NAME

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

SYNOPSIS

/usr/bin/ksh [±**abCefhikmnoprstuvx**] [±**o** *option*] . . . [−**c** *string*] [*arg*. . .]
/usr/xpg4/bin/sh [±**abCefhikmnoprstuvx**] [±**o** *option*] . . . [−**c** *string*] [*arg*. . .]

/usr/bin/rksh [±**abCefhikmnoprstuvx**] [±**o** *option*] . . . [−**c** *string*] [*arg*. . .]

AVAILABILITY

/usr/bin/ksh
/usr/bin/rksh

SUNWcsu

/usr/xpg4/bin/sh

SUNWxcu4

DESCRIPTION

/usr/xpg4/bin/sh is identical to **/usr/bin/ksh**, 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:

; & ( ) | < > NEWLINE 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 names for *functions* and *variables*. A *word* is a sequence of *characters* separated by one or more 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 *special-command* is a command that is carried out by the shell without creating a separate process. Except for documented side effects, most special commands can be implemented as separate utilities.

Commands

A *simple-command* is a sequence of blank-separated words which may be preceded by a variable assignment list. (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 if it terminates normally, or (octal) 200+*status* if it terminates abnormally (see **signal**(3C) for a list of status values).

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 **pipe**(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more *pipeline*s 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 (that is, the shell does *not* wait for that pipeline to finish). The symbol| **&** causes asynchronous execution of the preceding command or pipeline with a two-way pipe established to the parent shell.

The standard input and output of the spawned command can be written to and read from by the parent shell using the **−p** option of the special commands **read** and **print** described in **Special Commands**. The symbol **&&** (| |) causes the *list* following it to be executed only if the preceding pipeline returns **0** (or a non-zero) value. An arbitrary number of new-lines may appear in a *list*, instead of a semicolon, to delimit a command.

A *command* is either a *simple-command* or 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 *identifier* [**in** *word* . . .] **; do** *list* **; done**
: Each time a **for** command is executed, *identifier* 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 (see **Parameter Substitution** below). Execution ends when there are no more words in the list.

select *identifier* [**in** *word* . . .] **; do** *list* **; done**
: A **select** command prints to standard error (file descriptor 2), the set of *word*s, each preceded by a number. If **in** *word* . . . is omitted, then the positional parameters are used instead (see **Parameter Substitution** 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 *word*s, then the value of the variable *identifier* 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 *identifier* is set to **NULL**. (See **Blank Interpretation** about **NULL**). The contents of the line read from standard input is saved in the shell variable **REPLY**. The *list* is executed for each selection until a **break** or EOF is encountered. If the **REPLY** variable is set to **NULL** 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 file-name generation (see **File Name Generation** below).

if *list* **; then** *list* **;** [**elif** *list* **; then** *list* **;** . . .] [**else** *list* **;**] **fi**
: The *list* following **if** is executed and, if it returns an exit status of **0**, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is **0**, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns **0** 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 **0**, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns **0** exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list***)** Execute *list* in a separate environment. Note, that if two adjacent open parentheses are needed for nesting, a space must be inserted to avoid arithmetic evaluation as described below.

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

[[*expression***]]**
Evaluates *expression* and returns **0** exit status when *expression* is true. See **Conditional Expressions** below, for a description of *expression*.

function *identifier* **{** *list* **;}**
*identifier***( ) {** *list* **;}**
Define a function which is referenced by *identifier*. The body of the function is the *list* of commands between { and }. (See **Functions** below).

time *pipeline*
The *pipeline* is executed and the elapsed time as well as the user and system time are printed to standard error.

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

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

Comments

A word beginning with **#** causes that word and all the following characters up to a newline 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 and command substitution 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 special builtin commands but cannot be used to redefine the reserved words listed above. Aliases can be created, listed, and exported with the **alias** command and can be removed with the **unalias** command. Exported aliases remain in effect for scripts invoked by name, but must be reinitialized for separate invocations of the shell (see **Invocation** below). To prevent infinite loops in recursive aliasing, if the shell is not currently processing an alias of the same name, the word will be replaced by the value of the alias; otherwise, it will not be replaced.

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.

Aliases are frequently used as a short hand for full path names. An option to the aliasing facility allows the value of the alias to be automatically set to the full pathname of the corresponding command. These aliases are called *tracked* aliases. The value of a *tracked* alias is defined the first time the corresponding command is looked up and becomes undefined each time the **PATH** variable is reset. These aliases remain *tracked* so that the next subsequent reference will redefine the value. Several tracked aliases are compiled into the shell. The **−h** option of the **set** command makes each referenced command name into a tracked alias.

The following *exported aliases* are compiled into (and built-in to) the shell but can be unset or redefined:

autoload=′typeset −fu′
false=′let 0′
functions=′typeset −f′
hash=′alias −t′
history=′fc −l′
integer=′typeset −i′
nohup=′nohup ′
r=′fc −e −′
true=′:′
type=′whence −v′

An example concerning trailing blank characters and reserved words follows. If the user types:

$ **alias foo="/bin/ls "**
$ **alias while="/"**

The effect of executing:

$ **while true**
> do
> echo "Hello, World"
> done

is a never-ending sequence of **Hello, World** strings to the screen. However, if the user types:

$ **foo while**

the result will be an **ls** listing of **/**. Since the alias substitution for **foo** ends in a space character, the next word is checked for alias substitution. The next word, **while**, has also been aliased, so it is substituted as well. Since it is not in the proper position as a command word, it is not recognized as a reserved word.

If the user types:

$ **foo; while**

while retains its normal reserved-word properties.

Tilde Substitution

After alias substitution is performed, each word is checked to see if it begins with an unquoted ~. If it does, then the word up to a **/** is checked to see if it matches a user name. If a match is found, the ~ and the matched login name are replaced by the login directory of the matched user. This is called a *tilde* substitution. If no match is found, the original text is left unchanged. A ~ by itself, or in front of a **/**, is replaced by **$HOME**. A ~ followed by a **+** or – is replaced by **$PWD** and **$OLDPWD** respectively.

In addition, *tilde* substitution is attempted when the value of a *variable assignment* begins with a ~.

Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, followed by all of the characters preceding the first unquoted slash in the word, or all the characters in the word if there is no slash. In an assignment, multiple tilde-prefixes can be used: at the beginning of the word (that is, following the equal sign of the assignment), following any unquoted colon or both. A tilde-prefix in an assignment is terminated by the first unquoted colon or slash. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name from the user database.

A portable login name cannot contain characters outside the set given in the description of the **LOGNAME** environment variable. If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix will be replaced by the value of the variable **HOME.** If **HOME** is unset, the results are unspecified. Otherwise, the tilde-prefix will be replaced by a pathname of the home directory associated with the login name obtained using the **getpwnam** function. If the system does not recognize the login name, the results are undefined.

Tilde expansion generally occurs only at the beginning of words, but an exception based on historical practice has been included:

```
PATH=/posix/bin:˜dgk/bin
```

is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behavior because any of the following are reasonable substitutes:

```
PATH=$(printf %s ˜karels/bin : ˜bostic/bin)
for Dir in ˜maart/bin ˜srb/bin . . .
do
        PATH=${PATH:+$PATH:}$Dir
done
```

With the first command, explicit colons are used for each directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.

Note that expressions in operands such as:

make -k mumble LIBDIR=˜chet/lib

do not qualify as shell variable assignments and tilde expansion is not performed (unless the command does so itself, which **make** does not).

The special sequence **$˜** has been designated for future implementations to evaluate as a means of forcing tilde expansion in any word.

Because of the requirement that the word not be quoted, the following are not equivalent; only the last will cause tilde expansion:

\˜hlj/ ˜h\lj/ ˜"hlj"/ ˜hlj\/ ˜hlj/

The results of giving tilde with an unknown login name are undefined because the Korn-Shell **˜+** and **˜–** constructs make use of this condition, but, in general it is an error to give an incorrect login name with tilde. The results of having **HOME** unset are unspecified because some historical shells treat this as an error.

Command Substitution

The standard output from a *command* enclosed in parenthesis preceded by a dollar sign (**$(***command***)**) or a pair of grave accents (``) may be used as part or all of a word; trailing new-lines are removed. In the second (archaic) form, the string between the quotes is processed for special quoting characters before the command is executed. (See **Quoting** below.) The command substitution **$(cat** *file***)** can be replaced by the equivalent but faster **(<***file***)**. Command substitution of most special commands that do not perform input/output redirection are carried out without creating a separate process.

Command substitution allows the output of a command to be substituted in place of the command name itself. Command substitution occurs when the command is enclosed as follows:

$(*command*)

or (backquoted version):

`*command*`

The shell will expand the command substitution by executing *command* in a subshell environment and replacing the command substitution (the text of *command* plus the enclosing **$( )** or backquotes) with the standard output of the command, removing sequences of one or more newline characters at the end of the substitution. Embedded newline characters before the end of the output will not be removed; however, they may be treated as field delimiters and eliminated during field splitting, depending on the value of **IFS** and quoting that is in effect.

Within the backquoted style of command substitution, backslash shall retain its literal meaning, except when followed by:

**$ ` **

(dollar-sign, backquote, backslash). The search for the matching backquote is satisfied by the first backquote found without a preceding backslash; during this search, if a non-escaped backquote is encountered within a shell comment, a here-document, an embedded command substitution of the $(*command*) form, or a quoted string, undefined results occur. A single- or double-quoted string that begins, but does not end, within the **...** sequence produces undefined results.

With the $(*command*) form, all characters following the open parenthesis to the matching closing parenthesis constitute the *command*. Any valid shell script can be used for *command*, except:

- A script consisting solely of redirections produces unspecified results.
- See the restriction on single subshells described below.

The results of command substitution will not be field splitting and pathname expansion processed for further tilde expansion, parameter expansion, command substitution or arithmetic expansion. If a command substitution occurs inside double-quotes, it will not be performed on the results of the substitution.

Command substitution can be nested. To specify nesting within the backquoted version, the application must precede the inner backquotes with backslashes; for example:

` \`*command* \``

The **$( )** form of command substitution solves a problem of inconsistent behavior when using backquotes. For example:

Command	Output
echo ' \$x'	\$x
echo `echo ' \$x'`	$x
echo $(echo ' \$x')	\$x

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the new **$( )** form can process any kind of valid embedded script, the backquoted form cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

echo ` cat << eeof a here-doc with ` eof `	echo $(cat << eeof a here-doc with) eof)
echo ` echo abc # a comment with ` `	echo $(echo abc # a comment with))
echo ` echo '`' `	echo $(echo ')')

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts.

If the command substitution consists of a single subshell, such as:

$((*command*))

a portable application must separate the **$(** and **(** into two tokens (that is, separate them with white space). This is required to avoid any ambiguities with arithmetic expansion.

Arithmetic Expansion

An arithmetic expression enclosed in double parentheses preceded by a dollar sign (**$((***arithmetic-expression***))**) is replaced by the value of the arithmetic expression within the double parenthesis. Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion is as follows:

$((*expression*))

The expression is treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell will expand all tokens in the expression for parameter expansion, command substitution and quote removal.

Next, the shell will treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression will be processed according to the rules of the ISO C with the following exceptions:

- Only integer arithmetic is required.
- The **sizeof( )** operator and the prefix and postfix **++** and −− operators are not required.
- Selection, iteration and jump statements are not supported.

As an extension, the shell may recognize arithmetic expressions beyond those listed. If the expression is invalid, the expansion will fail and the shell will write a message to standard error indicating the failure.

A simple example using arithmetic expansion:

```
# repeat a command 100 times
x=100
while [ $x -gt 0 ]
do
        command
        x=$(($x-1))
done
```

Process Substitution

This feature is available in SunOS and only on versions of the UNIX operating system that support the **/dev/fd** directory for naming open files. Each command argument of the form **<(***list***)** or **>(***list***)** will run process *list* asynchronously connected to some file in **/dev/fd**. 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, **paste**s 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.

Parameter Substitution

A *parameter* is an *identifier*, one or more digits, or any of the characters ∗, **@**, **#**, **?**, –, **$**, and **!**. A *variable* (a *parameter* denoted by an *identifier*) has a *value* and zero or more *attributes*. *variable*s can be assigned *value*s and *attribute*s by using the **typeset** special command. The attributes supported by the shell are described later with the **typeset** special command. Exported variables pass values and attributes to the environment.

The shell supports a one-dimensional array facility. An element of an array variable is referenced by a *subscript*. A *subscript* is denoted by a **[**, followed by an *arithmetic expression* (see **Arithmetic Evaluation** below) followed by a **]**. To assign values to an array, use **set –A** *name value* **. . ..** The *value* of all subscripts must be in the range of 0 through 1023. Arrays need not be declared. Any reference to a variable with a valid subscript is legal and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing the element **0**. If an array *identifier* with subscript ∗ or **@** is used, then the value for each of the elements is substituted (separated by a field separator character).

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

name=*value* [*name*=*value*] . . .

If the integer attribute, –**i**, is set for *name*, 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 command. Parameter **$0** is set from argument zero when the shell is invoked. 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.

Parameter Expansion

The format for parameter expansion is as follows:

${*expression*}

where *expression* consists of all characters until the matching }. Any } escaped by a backslash or within a quoted string, and characters in embedded arithmetic expansions, command substitutions and variable expansions, are not examined in determining the matching }.

The simplest form for parameter expansion is:

${*parameter*}

The value, if any, of *parameter* will be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when *parameter* is followed by a character that could be interpreted as part of the name. The matching closing brace will be determined by counting brace levels, skipping over enclosed quoted strings and command substitutions.

If the parameter name or symbol is not enclosed in braces, the expansion will use the longest valid name whether or not the symbol represented by that name exists. When the shell is scanning its input to determine the boundaries of a name, it is not bound by its knowledge of what names are already defined. For example, if **F** is a defined shell variable, the command:

echo $Fred

does not echo the value of **$F** followed by **red**; it selects the longest possible valid name, **Fred**, which in this case might be unset.

If a parameter expansion occurs inside double-quotes:

- Pathname expansion will not be performed on the results of the expansion.
- Field splitting will not be performed on the results of the expansion, with the exception of @.

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of *parameter*, as described below), *word* will be subjected to tilde expansion, parameter expansion, command substitution and arithmetic expansion. If *word* is not needed, it will not be expanded. The } character that delimits the following parameter expansion modifications is determined as described previously in this section and in **dquote**. (For example, **${foo-bar}xyz}** would result in the expansion of **foo** followed by the string **xyz}** if **foo** is set, else the string **barxyz}**).

${*parameter*:–*word*} **Use Default Values.** If *parameter* is unset or null, the expansion of *word* will be substituted; otherwise, the value of *parameter* will be substituted.

${*parameter*:=*word*} **Assign Default Values.** If *parameter* is unset or null, the expansion of *word* will be assigned to *parameter*. In all cases, the final value of *parameter* will be substituted. Only variables, not positional parameters or special parameters, can be assigned in this way.

${*parameter*:?[*word*]} **Indicate Error if Null or Unset.** If *parameter* is unset or null, the expansion of *word* (or a message indicating it is unset if *word* is omitted) will be written to standard error and the shell will exit with a non-zero exit status. Otherwise, the value of *parameter* will be substituted. An interactive shell need not exit.

${*parameter*:+[*word*]} **Use Alternative Value.** If *parameter* is unset or null, null will be substituted; otherwise, the expansion of *word* will be substituted.

In the parameter expansions shown previously, use of the colon in the format results in a test for a parameter that is unset or null; omission of the colon results in a test for a parameter that is only unset. The following table summarizes the effect of the colon:

	parameter set and not null	**parameter set but null**	**parameter unset**
${*parameter***:-***word***}**	substitute *parameter*	substitute *word*	substitute *word*
${*parameter–word***}**	substitute *parameter*	substitute null	substitute *word*
${*parameter***:=***word***}**	substitute *parameter*	assign *word*	assign *word*

${*parameter***=***word***}**	substitute *parameter*	substitute *parameter*	assign null
${*parameter***:?***word***}**	substitute *parameter*	error, exit	error, exit
${*parameter***?***word***}**	substitute *parameter*	substitute null	error, exit
${*parameter***:+***word***}**	substitute *word*	substitute null	substitute null
${*parameter***+***word***}**	substitute *word*	substitute *word*	substitute null

In all cases shown with "substitute", the expression is replaced with the value shown. In all cases shown with "assign" *parameter* is assigned that value, which also replaces the expression.

${#***parameter***}** **String Length**. The length in characters of the value of *parameter*. If *parameter* is * or @, then all the positional parameters, starting with **$1, are substituted (separated by a field separator character).

The following four varieties of parameter expansion provide for substring processing. In each case, pattern matching notation (see **patmat**), rather than regular expression notation, will be used to evaluate the patterns. If *parameter* is * or @, then all the positional parameters, starting with **$1**, are substituted (separated by a field separator character). Enclosing the full parameter expansion string in double-quotes will not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces will have this effect.

${*parameter*%*word*} **Remove Smallest Suffix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the smallest portion of the suffix matched by the *pattern* deleted.

${*parameter*%%*word*} **Remove Largest Suffix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the largest portion of the suffix matched by the *pattern* deleted.

${*parameter*#*word*} **Remove Smallest Prefix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the smallest portion of the prefix matched by the *pattern* deleted.

${*parameter*##*word*} **Remove Largest Prefix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the largest portion of the prefix matched by the *pattern* deleted.

Examples:

${*parameter*:–*word*}

In this example, **ls** is executed only if **x** is null or unset. (The **$(ls)** command substitution notation is explained in **Command Substitution** above.)

```
${x:-$(ls)}
```

${*parameter*:=*word*}

```
unset X
echo ${X:=abc}
abc
```

${*parameter*:?*word*}

```
unset posix
echo ${posix:?}
sh: posix: parameter null or not set
```

${*parameter*:+*word*}

```
set a b c
echo ${3:+posix}
posix
```

${#_parameter_**}**

```
HOME=/usr/posix
echo ${#HOME}
10
```

${*parameter*%*word*}

```
x=file.c
echo ${x%.c}.o
file.o
```

${*parameter*%%*word*}

```
x=posix/src/std
echo ${x%%/*}
posix
```

${*parameter*#*word*}

```
x=$HOME/src/cmd
echo ${x#$HOME}
/src/cmd
```

${*parameter*##*word*}

```
x=/one/two/three
echo ${x##*/}
three
```

Parameters Set by Shell

The following parameters are automatically set by the shell:

The number of positional parameters in decimal.

– Flags supplied to the shell on invocation or by the **set** command.

? The decimal value returned by the last executed command.

$ The process number of this shell.

_ 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.

! The process number of the last background command invoked.

ERRNO The value of **errno** as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes.

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

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

OPTARG The value of the last option argument processed by the **getopts** special command.

OPTIND The index of the last option argument processed by the **getopts** special command.

PPID The process number of the parent of the shell.

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

RANDOM Each time this variable is referenced, a random integer, uniformly distributed between 0 and 32767, is generated. The sequence of random numbers can be initialized by assigning a numeric value to **RANDOM**.

REPLY This variable is set by the **select** statement and by the **read** special 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.

Variables Used by Shell

The following variables are used by the shell:

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

COLUMNS
If this variable is set, the value is used to define the width of the edit window for the shell edit modes and for printing **select** lists.

EDITOR If the value of this variable ends in **emacs**, **gmacs**, or **vi** and the **VISUAL** variable is not set, then the corresponding option (see the **set**

special command below) will be turned on.

ENV This variable, when the shell is invoked, is subjected to parameter expansion by the shell and the resulting value is used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of **ENV** is not an absolute pathname, the results are unspecified. **ENV** will be ignored if the user's real and effective user IDs or real and effective group IDs are different.

This variable can be used to set aliases and other items local to the invocation of a shell. The file referred to by **ENV** differs from **$HOME/.profile** in that **.profile** is typically executed at session startup, whereas the **ENV** file is executed at the beginning of each shell invocation. The **ENV** value is interpreted in a manner similar to a dot script, in that the commands are executed in the current environment and the file needs to be readable, but not executable. However, unlike dot scripts, no **PATH** searching is performed. This is used as a guard against Trojan Horse security breaches.

FCEDIT The default editor name for the **fc** command.

FPATH The search path for function definitions. By default the **FPATH** directories are searched after the **PATH** variable. If an executable file is found, then it is read and executed in the current environment. **FPATH** is searched before **PATH** when a function with the −**u** attribute is referenced. The preset alias **autoload** preset alias causes a function with the −**u** attribute to be created.

IFS Internal field separators, normally **space**, **tab**, and **new-line** that are used to separate command words which result from command or parameter substitution and for separating words with the special command **read**. The first character of the **IFS** variable is used to separate arguments for the **$**∗ substitution (See **Quoting** below).

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 **128**.

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

LC_ALL This variable provides a default value for the **LC_**∗ variables.

LC_COLLATE
This variable determines the behavior of range expressions, equivalence classes and multi-character collating elements within pattern matching.

LC_CTYPE Determines how the shell handles characters. When **LC_CTYPE** is set to a valid value, the shell can display and handle text and filenames containing valid characters for that locale. However, the shell is not multibyte (EUC) capable. In the "C" locale, only ASCII characters are valid. If **LC_CTYPE** (see **environ**(5)) is not set in the environment, the operational behavior of the shell is determined by the value of the **LANG** environment variable. If **LC_ALL** is set, its contents are used to override both the **LANG** and the other **LC_*** variables. If none of the above variables is set in the environment, the "C" (U.S. style) locale prevails.

LC_MESSAGES
This variable determines the language in which messages should be written.

LANG Provide a default value for the internationalization variables that are unset or null. If **LANG** is unset or null, the corresponding value from the default "C" locale will be used. If any of the internationalization variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LINENO This variable is set by the shell to a decimal number representing the current sequential line number (numbered starting with 1) within a script or function before it executes each command. If the user unsets or resets **LINENO,** the variable may lose its special meaning for the life of the shell. If the shell is not currently executing a script or function, the value of **LINENO** is unspecified.

LINES If this variable is set, the value is used to determine the column length for printing **select** lists. Select lists will print vertically until about two-thirds of **LINES** lines are filled.

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 substitution with the variable **$_** defined as the name of the file that has changed. The default message is **you have mail in $_**.

NLSPATH Determine the location of message catalogues for the processing of **LC_MESSAGES.**

PATH The search path for commands (see **Execution** below). The user may not change **PATH** if executing under **rksh** (except in **.profile**).

PPID This variable is set by the shell to the decimal process ID of the process that invoked the shell. In a subshell, **PPID will be set to the** current shell. For example, **echo $PPID** and **(echo $PPID)** would produce the same value.

PS1 The value of this variable is expanded for parameter substitution to define the primary prompt string which by 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.

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 substitution and precedes each line of an execution trace. If omitted, the execution trace prompt is ''**+** ''.

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.

TMOUT If set to a value greater than zero, the shell will terminate if a command is not entered within the prescribed number of seconds after issuing the **PS1** prompt. (Note that the shell can be compiled with a maximum bound for this value which cannot be exceeded.)

VISUAL If the value of this variable ends in **emacs**, **gmacs**, or **vi** then the corresponding option (see Special Command **set** below) will be turned on.

The shell gives default values to **PATH**, **PS1**, **PS2**, **PS3**, **PS4**, **MAILCHECK**, **FCEDIT**, **TMOUT** and **IFS**, while **HOME**, **SHELL ENV** 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**.

Blank Interpretation

After parameter and command substitution, the results of substitutions are scanned for the field separator characters (those found in **IFS**) and split into distinct arguments where such characters are found. Explicit null arguments (**""**) or (**''**) are retained. Implicit null arguments (those resulting from *parameters* that have no values) are removed.

File Name Generation

Following substitution, each command *word* is scanned for the characters ∗, **?**, and **[** unless the −**f** option has been **set**. If one of these characters appears then the word is regarded as a *pattern*. The word is replaced with lexicographically sorted file names that match the pattern. If no file name is found that matches the pattern, then the word is left unchanged. When a *pattern* is used for file name generation, the character period (**.**) at the start of a file name or immediately following a **/**, as well as the character **/** itself, must be matched explicitly. A file name beginning with a period will not be matched with a pattern with the period inside parentheses; that is

ls .@(r*)

would locate a file named **.restore**, but **ls @(.r*)** would not. In other instances of pattern matching the **/** and **.** are not treated specially.

- * Matches any string, including the null string.
- **?** 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 "! ", then any character not enclosed is matched. A – can be included in the character set by putting it as the first or last character.

A *pattern-list* is a list of one or more patterns separated from each other with a | . Composite patterns can be formed with one or more of the following:

- **?(***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.
- **@(***pattern-list***)** Matches exactly one of the given patterns.
- **!(***pattern-list***)** Matches anything, except one of the given patterns.

Quoting

Each of the *metacharacters* listed above (See **Definitions**) has a special meaning to the shell and causes termination of a word unless quoted. A character may be *quoted* (that is, made to stand for itself) by preceding it with a **** . The pair **** NEWLINE is removed. All characters enclosed between a pair of single quote marks (′′) are quoted. A single quote cannot appear within single quotes. Inside double quote marks (**""**), parameter and command substitution occur and **** quotes the characters **** , **`**, **"**, and **$**. The meaning of **$***
and **$@** is identical when not quoted or when used as a parameter assignment value or as a file name. However, when used as a command argument, **$*** is equivalent to **′′$1***d* **$2***d* **. . .′′**, where *d* is the first character of the **IFS** variable, whereas **$@** 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 special command names listed below cannot be altered by quoting them.

Arithmetic Evaluation

An ability to perform integer arithmetic is provided with the special command **let**. Evaluations are performed using *long* arithmetic. Constants are of the form [*base***#**] *n* where *base* is a decimal number between two and thirty-six representing the arithmetic base and *n* is a number in that base. If *base* is omitted then base 10 is used.

An arithmetic expression uses the same syntax, precedence, and associativity of expression as the C language. All the integral operators, other than **++**, **– –**, **?:**, and **,** are supported. Variables can be referenced by name within an arithmetic expression without using the parameter substitution syntax. When a variable is referenced, its value is evaluated as an arithmetic expression.

An internal integer representation of a *variable* can be specified with the –**i** option of the **typeset** special command. Arithmetic evaluation is performed on the value of each assignment to a variable with the –**i** attribute. If you do not specify an arithmetic base, the first assignment to the variable determines the arithmetic base. This base is used when parameter substitution occurs.

Since many of the arithmetic operators require quoting, an alternative form of the **let** command is provided. For any command which begins with a **((**, all the characters until a matching **))** are treated as a quoted expression. More precisely, **((**. . .**))** is equivalent to **let "**. . .**"**.

Prompting

When used interactively, the shell prompts with the parameter expanded value of **PS1** before reading a command. If at any time a new-line is typed and further input is needed to complete a command, then the secondary prompt (that is, 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. Word splitting and file name generation are not performed on the words between **[[** and **]]**. Each expression can be constructed from one or more of the following unary or binary expressions:

–**a** *file*	True, if *file* exists.
–**b** *file*	True, if *file* exists and is a block special file.
–**c** *file*	True, if *file* exists and is a character special file.
–**d** *file*	True, if *file* exists and is a character special file.
–**e** *file*	True, if *file* exists.
–**f** *file*	True, if *file* exists and is an ordinary file.
–**g** *file*	True, if *file* exists and is has its setgid bit set.
–**k** *file*	True, if *file* exists and is has its sticky bit set.
–**n** *string*	True, if length of *string* is non-zero.
–**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 has its setuid bit set.
–**w** *file*	True, if *file* exists and is writable by current process.
–**x** *file*	True, if *file* exists and is executable by current process. If *file* exists and is a directory, then the current process has permission to search in the directory.
–**z** *string*	True, if length of *string* is zero.
–**L** *file*	True, if *file* exists and is a symbolic link.
–**O** *file*	True, if *file* exists and is owned by the effective user id of this process.
–**G** *file*	True, if *file* exists and its group matches the effective group id of this process.

–**S** *file*	True, if *file* exists and is a socket.
file1 –**nt** *file2*	True, if *file1* exists and is newer than *file2*.
file1 –**ot** *file2*	True, if *file1* exists and is older than *file2*.
file1 –**ef** *file2*	True, if *file1* and *file2* exist and refer to the same file.
string **=** *pattern*	True, if *string* matches *pattern*.
string **!=** *pattern*	True, if *string* does not match *pattern*.
string1 **<** *string2*	True, if *string1* comes before *string2* based on ASCII value of their characters.
string1 **>** *string2*	True, if *string1* comes after *string2* based on ASCII value of their characters.
exp1 –**eq** *exp2*	True, if *exp1* is 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*.

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 and parameter substitution occur before *word* or *digit* is used except as noted below. File name generation occurs only if the pattern matches a single file, and blank interpretation is not performed.

<*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.

>|*word* Sames as **>**, except that it overrides the **noclobber** option.

>>*word* Use file *word* as standard output. If the file exists then output is appended to it (by first seeking to the EOF); otherwise, the file is created.

<>*word* Open file *word* for reading and writing as standard input.

<< [–]*word* The shell input is read up to a line that is the same as *word*, or to an EOF. No parameter substitution, command substitution or file name generation is performed on *word*. The resulting document, called a *here-document*, becomes the standard input. If any character of *word* is quoted, then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occur, \ NEWLINE is ignored, and \ must be used to quote the characters \ , **$**, `, and the first character of *word*. If − is appended to **<<**, then all leading tabs are stripped from *word* and from the document.

<&_digit_ The standard input is duplicated from file descriptor *digit* (see **dup**(2)). Similarly for the standard output using **>&**_digit_.

<&− The standard input is closed. Similarly for the standard output using **>&−**.

<&p The input from the co-process is moved to standard input.

>&p The output to the co-process is moved to standard output.

If one of the above is preceded by a digit, then the file descriptor number referred to is that specified by the digit (instead of the default 0 or 1). For example:

. . . 2>&1

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

The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example:

. . . 1>_fname_ **2>&1**

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

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**(5)) 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 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.

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 commands listed below that are preceded with a dagger).

If the −**k** flag 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

The **function** reserved word, described in the **Commands** section above, is used to define shell functions. 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 execute in the same process as the caller and share all files and present working directory with 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 after the function completes in the environment of the caller. Ordinarily, variables are shared between the calling program and the function. However, the **typeset** special command used within a function defines local variables whose scope includes the current function and all functions it calls.

The special command **return** is used to return from function calls. Errors within functions return control to the caller.

The names of all functions can be listed with **typeset +f**. **typeset** −**f** lists all function names as well as the text of all functions. **typeset** −**f** *function-names* lists the text of the named functions only. Functions can be undefined with the −**f** option of the **unset** special command.

Ordinarily, functions are **unset** when the shell executes a shell script. The −**xf** option of the **typeset** command allows a function to be exported to scripts that are executed without a separate invocation of the shell. Functions that need to be defined across separate invocations of the shell should be specified in the **ENV** file with the −**xf** option of **typeset**.

Function Definition Command

A function is a user-defined name that is used as a simple command to call a compound command with new positional parameters. A function is defined with a *function definition command*.

The format of a function definition command is as follows:

fname() *compound-command*[*io-redirect* . . .]

The function is named **fname**; it must be a name. An implementation may allow other characters in a function name as an extension. The implementation will maintain separate name spaces for functions and variables.

The **()** in the function definition command consists of two operators. Therefore, intermixing blank characters with the **fname**, **(**, and **)** is allowed, but unnecessary.

The argument *compound-command* represents a compound command.

When the function is declared, none of the expansions in **wordexp** will be performed on the text in *compound-command* or *io-redirect*; all expansions will be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* will be performed during the execution of the function itself, not the function definition.

When a function is executed, it will have the syntax-error and variable-assignment properties described for the special built-in utilities.

The *compound-command* will be executed whenever the function name is specified as the name of a simple command The operands to the command temporarily will become the positional parameters during the execution of the *compound-command*; the special parameter **#** will also be changed to reflect the number of operands. The special parameter **0** will be unchanged. When the function completes, the values of the positional parameters and the special parameter **#** will be restored to the values they had before the function was executed. If the special built-in **return** is executed in the *compound-command*, the function will complete and execution will resume with the next command after the function call.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls –l
#
[ "$i" = yes ] && foo() {
      ls -l
}
```

The exit status of a function definition will be **0** if the function was declared successfully; otherwise, it will be greater than zero. The exit status of a function invocation will be the exit status of the last command executed by the function.

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** (CTRL-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 **job**s 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.

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

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

You have jobs running.

You will then need to logout a second time to actually logout; however, your background jobs will continue to run.

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** special command below).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the **Special Commands** listed below, it is executed within the current shell process. Next, the command name is checked to see if it matches one of the user defined functions. If it does, the positional parameters are saved and then reset to the arguments of the **function** call. When the **function** completes or issues a **return**, the positional parameter list is restored and any trap set on **EXIT** within the function is executed. The value of a **function** is the value of the last command executed. A function is also executed in the current shell process. If a command name is not a **special command** or a user defined **function**, a process is created and an attempt is made to execute the command via **exec**(2).

The shell variable **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (**:**). The default path is **/bin:/usr/bin:** (specifying **/bin**, **/usr/bin**, and the current directory in that order). 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 path is searched for an executable file. If the file has execute permission but is not a directory or an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. All non-exported aliases, functions, and variables are removed in this case. A parenthesized command is executed in a sub-shell without removing non-exported quantities.

Command Re-entry

The text of the last **HISTSIZE** (default 128) 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 special command **fc** 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 **fc** then the value of the 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. 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, if **r** is aliased to **'fc –e –'** then 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**.

In-line Editing Option

Normally, each command line entered from a terminal device is simply typed followed by a new-line (RETURN or LINEFEED). 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 ending in either of these option names.

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.

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 side of the window, **<** if the line extends on the left, and * if the line extends on both sides 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.

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 LINEFEED key is entered after edit commands except when noted.

^F	Move cursor forward (right) one character.
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.)
^B	Move cursor backward (left) one character.
M-b	Move cursor backward one word.
^A	Move cursor to start of line.
^E	Move cursor to end of line.
^]*char*	Move cursor forward to character *char* on current line.
M-^]*char*	Move cursor backward to character *char* on current line.
^X^X	Interchange the cursor and mark.
erase	(User defined erase character as defined by the **stty**(1) command, usually **^H** or **#**.) Delete previous character.
^D	Delete current character.
M-d	Delete current word.
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 next character in **emacs** mode. Transpose two previous characters in **gmacs** 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.

kill (User defined kill character as defined by the **stty**(1) command, usually **^G** or @.) Kill the entire current line. If two *kill* characters are entered in succession, all kill characters from then on cause a line feed (useful when using paper terminals).

^Y Restore last item removed from line. (Yank item back to the line.)

^L Line feed and print current line.

^@ (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 null.

^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-< 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.

^R*string* Reverse search history for a previous command line containing *string*. If a parameter of zero is given, the search is forward. *string* is terminated by a RETURN or NEW LINE. If string is preceded by a ^, the matched line must begin with *string*. If *string* is omitted, then the next command line containing the most recent *string* is accessed. In this case a parameter of zero reverses the direction of the search.

^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**, *erase*, **^C**, **^D**, **^K**, **^R**, **^P**, **^N**, **^]**, **M-.**, **M-^]**, **M-_**, **M-b**, **M-c**, **M-d**, **M-f**, **M-h**, **M-l** and **M-^H**.

M-*letter* Soft-key. Your alias list is searched for an alias by the name _*letter* and if an alias of this name is defined, its value will be inserted on the input queue. The *letter* must not be one of the above meta-functions.

M-[*letter* Soft-key. Your alias list is searched for an alias by the name __*letter* and if an alias of this name is defined, its value will be inserted on the input queue. The can be used to program functions 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–_ Same as **M–.**.

M–* An asterisk is appended to the end of the word and a file name expansion is attempted.

M–ESC File name completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. 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.

M–= List files matching current word pattern if an asterisk were appended.

^U Multiply parameter of next command by 4.

**** Escape next character. Editing characters, the user's erase, kill and interrupt (normally **^?**) 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).

^V Display version of the shell.

M-# Insert a **#** at the beginning of the line and execute it. This causes a comment to be inserted in the history file.

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.

When in **vi** mode on most systems, canonical processing is initially enabled and the command will be echoed again if the speed is 1200 baud or greater and it contains any control characters or less than one second has elapsed since the prompt was printed. The ESC character terminates canonical processing for the remainder of the command and the user can then modify the command line. This scheme has the advantages of canonical processing with the type-ahead echoing of raw mode.

If the option **viraw** is also set, the terminal will always have canonical processing disabled. This mode is implicit for systems that do not support two alternate end of line delimiters, and may be helpful for certain terminals.

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.

^D Terminate the shell.

^V Escape next character. Editing characters and the user's erase or kill characters may be entered in a command line or in a search string if preceded by a **^V**. The **^V** removes the next character's editing features (if any).

**** Escape the next *erase* or *kill* character.

Motion Edit Commands

These commands will move the cursor.

[*count*]**l**	Cursor forward (right) one character.
[*count*]**w**	Cursor forward one alpha-numeric 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*]**b**	Cursor backward one word.
[*count*]**B**	Cursor to preceding blank separated word.
[*count*]**|**	Cursor to column *count*.
[*count*]**f***c*	Find the next character *c* in the current line.
[*count*]**F***c*	Find the previous character *c* in the current line.
[*count*]**t***c*	Equivalent to **f** followed by **h**.
[*count*]**T***c*	Equivalent to **F** followed by **l**.
[*count*]**;**	Repeats *count* times, the last single character find command, **f**, **F**, **t**, or **T**.
[*count*]**,**	Reverses the last single character find command *count* times.
0	Cursor to start of line.
^	Cursor to first non-blank character in line.
$	Cursor to end of line.
%	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*]**j**	Fetch next command. Each time **j** is entered the next command forward in time is accessed.
[*count*]**+**	Equivalent to **j**.
[*count*]**G**	The command number *count* is fetched. The default is the least recent history command.
/*string*	Search backward through history for a previous command containing *string*. *string* is terminated by a RETURN or NEWLINE. If *string* is preceded by a **^**, the matched line must begin with *string*. If *string* is **NULL**, 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. Search history for the *string* entered by the previous **/** command.

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$**.

[*count*]**s** Delete *count* characters and enter input mode.

S Equivalent to **cc**.

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 beginning of the line. Equivalent to **0i**.

[*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*]**r***c* Replace the *count* character(s) starting at the current cursor position with *c*, and advance the cursor.

[*count*]**x** Delete current character.

[*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 file name generation attempted. If no match is found, it rings the bell. Otherwise, the word is replaced by the matching pattern and input mode is entered.

**** Filename completion. Replaces the current word with the longest common prefix of all filenames matching the current word with an asterisk appended. 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.

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.

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 **fc** –**e ${VISUAL:**–**${EDITOR:**–**vi}}** *count* in the input buffer. 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 removing comments from previous comment commands in the history file.

= List the file names that match the current word if an asterisk were appended it.

@letter Your alias list is searched for an alias by the name **_***letter* and if an alias of this name is defined, its value will be inserted on the input queue for processing.

Special Commands

The following *simple-commands* are executed in the shell process. Input/Output redirection is permitted. Unless otherwise indicated, the output is written on file descriptor 1 and the exit status, when there is no syntax error, is **0**. Commands that are preceded by one or two † (daggers) 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. Words, following a command preceded by †† that are in the format of a variable assignment, are expanded with the same rules as a variable assignment. This means that tilde substitution is performed after the **=** sign and word

splitting and file name generation are not performed.

† **:** [*arg* . . .]
The command only expands parameters.

† **.** *file* **[** *arg* **. . .]**
Read the complete *file* then execute the commands. The commands are executed in the current shell environment. The search path specified by **PATH** is used to find the directory containing *file*. If any arguments *arg* are given, they become the positional parameters. Otherwise the positional parameters are unchanged. The exit status is the exit status of the last command executed.

†† **alias** [**−tx**] [*name*[**=***value*]] . . .
alias with no arguments prints the list of aliases in the form *name=value* on standard output. 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. The **−t** flag is used to set and list tracked aliases. The value of a tracked alias is the full pathname corresponding to the given *name*. The value becomes undefined when the value of **PATH** is reset but the aliases remained tracked. Without the **−t** flag, for each *name* in the argument list for which no *value* is given, the name and value of the alias is printed. The **−x** flag is used to set or print *exported alias*es. An *exported alias* is defined for scripts invoked by name. The exit status is non-zero if a *name* is given, but no value, and no alias has been defined for the *name*.

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"** section above for a description of the format of *job*.

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

† **continue** [*n*]
Resume the next iteration of the enclosed **for**, **while**, **until**, or **select** loop. If *n* is specified then resume at the *n*-th enclosed loop.

cd [*arg*]
cd *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 null (specifying the current directory). Note that the current directory is specified by a null path name, 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.

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

command [−**p**] [*command_name*] [**argument . . .**]
command [−**v** −**V**] *command_name*

The **command** utility causes the shell to treat the arguments as a simple command, suppressing the shell function lookup. The −**p** flag performs the command search using a default value for **PATH** that is guaranteed to find all of the standard utilities. The −**v** flag writes a string to standard output that indicates the pathname or command that will be used by the shell, in the current shell execution environment, to invoke *command_name*. The −**V** flag writes a string to standard output that indicates how the name given in the *command_name* operand will be interpreted by the shell, in the current shell execution environment.

echo [*arg* . . .]

See **echo**(1) for usage and description.

† **eval** [*arg* . . .]

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

† **exec** [*arg* . . .]

If *arg* is given, the command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and affect the current process. If no arguments are given the effect of this command is to modify file descriptors as prescribed by the input/output redirection list. In this case, any file descriptor numbers greater than 2 that are opened with this mechanism are closed when invoking another program.

† **exit** [*n*]

Causes the calling shell or shell script to exit with the exit status specified by *n*. The value will be the least significant 8 bits of the specified status. If *n* is omitted then the exit status is that of the last command executed. When **exit** occurs when executing a trap, the last command refers to the command that executed before the trap was invoked. An EOF will also cause the shell to exit except for a shell which has the **ignoreeof** option (See **set** below) turned on.

†† **export** [*name*[=*value*]] . . .

The given *name*s are marked for automatic export to the **environment** of subsequently-executed commands.

fc [−**e** *ename*] [−**nlr**] [*first* [*last*]]
fc −**e** − [*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** flag 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 **FCEDIT** (default **/bin/ed**) is used as the editor. When editing is complete, the edited command(s) is executed. 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 flag −**r** reverses the order of the commands and the flag −**n** suppresses command numbers when listing. In the second form the *command* is re-executed after the substitution *old***=***new* is performed. If there is not a *command* argument, the most recent command typed at this terminal is executed.

fg [%*job*. . .]

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

getopts *optstring name* [*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. *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.

getopts places the next option letter it finds inside variable *name* each time it is invoked with a **+** prepended 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 *name* to **?** for an unknown option and to **:** when a required option is missing. Otherwise, **getopts** prints an error message. The exit status is non-zero when there are no more options. See **getoptcvt**(1) for usage and description.

hash [*name* . . .]

For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The −**r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. If a command is found in a "relative" directory in the search path, after changing to that directory, the stored location of that command is recalculated. Commands for which this will be done are indicated by an asterisk (∗) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

jobs [−**lnp**] [%*job* . . .]

Lists information about each given job; or all active jobs if *job* is omitted. The −**l** flag lists process ids in addition to the normal information. The −**n** flag displays only jobs that have stopped or exited since last notified. The −**p** flag causes only the process group to be listed. See **"Jobs"** section above and **jobs**(1) for a description of the format of *job*.

kill [–*sig*] %*job* **. . .**
kill [–*sig*] *pid* . . .
kill –l

Sends either the **TERM** (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in **signal**(5) stripped of the prefix ''SIG'' with the exception that **SIGCHD** is named **CHLD**). If the signal being sent is **TERM** (terminate) or **HUP** (hangup), 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 second form, **kill** –**l**, the signal numbers and names are listed.

let *arg* **. . .**

Each *arg* is a separate *arithmetic expression* to be evaluated. See the **Arithmetic Evaluation** section 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.

login *argument* . . .

Equivalent to '**exec login** *argument*. . . .' See **login**(1) for usage and description.

† **newgrp** [*arg* . . .]

Equivalent to **exec /bin/newgrp** *arg* . . ..

print [–**Rnprsu**[*n*]] [*arg* . . .]

The shell output mechanism. With no flags or with flag – or – –, the arguments are printed on standard output as described by **echo**(1). The exit status is **0**, unless the output file is not open for writing.

–**n** Suppress NEWLINE from being added to the output.

–**R** | –**r** Raw mode. Ignore the escape conventions of **echo**. The –**R** option will print all subsequent arguments and options other than –**n**.

–**p** Write the arguments to the pipe of the process spawned with | **&** instead of standard output.

–**s** Write the arguments to the history file instead of standard output.

–**u** [*n*] Specify a one digit file descriptor unit number *n* on which the output will be placed. The default is **1**.

pwd Equivalent to **print –r – $PWD print –r – $PWD**

read [–**prsu**[*n*]] [*name***?***prompt*] [*name* . . .]

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. In raw mode, –**r**, the \ character is not treated specially. The first field is assigned to the first *name*, the second field to the second *name*, etc., with leftover fields assigned to the last *name*. The –**p** option causes the input line to be taken from the input pipe of a process spawned by the shell using | **&**. If the –**s** flag is present, the input will be saved as a command in the history file. The flag −**u** can be used to specify a one digit file descriptor unit *n* to read from. The file descriptor can be opened with the **exec** special command. The default value of *n* is **0**. If *name* is omitted then **REPLY** is used as the default *name*. The exit status is **0** unless the input file is not open for reading or an EOF is encountered. An EOF with the −**p** option causes cleanup for this process so that another can be spawned. If the first argument contains a **?**, the remainder of this word is used as a *prompt* on standard error when the shell is interactive. The exit status is **0** unless an EOF is encountered.

†† **readonly** [*name*[**=***value*]] . . .
The given *name*s are marked **readonly** and these names cannot be changed by subsequent assignment.

† **return** [*n*]
Causes a shell function or **'.'** script to return to the invoking script with the return status specified by *n*. The value will be the least significant 8 bits of the specified status. If *n* is omitted then the return status is that of the last command executed. If **return** is invoked while not in a function or a **'.'** script, then it is the same as an **exit**.

set [±**abCefhkmnopstuvx**] [±**o** *option*]. . . [±**A** *name*] [*arg* . . .]
The flags for this command have meaning as follows:

−**A** Array assignment. Unset the variable *name* and assign values sequentially from the list *arg*. If **+A** is used, the variable *name* is not unset first.

−**a** All subsequent variables that are defined are automatically exported.

−**b** Causes the shell to notify the user asynchronously of background job completions. The following message will be written to standard error:

"[%d]%c %s%s\n", <*job-number*>, <*current*>, <*status*>, <*job-name*>

where the fields are as follows:

<*current*> The character **+** identifies the job that would be used as a default for the **fg** or **bg** utilities; this job can also be specified using the *job_id* **%+** or **%%**. The character − identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job_id* **%−**. For other jobs, this field is a space character. At most one job can be identified with **+** and at most one job can be identified with −. If there is any suspended job, then the current job will be a suspended job. If there are at least two suspended jobs, then the previous job will also be a suspended job.

<*job-number*> A number that can be used to identify the process group to the **wait**, **fg**, **bg**, and **kill** utilities. Using these utilities, the job can be identified by prefixing the job

number with %.

<*status*> Unspecified.

<*job-name*> Unspecified.

When the shell notifies the user a job has been completed, it may remove the job's process ID from the list of those known in the current shell execution environment. Asynchronous notification will not be enabled by default.

–**C** Prevent existing files from being overwritten by the shell's **>** redirection operator; the **>|** redirection operator will override this **noclobber** option for an individual file.

–**e** 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 file name generation.

–**h** Each command becomes a tracked alias when first encountered.

–**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. On systems with job control, this flag 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**.
errexit Same as –**e**.
bgnice All background jobs are run at a lower priority. This is the default mode.
emacs Puts you in an **emacs** style in-line editor for command entry.
gmacs Puts you in a **gmacs** style in-line editor for command entry.
ignoreeof The shell will not exit on EOF. The command **exit** must be used.
keyword Same as –**k**.
markdirs All directory names resulting from file name generation have a trailing **/** appended.
monitor Same as –**m**.
noclobber Prevents redirection **>** from truncating existing files. Require **>|** to truncate a file when turned on. Equivalent to –**C**.
noexec Same as –**n**.
noglob Same as –**f**.
nolog Do not save function definitions in history file.

notify	Equivalent to –**b**.
nounset	Same as –**u**.
privileged	Same as –**p**.
verbose	Same as –**v**.
trackall	Same as –**h**.
vi	Puts you in insert mode of a **vi** style in-line editor until you hit escape character **033**. This puts you in control mode. A return sends the line.
viraw	Each character is processed as it is typed in **vi** mode.
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 is not equal to the real uid, or when the effective gid is not equal to the real gid. Turning this off causes the effective uid and gid to be set to the real uid and gid.

–**s** Sort the positional parameters lexicographically.

–**t** Exit after reading and executing one command.

–**u** Treat unset parameters as an error when substituting.

–**v** Print shell input lines as they are read.

–**x** Print commands and their arguments as they are executed.

– Turns off –**x** and –**v** flags and stops examining arguments for flags.

– – Do not change any of the flags; useful in setting **$1** to a value beginning with –. If no arguments follow this flag then the positional parameters are unset.

Using **+** rather than – causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags 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 $***n***+1 . . .** are renamed **$1 . . .**, default *n* is 1. The parameter *n* can be any arithmetic expression that evaluates to a non-negative number less than or equal to **$#**.

stop %*jobid . . .*
stop *pid . . .*

stop stops the execution of a background job(s) by using its *jobid*, or of any process by using its *pid*. (see **ps**(1)).

suspend

Stops the execution of the current shell (but not if it is the login shell).

test *expression*
Evaluate conditional expressions. See **Conditional Expressions** section above and **test**(1) for usage and description.

† **times** Print the accumulated user and system times for the shell and for processes run from the shell.

† **trap** [*arg sig* [*sig2* . . .]]
arg is a command to be read and executed when the shell receives signal(s) *sig*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) 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 *arg* is omitted or is −, then the trap(s) for each *sig* are reset to their original values. If *arg* is the null (or empty) string, then this signal is ignored by the shell and by the commands it invokes. If *sig* is **ERR** then *arg* will be executed whenever a command has a non-zero exit status. If *sig* is **DEBUG** then *arg* will be executed after each command. If *sig* is **0** or **EXIT** and the **trap** statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *sig* is **0** or **EXIT** for a **trap** set outside any function then the command *arg* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number. If *action* is −, the shell will reset each **condition** to the default value. If *action* is null (**''**), the shell will ignore each specified *condition* if it arises. Otherwise, the argument *action* will be read and executed by the shell when one of the corresponding conditions arises. The action of the trap will override a previous action (either default action or one explicitly set). The value of **$?** after the trap action completes will be the value it had before the trap was invoked.

The condition can be **EXIT**, **0** (equivalent to **EXIT**) or a signal specified using a symbolic name, without the **SIG** prefix, for example, **HUP**, **INT**, **QUIT**, **TERM**.

The environment in which the shell executes a trap on **EXIT** will be identical to the environment immediately after the last command executed before the trap on **EXIT** was taken.

Each time the trap is invoked, the **action** argument will be processed in a manner equivalent to:

eval "$action"

Signals that were ignored on entry to a non-interactive shell cannot be trapped or reset, although no error need be reported when attempting to do so. An interactive shell may reset or catch signals ignored on entry. Traps will remain in place for a given shell until explicitly changed with another **trap** command.

When a subshell is entered, traps are set to the default actions. This does not imply that the **trap** command cannot be used within the subshell **trap** command cannot be used within the subshell to set new traps.

The **trap** command with no arguments will write to standard output a list of commands associated with each condition. The format is:

trap — **%s %s** . . . <*action*>, <*condition*>

The shell will format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same trapping results. For example:

save_traps=$(trap)
. . .
eval "$save_traps"

If the trap name or number is invalid, a non-zero exit status will be returned; otherwise, **0** will be returned. For both interactive and non-interactive shells, invalid signal names or numbers will not be considered a syntax error and will not cause the shell to abort.

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.

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

†† **typeset** [±**HLRZfilrtux**[*n*]] [*name*[**=***value*]] . . .
Sets attributes and values for shell variables and functions. When **typeset** is invoked inside a function, a new instance of the variables *name* is created. The variables *value* and *type* are restored when the function completes. The following list of attributes may be specified:

–H This flag 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. Leading zeros are removed if the **–Z** flag is also set. The **–R** flag is turned off.

–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** flag is turned off.

–Z Right justify and fill with leading zeros if the first non-blank character is a digit and the **–L** flag has not been 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 flags are **–t**, **–u** and **–x**. The flag **–t** turns on execution tracing for this function. The flag **–u** causes this function to be marked undefined. The **FPATH** variable will be searched to find the function definition when the function is referenced.

The flag **−x** allows the function definition to remain in effect across shell procedures invoked by name.

−i Parameter is an integer. This makes arithmetic faster. If *n* is non-zero it defines the output arithmetic base; otherwise, the first assignment determines the output base.

−l All upper-case characters are converted to lower-case. The upper-case flag, **−u** is turned off.

−r The given *name*s are marked **readonly** and these names cannot be changed by subsequent assignment.

−t Tags the variables. Tags are user definable and have no special meaning to the shell.

−u All lower-case characters are converted to upper-case characters. The lower-case flag, **−l** is turned off.

−x The given *name*s are marked for automatic export to the **environment** of subsequently-executed commands.

The **−i** attribute can not be specified along with **−R**, **−L**, **−Z**, or **−f**.

Using **+** rather than − causes these flags to be turned off. If no *name* arguments are given but flags are specified, a list of *names* (and optionally the *values*) of the *variables* which have these flags set is printed. (Using **+** rather than − keeps the values from being printed.) If no *name*s and flags are given, the *names* and *attributes* of all *variables* are printed.

ulimit [**−HSacdfnstv**] [*limit*]

Set or display a resource limit. The available resources limits are listed below. Many systems do not contain 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** flags 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** or **S** options 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 that one resource is specified, then the limit name and unit is printed before the value.

−a Lists all of the current resource limits.

−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.

−f The number of 512-byte blocks on files written by child processes (files of any size may be read).

−n The number of file descriptors plus 1.

−s The number of K-bytes on the size of the stack area.

−t The number of seconds to be used by each process.

−v The number of K-bytes for virtual memory.

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** flag produces symbolic output.

unalias *name***. . .**

The *alias*es given by the list of *name*s are removed from the *alias* list.

unset [**−f**] *name* . . .

The variables given by the list of *name*s are unassigned, that is, their values and attributes are erased. **readonly** variables cannot be unset. If the **−f**, flag is set, then the names refer to *function* names. Unsetting **ERRNO**, **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 process waited for. See **Jobs** for a description of the format of *job*.

whence [**−pv**] *name* . . .

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

The **−v** flag produces a more verbose report.

The **−p** flag does a path search for *name* even if name is an alias, a function, or a reserved word.

Invocation

If the shell is invoked by **exec**(2), and the first character of argument zero (**$0**) is −, then the shell is assumed to be a **login** shell and commands are read from **/etc/profile** and then from either **.profile** in the current directory or **$HOME/.profile**, if either file exists. Next, commands are read from the file named by performing parameter substitution on the value of the environment variable **ENV** if the file exists. If the **−s** flag is not present and *arg* is, then a path search is performed on the first *arg* to determine the name of the script to execute. The script *arg* must have read 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 builtin command or function. Commands are then read as described below; the following flags are interpreted by the shell when it is invoked:

−c *string* If the **−c** flag is present then commands are read from *string*.

−s If the **−s** flag is present or if no arguments remain then commands are read from the standard input. Shell output, except for the output of the **Special Commands** listed above, is written to file descriptor 2.

−i If the **−i** flag is present or if the shell input and output are attached to a terminal (as told by **ioctl**(2)) 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 If the −**r** flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the **set** command above.

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:

- changing directory (see **cd**(1))
- setting the value of **SHELL**, **ENV**, or **PATH**
- specifying path or command names containing **/**
- redirecting output (**>**, **>|**, **<>**, and **>>**)
- changing group (see **newgrp**(1)).

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 (that is, **/usr/rbin**) that can be safely invoked by **rksh**.

ERRORS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. Otherwise, the shell returns the exit status of the last command executed (see also the **exit** command above). If the shell is being used non-interactively then execution of the shell file is abandoned. 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.

For a non-interactive shell, an error condition encountered by a special built-in or other type of utility will cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

Error	Special Built-in	Other Utilities
Shell language syntax error	will exit	will exit
Utility syntax error (option or operand error)	will exit	will not exit
Redirection error	will exit	will not exit
Variable assignment error	will exit	will not exit
Expansion error	will exit	will exit
Command not found	n/a	may exit
Dot script not found	will exit	n/a

An expansion error is one that occurs when the shell expansions are carried out (for example, **${x!y}**, because **!** is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as “will (may) exit” occur in a subshell, the subshell will (may) exit with a non-zero status, but the script containing the subshell will not exit because of the error.

In all of the cases shown in the table, an interactive shell will write a diagnostic message to standard error without exiting.

EXIT STATUS

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities is documented in this section. The exit status of the standard utilities is documented in their respective sections.

If a command is not found, the exit status will be **127**. If the command name is found, but it is not an executable utility, the exit status will be **126**. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status will be greater than zero.

When reporting the exit status with the special parameter **?**, the shell will report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal will be reported as greater than **128**.

FILES

/etc/profile
/etc/suid_profile
$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

cat(1), **chmod**(1), **cut**(1), **echo**(1), **env**(1), **getoptcvt**(1), **newgrp**(1), **paste**(1), **ps**(1), **shell_builtins**(1), **stty**(1), **vi**(1), **dup**(2), **exec**(2), **fork**(2), **ioctl**(2), **lseek**(2), **pipe**(2), **umask**(2), **ulimit**(2), **wait**(2), **rand**(3C), **signal**(3C), **a.out**(4), **profile**(4), **environ**(5), **signal**(5)

Morris I. Bolsky and David G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.

NOTES

If a command which is a *tracked alias* 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 **–t** option of the **alias** command to correct this situation.

Some very old shell scripts contain a ^ as a synonym for the pipe character | .

Using the **fc** 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 functions defined in the file.