, it can only use value of variables from the original type definition. These discipline functions cannot be redefined in any type instance. The **-t** option turns on execution tracing for this function. The **-u** option causes this function to be marked undefined. The **FPATH** variable will be searched to find the function definition when the function is referenced. If no options other than **-f** is specified, then the function definition will be displayed on standard output. If **+f** is specified, then a line containing the function name followed by a shell comment containing the line number and path name of the file where this function was defined, if any, is displayed. The exit status can be used to determine whether the function is defined so that **typeset -f .sh.math.name** will return 0 when math function *name* is defined and non-zero otherwise.
- b** The variable can hold any number of bytes of data. The data can be text or binary. The value is represented by the base64 encoding of the data. If **-Z** is also specified, the size in bytes of the data in the buffer will be determined by the size associated with the **-Z**. If the base64 string assigned results in more data, it will be truncated. Otherwise, it will be filled with bytes whose value is zero. The **printf** format **%B** can be used to output the actual data in this buffer instead of the base64 encoding of the data.
- g** Forces variables to be created or modified at the global scope, even when **typeset** is executed in a function defined by the **function name** syntax (see *Functions* above) or in a name space (see *Name Spaces* above).
- h** Used within type definitions to add information when generating information about the subvariable on the man page. It is ignored when used outside of a type definition. When used with **-f** the information is associated with the corresponding discipline function.
- i** Declares *vname* to be represented internally as integer. The right hand side of an assignment is evaluated as an arithmetic expression when assigning to an integer. If *n* is non-zero, it defines the output arithmetic base, otherwise the output base will be ten.
- l** Used with **-i**, **-E** or **-F**, to indicate long integer, or long float. Otherwise, all uppercase characters are converted to lowercase. The uppercase option, **-u**, is turned off. Equivalent to **-M tolower**.
- m** Moves or renames the variable. The value is the name of a variable whose value will be moved to *vname*. The original variable will be unset. Cannot be used with any other options.
- n** Declares *vname* to be a reference to the variable whose name is defined by the value of variable *vname*. This is usually used to reference a variable inside a function whose name has been passed as an argument. Cannot be used with other options except **-g**.
- p** The name, attributes and values for the given *vnames* are written on standard output in a form that can be used as shell input. If **+p** is specified, then the values are not displayed.
- r** The given *vnames* are marked read-only and these names cannot be changed by subsequent assignment.
- s** When given along with **-i**, restricts integer size to short.
- t** Tags the variables. Tags are user definable and have no special meaning to the shell.
- u** When given along with **-i**, specifies unsigned integer. Otherwise, all lowercase characters are converted to uppercase. The lowercase option, **-l**, is turned off. Equivalent to **-M toupper**.
- x** The given *vnames* are marked for automatic export to the *environment* of subsequently-executed commands. Variables whose names contain a **.** cannot be exported.

The **-i**, **-F**, **-E**, and **-X** options cannot be specified along with **-R**, **-L**, or **-Z**. The **-b** option

cannot be specified along with **-L**, **-u**, or **-l**. The **-f**, **-m**, **-n**, and **-T** options cannot be used together with any other option.

Using **+** rather than **-** causes these options to be turned off. If no *vname* arguments are given, a list of *vnames* (and optionally the *values*) of the *variables* is printed. (Using **+** rather than **-** keeps the values from being printed.) The **-p** option causes **typeset** followed by the option letters to be printed before each name rather than the names of the options. If any option other than **-p** is given, only those variables which have all of the given options are printed. Otherwise, the *vnames* and *attributes* of all *variables* that have attributes are printed.

ulimit [**-HSaMctdfkxlqenVuPpmrRbiswTv**] [*limit*]

Set or display a resource limit. The available resource limits are listed below. Many systems do not support one or more of these limits. The limit for a specified resource is set when *limit* is specified. The value of *limit* can be a number in the unit specified below with each resource, or the value **unlimited**. The **-H** and **-S** options specify whether the hard limit or the soft limit for the given resource is set. A hard limit cannot be increased once it is set. A soft limit can be increased up to the value of the hard limit. If neither the **H** nor **S** option is specified, the limit applies to both. The current resource limit is printed when *limit* is omitted. In this case, the soft limit is printed unless **H** is specified. When more than one resource is specified, then the limit name and unit is printed before the value.

- a** Lists all of the current resource limits.
- b** The socket buffer size in bytes.
- c** The number of 512-byte blocks on the size of core dumps.
- d** The number of K-bytes on the size of the data area.
- e** The scheduling priority.
- f** The number of 512-byte blocks on files that can be written by the current process or by child processes (files of any size may be read).
- i** The signal queue size.
- k** The max number of kqueues created by the current user.
- l** The locked address space in K-bytes.
- M** The address space limit in K-bytes.
- m** The number of K-bytes on the size of physical memory.
- n** The number of file descriptors plus 1.
- P** The max number of pseudo-terminals created by the current user.
- p** The number of 512-byte blocks for pipe buffering.
- q** The message queue size in K-bytes.
- R** The max time a real-time process can run before blocking, in microseconds. If this limit is exceeded the process is sent a **SIGXCPU** signal.
- r** The max real-time priority.
- s** The number of K-bytes on the size of the stack area.
- T** The number of threads.
- t** The number of CPU seconds to be used by each process.
- u** The number of processes.
- V** The number of open vnode monitors.
- v** The number of K-bytes for virtual memory.
- w** The swap size in K-bytes.
- x** The number of file locks.

If no option is given, **-f** is assumed.

umask [**-S**] [*mask*]

The user file-creation mask is set to *mask* (see *umask(2)*). *mask* can either be an octal number or a symbolic value as described in *chmod(1)*. If a symbolic value is given, the new umask value is the complement of the result of applying *mask* to the complement of the previous umask value. If *mask* is omitted, the current value of the mask is printed. The **-S** option causes the mode to be printed as a symbolic value. Otherwise, the mask is printed in octal.

unalias [**-a**] *name* ...

The aliases given by the list of *names* are removed from the alias list. The **-a** option causes all the aliases to be unset.

† **unset** [**-fnv**] *vname* ...

The variables given by the list of *vnames* are unassigned, i.e., except for subvariables within a type, their values and attributes are erased. For subvariables of a type, the values are reset to the default value from the type definition. Readonly variables cannot be unset. If the **-f** option is set, then the names refer to *function* names. If the **-v** option is set, then the names refer to *variable* names. The **-f** option overrides **-v**. If **-n** is set and *name* is a name reference, then *name* will be unset rather than the variable that it references. The default is equivalent to **-v**. Unsetting **LINENO**, **MAILCHECK**, **OPTARG**, **OPTIND**, **RANDOM**, **SECONDS**, **TMOUT**, and **_** removes their special meaning even if they are subsequently assigned to.

wait [*job* ...]

Wait for the specified *job* and report its termination status. If *job* is not given, then all currently active child processes are waited for. The exit status from this command is that of the last process waited for if *job* is specified; otherwise it is zero. See *Jobs* for a description of the format of *job*.

whence [**-afpPqtv**] *name* ...

For each *name*, indicate how it would be interpreted if used as a command name.

The **-v** option produces a more verbose report. The **-f** option skips the search for functions. The **-p** and **-P** options do a path search for *name* even if *name* is an alias, a function, or a reserved word. Both of these options turn off the **-v** option. The **-q** option causes **whence** to enter quiet mode. **whence** will return zero if all arguments are built-ins, functions, or are programs found on the path. The **-t** option only outputs the type of the given command. Like **-p** and **-P**, **-t** will turn off the **-v** option. The **-a** option is similar to the **-v** option but causes all interpretations of the given name to be reported.

Invocation.

If the shell is invoked by *exec*(2), initialization depends on argument zero (**\$0**) as follows. If the first character of **\$0** is **-**, or the **-l** option is given on the invocation command line, then the shell is assumed to be a *login* shell. If the basename of the command path in **\$0** is **rsh**, **rksh**, or **krsh**, then the shell becomes restricted. If the basename is **sh** or **rsh**, or the **-o posix** option is given on the invocation command line, then the shell is initialized in full POSIX compliance mode (see the **set** builtin command above for more information). After this, if the shell was assumed to be a *login* shell, commands are read from **/etc/profile** and then from **\$HOME/.profile** if it exists. Alternatively, the option **-l** causes the shell to be treated as a *login* shell. Next, for interactive shells, commands are read from the file named by **ENV** if the file exists, its name being determined by performing parameter expansion, command substitution, and arithmetic expansion on the value of that environment variable. If the **-s** option is not present and *arg* and a file by the name of *arg* exists, then it reads and executes this script. Otherwise, if the first *arg* does not contain a */*, a path search is performed on the first *arg* to determine the name of the script to execute. The script *arg* must have execute permission and any *setuid* and *setgid* settings will be ignored. If the script is not found on the path, *arg* is processed as if it named a built-in command or function. Commands are then read as described below; the following options are interpreted by the shell when it is invoked:

- D** A list of all double quoted strings that are preceded by a **\$** will be printed on standard output and the shell will exit. This set of strings will be subject to language translation when the locale is not C or POSIX. No commands will be executed.
- E** or **-o rc** or **--rc**
Read the file named by the **ENV** variable or by **\$HOME/.kshrc** if not defined after the profiles. On by default for interactive shells. Use **+E**, **+o rc** or **--norc** to turn off.
- c** Read and execute a script from the first *arg* instead of a file. The second *arg*, if present, becomes that script's command name (**\$0**). Any third and further *args* become positional parameters starting at **\$1**.

-s Read and execute a script from standard input instead of a file. The command name (**\$0**) cannot be set. Any *args* become the positional parameters starting at **\$1**. This option is forced on if no *arg* is given and is ignored if **-c** is also specified.

-i or **-o interactive** or **--interactive**

If the **-i** option is present or if the shell's standard input and standard error are attached to a terminal (as told by *tcgetattr(3)*), then this shell is *interactive*. In this case **TERM** is ignored (so that **kill 0** does not kill an interactive shell) and **INTR** is caught and ignored (so that **wait** is interruptible). In all cases, **QUIT** is ignored by the shell.

-r or **-o restricted** or **--restricted**

If the **-r** option is present, the shell is a restricted shell.

The remaining options and arguments are described under the **set** command above. An optional **-** as the first argument is ignored.

Rksh Only.

Rksh is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of **rksh** are identical to those of **ksh**, except that the following are disallowed:

- unsetting the restricted option
- changing directory (see *cd(1)*)
- setting or unsetting the value or attributes of **SHELL**, **ENV**, **FPATH**, or **PATH**
- specifying path or command names containing **/**
- redirecting output (**>**, **>|**, **<>**, and **>>**)
- adding or deleting built-in commands
- using **command -p** to invoke a command

The restrictions above are enforced after **.profile** and the **ENV** files are interpreted.

When a command to be executed is found to be a shell procedure, **rksh** invokes *ksh* to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the **.profile** has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably *not* the login directory).

The system administrator often sets up a directory of commands (e.g., **/usr/rbin**) that can be safely invoked by **rksh**.

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively, then execution of the shell file is abandoned unless the error occurs inside a subshell in which case the subshell is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above). Run time errors detected by the shell are reported by printing the command or function name and the error condition. If the line number that the error occurred on is greater than one, then the line number is also printed in square brackets (**[]**) after the command or function name.

FILES

/etc/profile

The system wide initialization file, executed for login shells.

\$HOME/.profile

The personal initialization file, executed for login shells after **/etc/profile**.

\$HOME/.kshrc

Default personal initialization file, executed for interactive shells when **ENV** is not set.

/etc/suid_profile

Alternative initialization file, executed instead of the personal initialization file when the real and effective user or group ID do not match.

/dev/null

The null device.

SEE ALSO

cat(1), *cd(1)*, *chmod(1)*, *cut(1)*, *date(1)*, *echo(1)*, *emacs(1)*, *env(1)*, *gmacs(1)*, *grep(1)*, *stty(1)*, *test(1)*, *umask(1)*, *vi(1)*, *dup(2)*, *exec(2)*, *fork(2)*, *getpwnam(3)*, *ioctl(2)*, *lseek(2)*, *paste(1)*, *pathconf(2)*, *pipe(2)*, *sysconf(3)*, *umask(2)*, *ulimit(2)*, *wait(2)*, *strftime(3)*, *wctrans(3)*, *rand(3)*, *profile(5)*, *environ(7)*.

Morris I. Bolsky and David G. Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.

POSIX – Part 2: Shell and Utilities, IEEE Std 1003.2-1992, ISO/IEC 9945-2, IEEE, 1993.

CAVEATS

If a command is executed, and then a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to *exec* the original command. Use the **hash** command or the **-t** option of the **alias** command to correct this situation.

Using the **hist** built-in command within a compound command will cause the whole command to disappear from the history file.

The built-in command *.file* reads the whole file before any commands are executed. Therefore, **alias** and **unalias** commands in the file will not apply to any commands defined in the file.

Traps are not processed while a job is waiting for a foreground process. Thus, a trap on **CHLD** won't be executed until the foreground job terminates.

In locales that use a multibyte character set such as UTF-8, the **KEYBD** trap is only triggered for ASCII characters (1-127).

It is a good idea to leave a space after the comma operator in arithmetic expressions to prevent the comma from being interpreted as the decimal point character in certain locales.