NAME

ksh, **rksh** — public domain Korn shell

SYNOPSIS

ksh [**-+abCefhiklmnpruvXx**] [**-+o** *option*] [**-c** *string* | **-s** | *file* [*argument* ...]]

DESCRIPTION

ksh is a command interpreter intended for both interactive and shell script use. Its command language is a superset of the **sh**(1) shell language.

The options are as follows:

-c *string*
: **ksh** will execute the command(s) contained in *string*.

-i
: Interactive shell. A shell is "interactive" if this option is used or if both standard input and standard error are attached to a tty(4). An interactive shell has job control enabled, ignores the SIGINT, SIGQUIT, and SIGTERM signals, and prints prompts before reading input (see the PS1 and PS2 parameters). For non-interactive shells, the **trackall** option is on by default (see the **set** command below).

-l
: Login shell. If the basename the shell is called with (i.e. argv[0]) starts with '-' or if this option is used, the shell is assumed to be a login shell and the shell reads and executes the contents of /etc/profile and $HOME/.profile if they exist and are readable.

-p
: Privileged shell. A shell is "privileged" if this option is used or if the real user ID or group ID does not match the effective user ID or group ID (see getuid(2) and . -- getgid(2)) A privileged shell does not process $HOME/.profile nor the ENV parameter (see below). Instead, the file /etc/suid_profile is processed. Clearing the privileged option causes the shell to set its effective user ID (group ID) to its real user ID (group ID).

-r
: Restricted shell. A shell is "restricted" if this option is used; if the basename the shell was invoked with was "rksh"; or if the SHELL parameter is set to "rksh". The following restrictions come into effect after the shell processes any profile and ENV files:

  - The **cd** command is disabled.
  - The SHELL, ENV, and PATH parameters cannot be changed.
  - Command names can't be specified with absolute or relative paths.
  - The **-p** option of the built-in command **command** can't be used.
  - Redirections that create files can't be used (i.e. '**>**', '**>|**', '**>>**', '**<>**').

-s
: The shell reads commands from standard input; all non-option arguments are positional parameters.

In addition to the above, the options described in the **set** built-in command can also be used on the command line: both [**-+abCefhkmnuvXx**] and [**-+o** *option*] can be used for single letter or long options, respectively.

If neither the **-c** nor the **-s** option is specified, the first non-option argument specifies the name of a file the shell reads commands from. If there are no non-option arguments, the shell reads commands from the standard input. The name of the shell (i.e. the contents of $0) is determined as follows: if the **-c** option is used and there is a non-option argument, it is used as the name; if commands are being read from a file, the file is used as the name; otherwise, the basename the shell was called with (i.e. argv[0]) is used.

If the ENV parameter is set when an interactive shell starts (or, in the case of login shells, after any profiles are processed), its value is subjected to parameter, command, arithmetic, and tilde ('~')

substitution and the resulting file (if any) is read and executed. In order to have an interactive (as opposed to login) shell process a startup file, `ENV` may be set and exported (see below) in `$HOME/.profile` - future interactive shell invocations will process any file pointed to by `$ENV`:

```
export ENV=$HOME/.kshrc
```

`$HOME/.kshrc` is then free to specify instructions for interactive shells. For example, the global configuration file may be sourced:

```
. /etc/ksh.kshrc
```

The above strategy may be employed to keep setup procedures for login shells in `$HOME/.profile` and setup procedures for interactive shells in `$HOME/.kshrc`. Of course, since login shells are also interactive, any commands placed in `$HOME/.kshrc` will be executed by login shells too.

The exit status of the shell is 127 if the command file specified on the command line could not be opened, or non-zero if a fatal syntax error occurred during the execution of a script. In the absence of fatal errors, the exit status is that of the last command executed, or zero, if no command is executed.

Command syntax

The shell begins parsing its input by breaking it into *words*. Words, which are sequences of characters, are delimited by unquoted whitespace characters (space, tab, and newline) or meta-characters (‘`<`’, ‘`>`’, ‘`|`’, ‘`;`’, ‘`(`’, ‘`)`’, and ‘`&`’). Aside from delimiting words, spaces and tabs are ignored, while newlines usually delimit commands. The meta-characters are used in building the following *tokens*: ‘**`<`**’, ‘**`<&`**’, ‘**`<<`**’, ‘**`>`**’, ‘**`>&`**’, ‘**`>>`**’, etc. are used to specify redirections (see Input/output redirection below); ‘`|`’ is used to create pipelines; ‘`|&`’ is used to create co-processes (see Co-processes below); ‘`;`’ is used to separate commands; ‘`&`’ is used to create asynchronous pipelines; ‘`&&`’ and ‘`||`’ are used to specify conditional execution; ‘`;;`’ is used in**`case`** statemen ts; `(( .. ))` is used in arithmetic expressions; and lastly, `( .. )` is used to create subshells.

Whitespace and meta-characters can be quoted individually using a backslash (‘’), or in groups using double (‘”’) or single (‘’’) quotes. The following characters are also treated specially by the shell and must be quoted if they are to represent themselves: ‘\’, ‘"’, ‘’’, ‘`#`’, ‘`$`’, ‘‘’, ‘’, ‘`{`’, ‘`}`’, ‘`*`’, ‘`?`’, and ‘`[`’. The first three of these are the above mentioned quoting characters (see Quoting below); ‘`#`’, if used at the beginning of a word, introduces a comment — everything after the ‘`#`’ up to the nearest newline is ignored; ‘`$`’ is used to introduce parameter, command, and arithmetic substitutions (see Substitution below); ‘‘’ introduces an old-style command substitution (see Substitution below); ‘’ begins a directory expansion (see Tilde expansion below); ‘`{`’ and ‘`}`’ delimit csh(1)Ns -style alternations (see Brace expansion below); and finally, ‘`*`’, ‘`?`’, and ‘`[`’ are used in file name generation (see File name patterns below).

As words and tokens are parsed, the shell builds commands, of which there are two basic types: *simple-commands*, typically programs that are executed, and *compound-commands*, such as **`for`** and **`if`** statements, grouping constructs, and function definitions.

A simple-command consists of some combination of parameter assignments (see Parameters below), input/output redirections (see Input/output redirections below), and command words; the only restriction is that parameter assignments come before any command words. The command words, if any, define the command that is to be executed and its arguments. The command may be a shell built-in command, a function, or an external command (i.e. a separate executable file that is located using the `PATH` parameter; see Command execution below).

All command constructs have an exit status. For external commands, this is related to the status returned by `wait`(2) (if the command could not be found, the exit status is 127; if it could not be executed, the exit status is 126). The exit status of other command constructs (built-in commands,

functions, compound-commands, pipelines, lists, etc.) are all well-defined and are described where the construct is described. The exit status of a command consisting only of parameter assignments is that of the last command substitution performed during the parameter assignment or 0 if there were no command substitutions.

Commands can be chained together using the '|' token to form pipelines, in which the standard output of each command but the last is piped (see pipe(2)) to the standard input of the following command. The exit status of a pipeline is that of its last command. A pipeline may be prefixed by the '`!`' reserved word, which causes the exit status of the pipeline to be logically complemented: if the original status was 0, the complemented status will be 1; if the original status was not 0, the complemented status will be 0.

Lists of commands can be created by separating pipelines by any of the following tokens: '`&&`', '`||`', '`&`', '`|&`', and '`;`'. The first two are for conditional execution: "*cmd1* && *cmd2*" executes *cmd2* only if the exit status of *cmd1* is zero; '`||`' is the opposite — *cmd2* is executed only if the exit status of *cmd1* is non-zero. '`&&`' and '`||`' have equal precedence which is higher than that of '`&`', '`|&`', and '`;`', which also have equal precedence. The '`&&`' and '`||`' operators are left-associative. For example, both of these commands will print only bar:

```
$ false && echo foo || echo bar
$ true || echo foo && echo bar
```

The '`&`' token causes the preceding command to be executed asynchronously; that is, the shell starts the command but does not wait for it to complete (the shell does keep track of the status of asynchronous commands; see Job control below). When an asynchronous command is started when job control is disabled (i.e. in most scripts), the command is started with signals `SIGINT` and `SIGQUIT` ignored and with input redirected from `/dev/null` (however, redirections specified in the asynchronous command have precedence). The '`|&`' operator starts a co-process which is a special kind of asynchronous process (see Co-processes below). A command must follow the '`&&`' and '`||`' operators, while it need not follow '`&`', '`|&`', or '`;`'. The exit status of a list is that of the last command executed, with the exception of asynchronous lists, for which the exit status is 0.

Compound commands are created using the following reserved words. These words are only recognized if they are unquoted and if they are used as the first word of a command (i.e. they can't be preceded by parameter assignments or redirections):

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

Note: Some shells (but not this one) execute control structure commands in a subshell when one or more of their file descriptors are redirected, so any environment changes inside them may fail. To be portable, the **exec** statement should be used instead to redirect file descriptors before the control structure.

In the following compound command descriptions, command lists (denoted as *list*) that are followed by reserved words must end with a semicolon, a newline, or a (syntactically correct) reserved word. For example, the following are all valid:

```
$ { echo foo; echo bar; }
$ { echo foo; echo bar<newline> }
$ { { echo foo; echo bar; } }
```

This is not valid:

```
$ { echo foo; echo bar }
```

(*list*)
Execute *list* in a subshell. There is no implicit way to pass environment changes from a subshell back to its parent.

{ *list*; }
Compound construct; *list* is executed, but not in a subshell. Note that '{' and '}' are reserved words, not meta-characters.

case *word* **in** [[(] *pattern* [| *pattern*] ...)*list* ;;] ... **esac**
The **case** statement attempts to match *word* against a specified *pattern*; the *list* associated with the first successfully matched pattern is executed. Patterns used in **case** statements are the same as those used for file name patterns except that the restrictions regarding '.' and '/' are dropped. Note that any unquoted space before and after a pattern is stripped; any space within a pattern must be quoted. Both the word and the patterns are subject to parameter, command, and arithmetic substitution, as well as tilde substitution. For historical reasons, open and close braces may be used instead of **in** and **esac** e.g. **case $foo { *) echo bar; }**. The exit status of a**case** statemen t is that of the executed *list*; if no *list* is executed, the exit status is zero.

for *name* [**in** *word* ...]; **do** *list*; **done**
For each *word* in the specified word list, the parameter *name* is set to the word and *list* is executed. If**in** is not used to specify a w ord list, the positional parameters ($1, $2, etc.) are used instead. For historical reasons, open and close braces may be used instead of **do** and **done** e.g. **for i; { echo $i; }**. The exit status of a**for** statemen t is the last exit status of *list*; if *list* is never executed, the exit status is zero.

if *list*; **then** *list*; [**elif** *list*; **then** *list*;] ... [**else** *list*;] **fi**
If the exit status of the first *list* is zero, the second *list* is executed; otherwise, the *list* following the **elif**, if any, is executed with similar consequences. If all the lists following the **if** and **elif**s fail (i.e. exit with non-zero status), the *list* following the **else** is executed. The exit status of an**if** statemen t is that of non-conditional *list* that is executed; if no non-conditional *list* is executed, the exit status is zero.

select *name* [**in** *word* ...]; **do** *list*; **done**
The **select** statement provides an automatic method of presenting the user with a menu and selecting from it. An enumerated list of the specified *word*(s) is printed on standard error, followed by a prompt (PS3: normally '#? '). A n umber corresponding to one of the enumerated words is then read from standard input, *name* is set to the selected word (or unset if the selection is not valid), REPLY is set to what was read (leading/trailing space is stripped), and *list* is executed. If a blank line (i.e. zero or more IFS characters) is entered, the menu is reprinted without executing *list*.

When *list* completes, the enumerated list is printed if REPLY is NULL, the prompt is printed, and so on. This process continues until an end-of-file is read, an interrupt is received, or a **break** statement is executed inside the loop. If "in word ..." is omitted, the positional parameters are used (i.e. $1, $2, etc.). For historical reasons, open and close braces may be used instead of **do** and **done** e.g. **select i; { echo $i; }**. The exit status of a **select** statement is zero if a **break** statement is used to exit the loop, non-zero otherwise.

until *list*; **do** *list*; **done**
This works like **while**, except that the body is executed only while the exit status of the first *list* is non-zero.

while *list*; **do** *list*; **done**
A **while** is a pre-checked loop. Its body is executed as often as the exit status of the first *list* is zero. The exit status of a **while** statement is the last exit status of the *list* in the body of the loop; if the body is not executed, the exit status is zero.

function *name* { *list*; }
Defines the function *name* (see Functions below). Note that redirections specified after a function definition are performed whenever the function is executed, not when the function definition is executed.

name() *command*
Mostly the same as **function** (see Functions below).

time [**-p**] [*pipeline*]
The **time** reserved word is described in the Command execution section.

((*expression* **))**
The arithmetic expression *expression* is evaluated; equivalent to **let** *expression* (see Arithmetic expressions and the **let** command, below).

[[*expression* **]]**
Similar to the **test** and **[** ... **]** commands (described later), with the following exceptions:

- Field splitting and file name generation are not performed on arguments.
- The **-a** (AND) and **-o** (OR) operators are replaced with '**&&**' and '**||**', respectively.
- Operators (e.g. '**-f**', '=', '!') must be unquoted.
- The second operand of the '!=' and '=' expressions are patterns (e.g. the comparison **[[foobar = f*r]]**succeeds).
- There are two additional binary operators, '**<**' and '**>**', which return true if their first string operand is less than, or greater than, their second string operand, respectively.
- The single argument form of **test**, which tests if the argument has a non-zero length, is not valid; explicit operators must always be used e.g. instead of [*str*] use [[-n *str*]].
- Parameter, command, and arithmetic substitutions are performed as expressions are evaluated and lazy expression evaluation is used for the '**&&**' and '**||**' operators. This means that in the following statement, **$(< foo)** is evaluated if and only if the file **foo** exists and is readable:

```
$ [[ -r foo && $(< foo) = b*r ]]
```

Quoting

Quoting is used to prevent the shell from treating characters or words specially. There are three methods of quoting. First, '\' quotes the following character, unless it is at the end of a line, in which case both the '\' and the newline are stripped. Second, a single quote (''') quotes everything up to the next single quote (this may span lines). Third, a double quote ('') quotes all characters, except '$', '`' and '\', up to the next unquoted double quote. '$' and '`' inside double quotes have their usual meaning (i.e. parameter, command, or arithmetic substitution) except no

field splitting is carried out on the results of double-quoted substitutions. If a '\' inside a double-quoted string is followed by '\', '$', '`', or '"', it is replaced by the second character; if it is followed by a newline, both the '\' and the newline are stripped; otherwise, both the '\' and the character following are unchanged.

Aliases

There are two types of aliases: normal command aliases and tracked aliases. Command aliases are normally used as a short hand for a long or often used command. The shell expands command aliases (i.e. substitutes the alias name for its value) when it reads the first word of a command. An expanded alias is re-processed to check for more aliases. If a command alias ends in a space or tab, the following word is also checked for alias expansion. The alias expansion process stops when a word that is not an alias is found, when a quoted word is found, or when an alias word that is currently being expanded is found.

The following command aliases are defined automatically by the shell:

autoload='typeset -fu'
functions='typeset -f'
hash='alias -t'
history='fc -l'
integer='typeset -i'
local='typeset'
login='exec login'
nohup='nohup '
r='fc -s'
stop='kill -STOP'
type='whence -v'

Tracked aliases allow the shell to remember where it found a particular command. The first time the shell does a path search for a command that is marked as a tracked alias, it saves the full path of the command. The next time the command is executed, the shell checks the saved path to see that it is still valid, and if so, avoids repeating the path search. Tracked aliases can be listed and created using **alias -t**. Note that changing the `PATH` parameter clears the saved paths for all tracked aliases. If the **trackall** option is set (i.e. **set -o trackall** or **set -h**), the shell tracks all commands. This option is set automatically for non-interactive shells. For interactive shells, only the following commands are automatically tracked: cat(1), cc(1), chmod(1), cp(1), date(1), ed(1), emacs(1), grep(1), ls(1), mail(1), make(1), mv(1), pr(1), rm(1), sed(1), sh(1), vi(1), and who(1).

Substitution

The first step the shell takes in executing a simple-command is to perform substitutions on the words of the command. There are three kinds of substitution: parameter, command, and arithmetic. Parameter substitutions, which are described in detail in the next section, take the form $*name* or ${ . . .}; command substitutions take the form $(*command*) or `*command*`; and arithmetic substitutions take the form $((*expression*)).

If a substitution appears outside of double quotes, the results of the substitution are generally subject to word or field splitting according to the current value of the `IFS` parameter. The `IFS` parameter specifies a list of characters which are used to break a string up into several words; any characters from the set space, tab, and newline that appear in the `IFS` characters are called "IFS whitespace". Sequences of one or more`IFS` whitespace c haracters, in combination with zero or one non-`IFS` whitespace characters, delimit a field. As a special case, leading and trailing `IFS` whitespace is stripped (i.e. no leading or trailing empty field is created by it); leading non-`IFS` whitespace does create an empty field.

Example: If `IFS` is set to "<space>:", and VAR is set to "<space>A<space>:<space><space>B::D", the substitution for $VAR results in four fields: 'A', 'B', '' (an empty field), and 'D'. Note that if the `IFS` parameter is set to the `NULL` string, no field splitting is done; if the parameter is unset, the default value of space, tab, and newline is used.

Also, note that the field splitting applies only to the immediate result of the substitution. Using the previous example, the substitution for $VAR:E results in the fields: 'A', 'B', '', and 'D:E', not 'A', 'B', '', 'D', and 'E'. This behavior is POSIX compliant, but incompatible with some other shell implementations which do field splitting on the word which contained the substitution or use `IFS` as a general whitespace delimiter.

The results of substitution are, unless otherwise specified, also subject to brace expansion and file name expansion (see the relevant sections below).

A command substitution is replaced by the output generated by the specified command, which is run in a subshell. For $(*command*) substitutions, normal quoting rules are used when *command* is parsed; however, for the '*command*' form, a '\' followed by any of '$', '`', or '\' is stripped (a '\' followed by any other character is unchanged). As a special case in command substitutions, a command of the form <*file* is interpreted to mean substitute the contents of *file*. Note that**$(< foo)** has the same effect as **$(cat foo)**, but it is carried out more efficiently because no process is started.

Arithmetic substitutions are replaced by the value of the specified expression. For example, the command **echo $((2+3*4))** prints 14. See Arithmetic expressions for a description of an expression.

Parameters

Parameters are shell variables; they can be assigned values and their values can be accessed using a parameter substitution. A parameter name is either one of the special single punctuation or digit character parameters described below, or a letter followed by zero or more letters or digits ('_' counts as a letter). The latter form can be treated as arrays by appending an array index of the form [*expr*] where *expr* is an arithmetic expression. Parameter substitutions take the form $*name*, ${*name*}, or ${*name*[*expr*]} where *name* is a parameter name. If *expr* is a literal '@' then the named array is expanded using the same quoting rules as '$@', while if *expr* is a literal '*' then the named array is expanded using the same quoting rules as '$*'. If substitution is performed on a parameter (or an array parameter element) that is not set, a null string is substituted unless the **nounset** option (**set -o nounset**or **set -u**) is set, in whic h case an error occurs.

Parameters can be assigned values in a number of ways. First, the shell implicitly sets some parameters like '#', `PWD`, and '$'; this is the only way the special single character parameters are set. Second, parameters are imported from the shell's environment at startup. Third, parameters can be assigned values on the command line: for example, **FOO=bar** sets the parameter "FOO" to "bar"; multiple parameter assignments can be given on a single command line and they can be followed by a simple-command, in which case the assignments are in effect only for the duration of the command (such assignments are also exported; see below for the implications of this). Note that both the parameter name and the '=' must be unquoted for the shell to recognize a parameter assignment. The fourth way of setting a parameter is with the **export**, **readonly**, and **typeset** commands; see their descriptions in the Command execution section. Fifth, **for** and **select** loops set parameters as well as the **getopts**, **read**, and **set -A** commands. Lastly, parameters can be assigned values using assignment operators inside arithmetic expressions (see Arithmetic expressions below) or using the ${*name*=*value*} form of the parameter substitution (see below).

Parameters with the export attribute (set using the **export** or **typeset -x** commands, or by parameter assignments followed by simple commands) are put in the environment (see environ(7)) of commands run by the shell as *name*=*value* pairs. The order in which parameters appear in the

environment of a command is unspecified. When the shell starts up, it extracts parameters and their values from its environment and automatically sets the export attribute for those parameters.

Modifiers can be applied to the ${*name*} form of parameter substitution:

${*name*:-*word*}
: If *name* is set and not NULL, it is substituted; otherwise, *word* is substituted.

${*name*:+*word*}
: If *name* is set and not NULL, *word* is substituted; otherwise, nothing is substituted.

${*name*:=*word*}
: If *name* is set and not NULL, it is substituted; otherwise, it is assigned *word* and the resulting value of *name* is substituted.

${*name*:?*word*}
: If *name* is set and not NULL, it is substituted; otherwise, *word* is printed on standard error (preceded by *name*:) and an error occurs (normally causing termination of a shell script, function, or script sourced using the '.' built-in). If *word* is omitted, the string "parameter null or not set" is used instead.

In the above modifiers, the ':' can be omitted, in which case the conditions only depend on *name* being set (as opposed to set and not NULL). If*word* is needed, parameter, command, arithmetic, and tilde substitution are performed on it; if *word* is not needed, it is not evaluated.

The following forms of parameter substitution can also be used:

${#*name*}
: The number of positional parameters if *name* is '*', '@', or not specified; otherwise the length of the string value of parameter *name*.

${#*name*[*]}
${#*name*[@]}
: The number of elements in the array *name*.

${*name*#*pattern*}
${*name*##*pattern*}
: If *pattern* matches the beginning of the value of parameter *name*, the matched text is deleted from the result of substitution. A single '#' results in the shortest match, and two of them result in the longest match.

${*name*%*pattern*}
${*name*%%*pattern*}
: Like ${..#..} substitution, but it deletes from the end of the value.

The following special parameters are implicitly set by the shell and cannot be set directly using assignments:

!
: Process ID of the last background process started. If no background processes have been started, the parameter is not set.

#
: The number of positional parameters ($1, $2, etc.).

$
: The PID of the shell, or the PID of the original shell if it is a subshell. Do *NOT* use this mechanism for generating temporary file names; see mktemp(1) instead.

-
: The concatenation of the current single letter options (see the **set** command below for a list of options).

`?` The exit status of the last non-asynchronous command executed. If the last command was killed by a signal, **`$?`** is set to 128 plus the signal number.

`0` The name of the shell, determined as follows: the first argument to **`ksh`** if it was invoked with the **`-c`** option and arguments were given; otherwise the *`file`* argument, if it was supplied; or else the basename the shell was invoked with (i.e. `argv[0]`). `$0` is also set to the name of the current script or the name of the current function, if it was defined with the **`function`** keyword (i.e. a Korn shell style function).

`1 ... 9` The first nine positional parameters that were supplied to the shell, function, or script sourced using the '.' built-in. Further positional parameters may be accessed using ${*number*}.

`*` All positional parameters (except parameter 0) i.e. $1, $2, $3, ... If used outside of double quotes, parameters are separate words (which are subjected to word splitting); if used within double quotes, parameters are separated by the first character of the `IFS` parameter (or the empty string if `IFS` is `NULL`).

`@` Same as **`$`***, unless it is used inside double quotes, in which case a separate word is generated for each positional parameter. If there are no positional parameters, no word is generated. **`$@`** can be used to access arguments, verbatim, without losing `NULL` arguments or splitting arguments with spaces.

The following parameters are set and/or used by the shell:

`_` (underscore)
When an external command is executed by the shell, this parameter is set in the environment of the new process to the path of the executed command. In interactive use, this parameter is also set in the parent shell to the last word of the previous command. When `MAILPATH` messages are evaluated, this parameter contains the name of the file that changed (see the `MAILPATH` parameter, below).

`CDPATH` Search path for the **`cd`** built-in command. It works the same way as `PATH` for those directories not beginning with '`/`' or '`.`' in**`cd`** commands. Note that if `CDPATH` is set and does not contain '.' or contains an empty path, the current directory is not searched. Also, the **`cd`** built-in command will display the resulting directory when a match is found in any search path other than the empty path.

`COLUMNS` Set to the number of columns on the terminal or window. Currently set to the "cols" value as reported by `stty`(1) if that value is non-zero. This parameter is used by the interactive line editing modes, and by the **`select`**, **`set -o`**, and **`kill -l`** commands to format information columns.

`EDITOR` If the `VISUAL` parameter is not set, this parameter controls the command-line editing mode for interactive shells. See the `VISUAL` parameter below for how this works.

Note: traditionally, `EDITOR` was used to specify the name of an (old-style) line editor, such as ed(1), and `VISUAL` was used to specify a (new-style) screen editor, such as vi(1). Hence if `VISUAL` is set, it overrides `EDITOR`.

`ENV` If this parameter is found to be set after any profile files are executed, the expanded value is used as a shell startup file. It typically contains function and alias definitions.

`EXECSHELL` If set, this parameter is assumed to contain the shell that is to be used to execute commands that `execve`(2) fails to execute and which do not start with a "#!*`shell`*" sequence.

`FCEDIT` The editor used by the **`fc`** command (see below).

`FPATH` Like `PATH`, but used when an undefined function is executed to locate the file defining the function. It is also searched when a command can't be found using `PATH`. See Functions below for more information.

`HISTFILE` The name of the file used to store command history. When assigned to, history is loaded from the specified file. Also, several invocations of the shell running on the same machine will share history if their `HISTFILE` parameters all point to the same file.

Note: If `HISTFILE` isn't set, no history file is used. This is different from the original Korn shell, which uses `$HOME/.sh_history`.

`HISTSIZE` The number of commands normally stored for history. The default is 500.

`HOME` The default directory for the **`cd`** command and the value substituted for an unqualified `~` (see Tilde expansion below).

`IFS` Internal field separator, used during substitution and by the **`read`** command, to split values into distinct arguments; normally set to space, tab, and newline. See Substitution above for details.

Note: This parameter is not imported from the environment when the shell is started.

`KSH_VERSION`
The version of the shell and the date the version was created (read-only).

`LINENO` The line number of the function or shell script that is currently being executed.

`LINES` Set to the number of lines on the terminal or window.

`MAIL` If set, the user will be informed of the arrival of mail in the named file. This parameter is ignored if the `MAILPATH` parameter is set.

`MAILCHECK` How often, in seconds, the shell will check for mail in the file(s) specified by `MAIL` or `MAILPATH`. If set to 0, the shell checks before each prompt. The default is 600 (10 minutes).

`MAILPATH` A list of files to be checked for mail. The list is colon separated, and each file may be followed by a '`?`' and a message to be printed if new mail has arrived. Command, parameter, and arithmetic substitution is performed on the message and, during substitution, the parameter **`$_`** contains the name of the file. The default message is "you have mail in $_".

`OLDPWD` The previous working directory. Unset if**`cd`** has not successfully c hanged directories since the shell started, or if the shell doesn't know where it is.

`OPTARG` When using **`getopts`**, it contains the argument for a parsed option, if it requires one.

`OPTIND` The index of the next argument to be processed when using **`getopts`**. Assigning 1 to this parameter causes **`getopts`** to process arguments from the beginning the next time it is invoked.

`PATH` A colon separated list of directories that are searched when looking for commands and files sourced using the '.' command (see below). An empty string resulting from a leading or trailing colon, or two adjacent colons, is treated as a '.' (the current directory).

`POSIXLY_CORRECT`
If set, this parameter causes the **`posix`** option to be enabled. See POSIX mode below.

`PPID` The process ID of the shell's parent (read-only).

`PS1` The primary prompt for interactive shells. Parameter, command, and arithmetic substitutions are performed, and the prompt string can be customised using backslash-escaped special characters.

Note that since the command-line editors try to figure out how long the prompt is (so they know how far it is to the edge of the screen), escape codes in the prompt tend to mess things up. You can tell the shell not to count certain sequences (such as escape codes) by using the `\[...\]` substitution (see below) or by prefixing your prompt with a non-printing character (such as control-A) followed by a carriage return and then delimiting the escape codes with this non-printing character. By the way, don't blame me for this hack; it's in the original **ksh**.

The default prompt is '$ ' for non-root users, '# ' for root. If **ksh** is invoked by root and `PS1` does not contain a '#' character, the default value will be used even if `PS1` already exists in the environment.

The following backslash-escaped special characters can be used to customise the prompt:

`\a`	Insert an ASCII bell character.
`\d`	The current date, in the format "Day Month Date" for example "Wed Nov 03".
`\D{`*format*`}`	The current date, with *format* converted by strftime(3). The braces must be specified.
`\e`	Insert an ASCII escape character.
`\h`	The hostname, minus domain name.
`\H`	The full hostname, including domain name.
`\j`	Current number of jobs running (see Job control below).
`\l`	The controlling terminal.
`\n`	Insert a newline character.
`\r`	Insert a carriage return character.
`\s`	The name of the shell.
`\t`	The current time, in 24-hour HH:MM:SS format.
`\T`	The current time, in 12-hour HH:MM:SS format.
`\@`	The current time, in 12-hour HH:MM:SS AM/PM format.
`\A`	The current time, in 24-hour HH:MM format.
`\u`	The current user's username.
`\v`	The current version of **ksh**.
`\V`	Like 'v', but more verbose.
`\w`	The current working directory. `$HOME` is abbreviated as '~'.
`\W`	The basename of the current working directory. `$HOME` is abbreviated as '~'.
`\!`	The current history number. An unescaped '`!`' will produce the current history number too, as per the POSIX specification. A literal '`!`' can be put in the prompt by placing '`!!`' in`PS1`.
`\#`	The current command number. This could be different to the current history number, if `HISTFILE` contains a history list from a previous session.
`\$`	The default prompt i.e. '# ' if the effective UID is 0, otherwise '$ '. Since the shell interprets '$' as a special character within double quotes, it is safer in this case to escape the backslash than to try quoting it.

nnn	The octal character *nnn*.
\\	Insert a single backslash character.
\[	Normally the shell keeps track of the number of characters in the prompt. Use of this sequence turns off that count.
\]	Use of this sequence turns the count back on.

Note that the backslash itself may be interpreted by the shell. Hence, to set PS1 either escape the backslash itself, or use double quotes. The latter is more practical:

```
PS1="\u "
```

This is a more complex example, which does not rely on the above backslash-escaped sequences. It embeds the current working directory, in reverse video, in the prompt string:

```
x=$(print \\001)
PS1="$x$(print \\r)$x$(tput so)$x\$PWD$x$(tput se)$x> "
```

PS2 Secondary prompt string, by default '> ', used when more input is needed to complete a command.

PS3 Prompt used by the **select** statement when reading a menu selection. The default is '#? '.

PS4 Used to prefix commands that are printed during execution tracing (see the **set -x** command below). Parameter, command, and arithmetic substitutions are performed before it is printed. The default is '+ '.

PWD The current working directory. May be unset or NULL if the shell doesn't know where it is.

RANDOM A random number generator. Every time RANDOM is referenced, it is assigned the next random number in the range 0-32767. By default, arc4random(3) is used to produce values. If the variable RANDOM is assigned a value, the value is used as the seed to srand_deterministic(3) and subsequent references of RANDOM produce a predictable sequence.

REPLY Default parameter for the **read** command if no names are given. Also used in**select** loops to store the value that is read from standard input.

SECONDS The number of seconds since the shell started or, if the parameter has been assigned an integer value, the number of seconds since the assignment plus the value that was assigned.

TMOUT If set to a positive integer in an interactive shell, it specifies the maximum number of seconds the shell will wait for input after printing the primary prompt (PS1). If the time is exceeded, the shell exits.

TMPDIR The directory temporary shell files are created in. If this parameter is not set, or does not contain the absolute path of a writable directory, temporary files are created in /tmp.

VISUAL If set, this parameter controls the command-line editing mode for interactive shells. If the last component of the path specified in this parameter contains the string "vi", "emacs", or "gmacs", the vi(1), emacs, or gmacs (Gosling emacs) editing mode is enabled, respectively. See also theEDITOR parameter, abo ve.

Tilde expansion

Tilde expansion, which is done in parallel with parameter substitution, is done on words starting with an unquoted ‘~’. The characters following the tilde, up to the first ‘/’, if any, are assumed to be a login name. If the login name is empty, ‘+’, or ‘-’, the value of the `HOME`, `PWD`, or `OLDPWD` parameter is substituted, respectively. Otherwise, the password file is searched for the login name, and the tilde expression is substituted with the user's home directory. If the login name is not found in the password file or if any quoting or parameter substitution occurs in the login name, no substitution is performed.

In parameter assignments (such as those preceding a simple-command or those occurring in the arguments of **alias**, **export**, **readonly**, and **typeset**), tilde expansion is done after any assignment (i.e. after the equals sign) or after an unquoted colon (‘:’); login names are also delimited by colons.

The home directory of previously expanded login names are cached and re-used. The **alias -d** command may be used to list, change, and add to this cache (e.g. **alias -d fac=/usr/local/facilities; cd ~fac/bin**).

Brace expansion (alternation)

Brace expressions take the following form:

prefix{*str1*,...,*strN*}*suffix*

The expressions are expanded to *N* words, each of which is the concatenation of *prefix*, *str*i, and *suffix* (e.g. “a{c,b{X,Y},d}e” expands to four words: “ace”, “abXe”, “abYe”, and “ade”). As noted in the example, brace expressions can be nested and the resulting words are not sorted. Brace expressions must contain an unquoted comma (‘,’) for expansion to occur (e.g. **{}** and **{foo}** are not expanded). Brace expansion is carried out after parameter substitution and before file name generation.

File name patterns

A file name pattern is a word containing one or more unquoted ‘?’, ‘*’, ‘+’, ‘@’, or ‘!’ characters or “[..]” sequences. Once brace expansion has been performed, the shell replaces file name patterns with the sorted names of all the files that match the pattern (if no files match, the word is left unchanged). The pattern elements have the following meaning:

? Matches any single character.

* Matches any sequence of characters.

[..] Matches any of the characters inside the brackets. Ranges of characters can be specified by separating two characters by a ‘-’ (e.g. “[a0-9]” matches the letter ‘a’ or any digit). In order to represent itself, a ‘-’ must either be quoted or the first or last character in the character list. Similarly, a ‘]’ must be quoted or the first character in the list if it is to represent itself instead of the end of the list. Also, a ‘!’ appearing at the start of the list has special meaning (see below), so to represent itself it must be quoted or appear later in the list.

Within a bracket expression, the name of a *character class* enclosed in ‘[:’ and ‘:]’ stands for the list of all characters belonging to that class. Supported character classes:

```
alnum cntrl lower space
alpha digit print upper
blank graph punct xdigit
```

These match characters using the macros specified in isalnum(3), isalpha(3), and so on. A character class may not be used as an endpoint of a range.

[!..] Like [..], except it matches any character not inside the brackets.

*(*pattern*|...|*pattern*)
Matches any string of characters that matches zero or more occurrences of the specified patterns. Example: The pattern*(foo|bar) matc hes the strings "", "foo", "bar", "foobarfoo", etc.

+(*pattern*|...|*pattern*)
Matches any string of characters that matches one or more occurrences of the specified patterns. Example: The pattern+(foo|bar) matc hes the strings "foo", "bar", "foobar", etc.

?(*pattern*|...|*pattern*)
Matches the empty string or a string that matches one of the specified patterns. Example: The pattern ?(foo|bar) only matches the strings "", "foo", and "bar".

@(*pattern*|...|*pattern*)
Matches a string that matches one of the specified patterns. Example: The pattern @(foo|bar) only matches the strings "foo" and "bar".

!(*pattern*|...|*pattern*)
Matches any string that does not match one of the specified patterns. Examples: The pattern !(foo|bar) matches all strings except "foo" and "bar"; the pattern !(*) matches no strings; the pattern !(?)* matches all strings (think about it).

Unlike most shells, **ksh** never matches '.' and '..'.

Note that none of the above pattern elements match either a period ('.') at the start of a file name or a slash ('/'), even if they are explicitly used in a [..] sequence; also, the names '.' and '..' are never matched, even by the pattern '.*'.

If the **markdirs** option is set, any directories that result from file name generation are marked with a trailing '/'.

Input/output redirection

When a command is executed, its standard input, standard output, and standard error (file descriptors 0, 1, and 2, respectively) are normally inherited from the shell. Three exceptions to this are commands in pipelines, for which standard input and/or standard output are those set up by the pipeline, asynchronous commands created when job control is disabled, for which standard input is initially set to be from /dev/null, and commands for which any of the following redirections have been specified:

> *file*
Standard output is redirected to *file*. If*file* does not exist, it is created; if it does exist, is a regular file, and the **noclobber** option is set, an error occurs; otherwise, the file is truncated. Note that this means the command**cmd < foo > foo** will open *foo* for reading and then truncate it when it opens it for writing, before *cmd* gets a chance to actually read *foo*.

>| *file*
Same as **>**, except the file is truncated, even if the **noclobber** option is set.

>> *file*
Same as **>**, except if *file* exists it is appended to instead of being truncated. Also, the file is opened in append mode, so writes always go to the end of the file (see . -- open(2))

< *file*
Standard input is redirected from *file*, which is opened for reading.

<> *file*
Same as **<**, except the file is opened for reading and writing.

<< *marker*
After reading the command line containing this kind of redirection (called a "here document"), the shell copies lines from the command source into a temporary file until a line matching *marker* is read. When the command is executed, standard input is redirected from the temporary file. If *marker* contains no quoted characters, the contents of the temporary file are processed as if enclosed in double quotes each time the command is executed, so parameter, command, and arithmetic substitutions are performed, along with backslash (‘’) escapes for ‘$’, ‘‘’, ‘\’, and `\newline`. If multiple here documents are used on the same command line, they are saved in order.

<<- *marker*
Same as **<<**, except leading tabs are stripped from lines in the here document.

<& *fd* Standard input is duplicated from file descriptor *fd*. *fd* can be a single digit, indicating the number of an existing file descriptor; the letter ‘p’, indicating the file descriptor associated with the output of the current co-process; or the character ‘-’, indicating standard input is to be closed.

>& *fd* Same as **<&**, except the operation is done on standard output.

In any of the above redirections, the file descriptor that is redirected (i.e. standard input or standard output) can be explicitly given by preceding the redirection with a single digit. Parameter, command, and arithmetic substitutions, tilde substitutions, and (if the shell is interactive) file name generation are all performed on the *file*, *marker*, and *fd* arguments of redirections. Note, however, that the results of any file name generation are only used if a single file is matched; if multiple files match, the word with the expanded file name generation characters is used. Note that in restricted shells, redirections which can create files cannot be used.

For simple-commands, redirections may appear anywhere in the command; for compound-commands (**if** statements, etc.), any redirections must appear at the end. Redirections are processed after pipelines are created and in the order they are given, so the following will print an error with a line number prepended to it:

```
$ cat /foo/bar 2>&1 > /dev/null | cat -n
```

Arithmetic expressions

Integer arithmetic expressions can be used with the **let** command, inside $((..)) expressions, inside array references (e.g. *name*[*expr*]), as numeric arguments to the **test** command, and as the value of an assignment to an integer parameter.

Expressions may contain alpha-numeric parameter identifiers, array references, and integer constants and may be combined with the following C operators (listed and grouped in increasing order of precedence):

Unary operators:

```
+ - !  ++ --
```

Binary operators:

```
,
= *= /= %= += -= <<= >>= &= ^= |=
||
&&
|
```

```
&
== !=
< <= >= >
<< >>
+ -
* / %
```

Ternary operators:

```
?: (precedence is immediately higher than assignment)
```

Grouping operators:

```
( )
```

A parameter that is NULL or unset evaluates to 0. Integer constants may be specified with arbitrary bases using the notation *base*#*number*, where *base* is a decimal integer specifying the base, and *number* is a number in the specified base. Additionally, integers may be prefixed with '0X' or '0x' (specifying base 16) or '0' (base 8) in all forms of arithmetic expressions, except as numeric arguments to the **test** command.

The operators are evaluated as follows:

unary +
: Result is the argument (included for completeness).

unary -
: Negation.

!
: Logical NOT; the result is 1 if argument is zero, 0 if not.

~
: Arithmetic (bit-wise) NOT.

++
: Increment; must be applied to a parameter (not a literal or other expression). The parameter is incremented by 1. When used as a prefix operator, the result is the incremented value of the parameter; when used as a postfix operator, the result is the original value of the parameter.

--
: Similar to **++**, except the parameter is decremented by 1.

,
: Separates two arithmetic expressions; the left-hand side is evaluated first, then the right. The result is the value of the expression on the right-hand side.

=
: Assignment; the variable on the left is set to the value on the right.

*= /= += -= <<= >>= &= ^= |=
: Assignment operators. ⟨*var*⟩⟨*op*⟩=⟨*expr*⟩ is the same as ⟨*var*⟩=⟨*var*⟩⟨*op*⟩⟨*expr*⟩, with any operator precedence in ⟨*expr*⟩ preserved. For example, "var1 *= 5 + 3" is the same as specifying "var1 = var1 * (5 + 3)".

||
: Logical OR; the result is 1 if either argument is non-zero, 0 if not. The right argument is evaluated only if the left argument is zero.

&&
: Logical AND; the result is 1 if both arguments are non-zero, 0 if not. The right argument is evaluated only if the left argument is non-zero.

|
: Arithmetic (bit-wise) OR.

^ Arithmetic (bit-wise) XOR (exclusive-OR).

& Arithmetic (bit-wise) AND.

== Equal; the result is 1 if both arguments are equal, 0 if not.

!= Not equal; the result is 0 if both arguments are equal, 1 if not.

< Less than; the result is 1 if the left argument is less than the right, 0 if not.

<= >= >
Less than or equal, greater than or equal, greater than. See **<**.

<< >> Shift left (right); the result is the left argument with its bits shifted left (right) by the amount given in the right argument.

+ - * /
Addition, subtraction, multiplication, and division.

% Remainder; the result is the remainder of the division of the left argument by the right. The sign of the result is unspecified if either argument is negative.

⟨*arg1*⟩?⟨*arg2*⟩:⟨*arg3*⟩
If ⟨*arg1*⟩ is non-zero, the result is ⟨*arg2*⟩; otherwise the result is ⟨*arg3*⟩.

Co-processes

A co-process, which is a pipeline created with the '|&' operator, is an asynchronous process that the shell can both write to (using **print -p**) and read from (using **read -p**). The input and output of the co-process can also be manipulated using **>&p** and **<&p** redirections, respectively. Once a co-process has been started, another can't be started until the co-process exits, or until the co-process's input has been redirected using an **exec** *n***>&p** redirection. If a co-process's input is redirected in this way, the next co-process to be started will share the output with the first co-process, unless the output of the initial co-process has been redirected using an **exec** *n***<&p** redirection.

Some notes concerning co-processes:

- The only way to close the co-process's input (so the co-process reads an end-of-file) is to redirect the input to a numbered file descriptor and then close that file descriptor e.g. **exec 3>&p; exec 3>&-**.
- In order for co-processes to share a common output, the shell must keep the write portion of the output pipe open. This means that end-of-file will not be detected until all co-processes sharing the co-process's output have exited (when they all exit, the shell closes its copy of the pipe). This can be avoided by redirecting the output to a numbered file descriptor (as this also causes the shell to close its copy). Note that this behaviour is slightly different from the original Korn shell which closes its copy of the write portion of the co-process output when the most recently started co-process (instead of when all sharing co-processes) exits.
- **print -p**will ignore SIGPIPE signals during writes if the signal is not being trapped or ignored; the same is true if the co-process input has been duplicated to another file descriptor and **print -u***n* is used.

Functions

Functions are defined using either Korn shell **function** *function-name* syntax or the Bourne/POSIX shell *function-name*() syntax (see below for the difference between the two forms). Functions are like .-scripts (i.e. scripts sourced using the '.' built-in) in that they are executed in the current environment. However, unlike .-scripts, shell arguments (i.e. positional parameters $1, $2, etc.) are never visible inside them. When the shell is determining the location of a com-

mand, functions are searched after special built-in commands, before regular and non-regular built-ins, and before the PATH is searched.

An existing function may be deleted using **unset -f** *function-name*. A list of functions can be obtained using **typeset +f** and the function definitions can be listed using **typeset -f**. The **autoload** command (which is an alias for **typeset -fu**) may be used to create undefined functions: when an undefined function is executed, the shell searches the path specified in the FPATH parameter for a file with the same name as the function, which, if found, is read and executed. If after executing the file the named function is found to be defined, the function is executed; otherwise, the normal command search is continued (i.e. the shell searches the regular built-in command table and PATH). Note that if a command is not found usingPATH, an attempt is made to autoload a function using FPATH (this is an undocumented feature of the original Korn shell).

Functions can have two attributes, "trace" and "export", which can be set with **typeset -ft** and **typeset -fx**, respectively. When a traced function is executed, the shell's**xtrace** option is turned on for the function's duration; otherwise, the **xtrace** option is turned off. The "export" attribute of functions is currently not used. In the original Korn shell, exported functions are visible to shell scripts that are executed.

Since functions are executed in the current shell environment, parameter assignments made inside functions are visible after the function completes. If this is not the desired effect, the **typeset** command can be used inside a function to create a local parameter. Note that special parameters (e.g. **$$**, **$!**) can't be scoped in this way.

The exit status of a function is that of the last command executed in the function. A function can be made to finish immediately using the **return** command; this may also be used to explicitly specify the exit status.

Functions defined with the **function** reserved word are treated differently in the following ways from functions defined with the **()** notation:

- The $0 parameter is set to the name of the function (Bourne-style functions leave $0 untouched).
- Parameter assignments preceding function calls are not kept in the shell environment (executing Bourne-style functions will keep assignments).
- OPTIND is saved/reset and restored on entry and exit from the function so **getopts** can be used properly both inside and outside the function (Bourne-style functions leave OPTIND untouched, so using **getopts** inside a function interferes with using **getopts** outside the function).

POSIX mode

The shell is intended to be POSIX compliant; however, in some cases, POSIX behaviour is contrary either to the original Korn shell behaviour or to user convenience. How the shell behaves in these cases is determined by the state of the **posix** option (**set -o posix**). If it is on, the POSIX behaviour is followed; otherwise, it is not. The **posix** option is set automatically when the shell starts up if the environment contains the POSIXLY_CORRECT parameter. The shell can also be compiled so that it is in POSIX mode by default; however, this is usually not desirable.

The following is a list of things that are affected by the state of the **posix** option:

- **kill -l**output. In POSIX mode, only signal names are listed (in a single line); in non-POSIX mode, signal numbers, names, and descriptions are printed (in columns).
- **echo** options. In POSIX mode, **-e** and **-E** are not treated as options, but printed like other arguments; in non-POSIX mode, these options control the interpretation of backslash sequences.

- **fg** exit status. In POSIX mode, the exit status is 0 if no errors occur; in non-POSIX mode, the exit status is that of the last foregrounded job.
- **eval** exit status. If **eval** gets to see an empty command (i.e. **eval 'false'**), its exit status in POSIX mode will be 0. In non-POSIX mode, it will be the exit status of the last command substitution that was done in the processing of the arguments to **eval** (or 0 if there were no command substitutions).
- **getopts**. In POSIX mode, options must start with a '-'; in non-POSIX mode, options can start with either '-' or '+'.
- Brace expansion (also known as alternation). In POSIX mode, brace expansion is disabled; in non-POSIX mode, brace expansion is enabled. Note that **set -o posix** (or setting the POSIXLY_CORRECT parameter) automatically turns the **braceexpand** option off; however, it can be explicitly turned on later.
- **set -**. In POSIX mode, this does not clear the**verbose** or **xtrace** options; in non-POSIX mode, it does.
- **set** exit status. In POSIX mode, the exit status of **set** is 0 if there are no errors; in non-POSIX mode, the exit status is that of any command substitutions performed in generating the **set** command. For example, **set -- 'false'; echo $?** prints 0 in POSIX mode, 1 in non-POSIX mode. This construct is used in most shell scripts that use the old getopt(1) command.
- Argument expansion of the **alias**, **export**, **readonly**, and **typeset** commands. In POSIX mode, normal argument expansion is done; in non-POSIX mode, field splitting, file globbing, brace expansion, and (normal) tilde expansion are turned off, while assignment tilde expansion is turned on.
- Signal specification. In POSIX mode, signals can be specified as digits, only if signal numbers match POSIX values (i.e. HUP=1, INT=2, QUIT=3, ABRT=6, KILL=9, ALRM=14, and TERM=15); in non-POSIX mode, signals can always be digits.
- Alias expansion. In POSIX mode, alias expansion is only carried out when reading command words; in non-POSIX mode, alias expansion is carried out on any word following an alias that ended in a space. For example, the following **for** loop uses parameter 'i' in POSIX mode and 'j' in non-POSIX mode:

```
alias a='for ' i='j'
a i in 1 2; do echo i=$i j=$j; done
```

- **test**. In POSIX mode, the expression '**-t**' (preceded by some number of '!' arguments) is always true as it is a non-zero length string; in non-POSIX mode, it tests if file descriptor 1 is a tty(4) (i.e. the *fd* argument to the **-t** test may be left out and defaults to 1).

Strict Bourne shell mode

When the **sh** option is enabled (see the **set** command), **ksh** will behave like sh(1) in the following ways:

- The parameter **$_** is not set to:
  - the expanded alias' full program path after entering commands that are tracked aliases
  - the last argument on the command line after entering external commands
  - the file that changed when MAILPATH is set to monitor a mailbox
- File descriptors are left untouched when executing **exec** with no arguments.

- Backslash-escaped special characters are not substituted in `PS1`.
- Sequences of '((...))' are not interpreted as arithmetic expressions.

Command execution

After evaluation of command-line arguments, redirections, and parameter assignments, the type of command is determined: a special built-in, a function, a regular built-in, or the name of a file to execute found using the `PATH` parameter. The checks are made in the above order. Special built-in commands differ from other commands in that the `PATH` parameter is not used to find them, an error during their execution can cause a non-interactive shell to exit, and parameter assignments that are specified before the command are kept after the command completes. Just to confuse things, if the **posix** option is turned off (see the **set** command below), some special commands are very special in that no field splitting, file globbing, brace expansion, nor tilde expansion is performed on arguments that look like assignments. Regular built-in commands are different only in that the `PATH` parameter is not used to find them.

The original **ksh** and POSIX differ somewhat in which commands are considered special or regular:

POSIX special commands

., **:**, **break**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **times**, **trap**, **unset**

Additional **ksh** special commands

builtin, **typeset**

Very special commands (when POSIX mode is off)

alias, **readonly**, **set**, **typeset**

POSIX regular commands

alias, **bg**, **cd**, **command**, **false**, **fc**, **fg**, **getopts**, **jobs**, **kill**, **pwd**, **read**, **true**, **umask**, **unalias**, **wait**

Additional **ksh** regular commands

[, **echo**, **let**, **print**, **suspend**, **test**, **ulimit**, **whence**

Once the type of command has been determined, any command-line parameter assignments are performed and exported for the duration of the command.

The following describes the special and regular built-in commands:

. *file* [*arg* . . .]
: Execute the commands in *file* in the current environment. The file is searched for in the directories of `PATH`. If arguments are given, the positional parameters may be used to access them while *file* is being executed. If no arguments are given, the positional parameters are those of the environment the command is used in.

: [. . .]
: The null command. Exit status is set to zero.

alias [**-d** | **-t** [**-r**] | **+-x**] [**-p**] [**+**] [*name* [=*value*] . . .]
: Without arguments, **alias** lists all aliases. For any name without a value, the existing alias is listed. Any name with a value defines an alias (see Aliases above).

  When listing aliases, one of two formats is used. Normally, aliases are listed as *name*=*value*, where *value* is quoted. If options were preceded with '**+**', or a lone '**+**' is given on the command line, only *name* is printed.

The **-d** option causes directory aliases, which are used in tilde expansion, to be listed or set (see Tilde expansion above).

If the **-p** option is used, each alias is prefixed with the string "alias ".

The **-t** option indicates that tracked aliases are to be listed/set (values specified on the command line are ignored for tracked aliases). The **-r** option indicates that all tracked aliases are to be reset.

The **-x** option sets (**+x** clears) the export attribute of an alias or, if no names are given, lists the aliases with the export attribute (exporting an alias has no effect).

bg [*job ...*]
Resume the specified stopped job(s) in the background. If no jobs are specified,**%+** is assumed. See Job control below for more information.

bind [**-l**]
The current bindings are listed. If the **-l** flag is given, **bind** instead lists the names of the functions to which keys may be bound. See Emacs editing mode for more information.

bind [**-m**] *string*=[*substitute*] *...*
bind *string*=[*editing-command*] *...*
The specified editing command is bound to the given *string*. Future input of the *string* will cause the editing command to be immediately invoked. If the**-m** flag is given, the specified input *string* will afterwards be immediately replaced by the given *substitute* string, which may contain editing commands. Control characters may be written using caret notation. For example, ^X represents Control-X.

If a certain character occurs as the first character of any bound multi-character *string* sequence, that character becomes a command prefix character. Any character sequence that starts with a command prefix character but that is not bound to a command or substitute is implicitly considered as bound to the 'error' command. By default, two command prefix characters exist: Escape (^[) and Control-X (^X).

The following default bindings show how the arrow keys on an ANSI terminal or xterm are bound (of course some escape sequences won't work out quite this nicely):

```
bind '[[A'=up-history
bind '[[B'=down-history
bind '[[C'=forward-char
bind '[[D'=backward-char
```

break [*level*]
Exit the *level*th inner-most **for**, **select**, **until**, or **while** loop. *level* defaults to 1.

builtin *command* [*arg ...*]
Execute the built-in command *command*.

cd [**-LP**] [*dir*]
Set the working directory to *dir*. If the parameterCDPATH is set, it lists the search path for the directory containing *dir*. A NULL path means the current directory. If *dir* is found in any component of the CDPATH search path other than the NULL path, the name of the new working directory will be written to standard output. If *dir* is missing, the home directory HOME is used. If *dir* is '-', the previous working directory is used (see the OLDPWD parameter).

If the **-L** option (logical path) is used or if the **physical** option isn't set (see the **set** command below), references to '..' in *dir* are relative to the path used to get to the direc-

tory. If the **-P** option (physical path) is used or if the **physical** option is set, '..' is relative to the filesystem directory tree. The PWD and OLDPWD parameters are updated to reflect the current and old working directory, respectively.

cd [**-LP**] *old new*
The string *new* is substituted for *old* in the current directory, and the shell attempts to change to the new directory.

command [**-pVv**] *cmd* [*arg ...*]
If neither the **-v** nor **-V** option is given, *cmd* is executed exactly as if **command** had not been specified, with two exceptions: firstly, *cmd* cannot be an alias or a shell function; and secondly, special built-in commands lose their specialness (i.e. redirection and utility errors do not cause the shell to exit, and command assignments are not permanent).

If the **-p** option is given, a default search path is used instead of the current value of PATH (the actual value of the default path is system dependent: on POSIX-ish systems, it is the value returned by **getconf PATH**). Nevertheless, reserved words, aliases, shell functions, and builtin commands are still found before external commands.

If the **-v** option is given, instead of executing *cmd*, information about what would be executed is given (and the same is done for *arg ...*). For special and regular built-in commands and functions, their names are simply printed; for aliases, a command that defines them is printed; and for commands found by searching the PATH parameter, the full path of the command is printed. If no command is found (i.e. the path search fails), nothing is printed and **command** exits with a non-zero status. The **-V** option is like the **-v** option, except it is more verbose.

continue [*level*]
Jumps to the beginning of the *level*th inner-most **for**, **select**, **until**, or **while** loop. *level* defaults to 1.

echo [**-Een**] [*arg ...*]
Prints its arguments (separated by spaces) followed by a newline, to the standard output. The newline is suppressed if any of the arguments contain the backslash sequence '\c'. See the **print** command below for a list of other backslash sequences that are recognized.

The options are provided for compatibility with BSD shell scripts. The **-n** option suppresses the trailing newline, **-e** enables backslash interpretation (a no-op, since this is normally done), and **-E** suppresses backslash interpretation. If the **posix** option is set, only the first argument is treated as an option, and only if it is exactly "-n".

eval *command ...*
The arguments are concatenated (with spaces between them) to form a single string which the shell then parses and executes in the current environment.

exec [*command* [*arg ...*]]
The command is executed without forking, replacing the shell process.

If no command is given except for I/O redirection, the I/O redirection is permanent and the shell is not replaced. Any file descriptors greater than 2 which are opened or 'd -- dup(2)Ns in this way are not made available to other executed commands (i.e. commands that are not built-in to the shell). Note that the Bourne shell differs here; it does pass these file descriptors on.

exit [*status*]
The shell exits with the specified exit status. If *status* is not specified, the exit status is the current value of the **$?** parameter.

export [**-p**] [*parameter*[=*value*]]
Sets the export attribute of the named parameters. Exported parameters are passed in the environment to executed commands. If values are specified, the named parameters are also assigned.

If no parameters are specified, the names of all parameters with the export attribute are printed one per line, unless the **-p** option is used, in which case **export** commands defining all exported parameters, including their values, are printed.

false A command that exits with a non-zero status.

fc [**-e** *editor* | **-l** [**-n**]] [**-r**] [*first* [*last*]]
Fix command. *first* and *last* select commands from the history. Commands can be selected by history number or a string specifying the most recent command starting with that string. The **-l** option lists the command on standard output, and **-n** inhibits the default command numbers. The **-r** option reverses the order of the list. Without **-l**, the selected commands are edited by the editor specified with the **-e** option, or if no **-e** is specified, the editor specified by the FCEDIT parameter (if this parameter is not set, /bin/ed is used), and then executed by the shell.

fc -s[**-g**] [*old*=*new*] [*prefix*]
Re-execute the most recent command beginning with *prefix*, or the previous command if no *prefix* is specified, performing the optional substitution of *old* with *new*. If **-g** is specified, all occurrences of *old* are replaced with *new*. The editor is not invoked when the **-s** flag is used. The obsolescent equivalent "**-e** -" is also accepted. This command is usually accessed with the predefined **alias r='fc -s'**.

fg [*job* ...]
Resume the specified job(s) in the foreground. If no jobs are specified, **%+** is assumed. See Job control below for more information.

getopts *optstring name* [*arg* ...]
Used by shell procedures to parse the specified arguments (or positional parameters, if no arguments are given) and to check for legal options. *optstring* contains the option letters that **getopts** is to recognize. If a letter is followed by a colon, the option is expected to have an argument. Options that do not take arguments may be grouped in a single argument. If an option takes an argument and the option character is not the last character of the argument it is found in, the remainder of the argument is taken to be the option's argument; otherwise, the next argument is the option's argument.

Each time **getopts** is invoked, it places the next option in the shell parameter *name* and the index of the argument to be processed by the next call to **getopts** in the shell parameter OPTIND. If the option was introduced with a '+', the option placed in *name* is prefixed with a '+'. When an option requires an argument, **getopts** places it in the shell parameter OPTARG.

When an illegal option or a missing option argument is encountered, a question mark or a colon is placed in *name* (indicating an illegal option or missing argument, respectively) and OPTARG is set to the option character that caused the problem. Furthermore, if *optstring* does not begin with a colon, a question mark is placed in *name*, OPTARG is unset, and an error message is printed to standard error.

When the end of the options is encountered, **getopts** exits with a non-zero exit status. Options end at the first (non-option argument) argument that does not start with a '-', or when a '--' argument is encountered.

Option parsing can be reset by setting `OPTIND` to 1 (this is done automatically whenever the shell or a shell procedure is invoked).

Warning: Changing the value of the shell parameter `OPTIND` to a value other than 1, or parsing different sets of arguments without resetting `OPTIND`, may lead to unexpected results.

hash [**-r**] [*name ...*]
Without arguments, any hashed executable command pathnames are listed. The **-r** option causes all hashed commands to be removed from the hash table. Each *name* is searched as if it were a command name and added to the hash table if it is an executable command.

jobs [**-lnp**] [*job ...*]
Display information about the specified job(s); if no jobs are specified, all jobs are displayed. The **-n** option causes information to be displayed only for jobs that have changed state since the last notification. If the **-l** option is used, the process ID of each process in a job is also listed. The **-p** option causes only the process group of each job to be printed. See Job control below for the format of *job* and the displayed job.

kill [**-s** *signame* | *-signum* | *-signame*] { *job* | *pid* | *pgrp* } ...
Send the specified signal to the specified jobs, process IDs, or process groups. If no signal is specified, the `TERM` signal is sent. If a job is specified, the signal is sent to the job's process group. See Job control below for the format of *job*.

kill -l[*exit-status ...*]
Print the signal name corresponding to *exit-status*. If no arguments are specified, a list of all the signals, their numbers, and a short description of them are printed.

let [*expression ...*]
Each expression is evaluated (see Arithmetic expressions above). If all expressions are successfully evaluated, the exit status is 0 (1) if the last expression evaluated to non-zero (zero). If an error occurs during the parsing or evaluation of an expression, the exit status is greater than 1. Since expressions may need to be quoted, ((*expr*)) is syntactic sugar for let *expr*.

print [**-nprsu**[*n*] | **-R** [**-en**]] [*argument ...*]
print prints its arguments on the standard output, separated by spaces and terminated with a newline. The **-n** option suppresses the newline. By default, certain C escapes are translated. These include '`\b`', '`\f`', '`\n`', '`\r`', '`\t`', '`\v`', and `\0###` ('`#`' is an octal digit, of which there may be 0 to 3). '`\c`' is equivalent to using the **-n** option. '\' expansion may be inhibited with the **-r** option. The **-s** option prints to the history file instead of standard output; the **-u** option prints to file descriptor *n* (*n* defaults to 1 if omitted); and the **-p** option prints to the co-process (see Co-processes above).

The **-R** option is used to emulate, to some degree, the BSD `echo`(1) command, which does not process '\' sequences unless the **-e** option is given. As above, the **-n** option suppresses the trailing newline.

pwd [**-LP**]
Print the present working directory. If the **-L** option is used or if the **physical** option isn't set (see the **set** command below), the logical path is printed (i.e. the path used to **cd** to the current directory). If the **-P** option (physical path) is used or if the **physical** option is set, the path determined from the filesystem (by following '..' directories to the root directory) is printed.

read [**-prsu**[*n*]] [*parameter ...*]
Reads a line of input from the standard input, separates the line into fields using the IFS parameter (see Substitution above), and assigns each field to the specified parameters. If there are more parameters than fields, the extra parameters are set to NULL, or alternatively, if there are more fields than parameters, the last parameter is assigned the remaining fields (inclusive of any separating spaces). If no parameters are specified, the REPLY parameter is used. If the input line ends in a backslash and the **-r** option was not used, the backslash and the newline are stripped and more input is read. If no input is read, **read** exits with a non-zero status.

The first parameter may have a question mark and a string appended to it, in which case the string is used as a prompt (printed to standard error before any input is read) if the input is a tty(4) (e.g. **read nfoo?'number of foos: '**).

The **-u***n* and **-p** options cause input to be read from file descriptor *n* (*n* defaults to 0 if omitted) or the current co-process (see Co-processes above for comments on this), respectively. If the **-s** option is used, input is saved to the history file.

readonly [**-p**] [*parameter* [=*value*] ...]
Sets the read-only attribute of the named parameters. If values are given, parameters are set to them before setting the attribute. Once a parameter is made read-only, it cannot be unset and its value cannot be changed.

If no parameters are specified, the names of all parameters with the read-only attribute are printed one per line, unless the **-p** option is used, in which case **readonly** commands defining all read-only parameters, including their values, are printed.

return [*status*]
Returns from a function or **.** script, with exit status *status*. If no *status* is given, the exit status of the last executed command is used. If used outside of a function or **.** script, it has the same effect as **exit**. Note that **ksh** treats both profile and ENV files as **.** scripts, while the original Korn shell only treats profiles as **.** scripts.

set [**+-abCefhkmnpsuvXx**] [**+-o** *option*] [**+-A** *name*] [**--**] [*arg ...*]
The **set** command can be used to set (**-**) or clear (**+**) shell options, set the positional parameters, or set an array parameter. Options can be changed using the **+-o** *option* syntax, where *option* is the long name of an option, or using the **+-***letter* syntax, where *letter* is the option's single letter name (not all options have a single letter name). The following table lists both option letters (if they exist) and long names along with a description of what the option does:

-A *name* Sets the elements of the array parameter *name* to *arg ...* If **-A** is used, the array is reset (i.e. emptied) first; if **+A** is used, the first N elements are set (where N is the number of arguments); the rest are left untouched.

-a | **allexport**
All new parameters are created with the export attribute.

-b | **notify** Print job notification messages asynchronously, instead of just before the prompt. Only used if job control is enabled (**-m**).

-C | **noclobber**
Prevent **>** redirection from overwriting existing files. Instead, **>|** must be used to force an overwrite.

-e | errexit Exit (after executing the `ERR` trap) as soon as an error occurs or a command fails (i.e. exits with a non-zero status). This does not apply to commands whose exit status is explicitly tested by a shell construct such as **if**, **until**, **while**, or **!** statements. For **&&** or **||**, only the status of the last command is tested.

-f | noglob Do not expand file name patterns.

-h | trackall
Create tracked aliases for all executed commands (see Aliases above). Enabled by default for non-interactive shells.

-k | keyword Parameter assignments are recognized anywhere in a command.

-m | monitor Enable job control (default for interactive shells).

-n | noexec Do not execute any commands. Useful for checking the syntax of scripts (ignored if interactive).

-p | privileged
The shell is a privileged shell. It is set automatically if, when the shell starts, the real UID or GID does not match the effective UID (EUID) or GID (EGID), respectively. See above for a description of what this means.

-s | stdin If used when the shell is invoked, commands are read from standard input. Set automatically if the shell is invoked with no arguments.

When **-s** is used with the **set** command it causes the specified arguments to be sorted before assigning them to the positional parameters (or to array *name*, if **-A** is used).

-u | nounset Referencing of an unset parameter is treated as an error, unless one of the '-', '+', or '=' modifiers is used.

-v | verbose Write shell input to standard error as it is read.

-X | markdirs
Mark directories with a trailing '/' during file name generation.

-x | xtrace Print commands and parameter assignments when they are executed, preceded by the value of `PS4`.

bgnice Background jobs are run with lower priority.

braceexpand Enable brace expansion (a.k.a. alternation).

csh-history Enables a subset of csh(1)Ns -style history editing using the '!' character.

emacs Enable BRL emacs-like command-line editing (interactive shells only); see Emacs editing mode.

emacs-usemeta
In emacs command-line editing, use the 8th bit as meta (^[) prefix. This is the default.

gmacs Enable gmacs-like command-line editing (interactive shells only). Currently identical to emacs editing except that transpose (^T) acts slightly differently.

ignoreeof The shell will not (easily) exit when end-of-file is read; **exit** must be used. To avoid infinite loops, the shell will exit if EOF is read 13 times in a row.

interactive The shell is an interactive shell. This option can only be used when the shell is invoked. See above for a description of what this means.

login The shell is a login shell. This option can only be used when the shell is invoked. See above for a description of what this means.

nobeep Do not beep on bell.

nohup Do not kill running jobs with a SIGHUP signal when a login shell exits. Currently set by default; this is different from the original Korn shell (which doesn't have this option, but does send the SIGHUP signal).

nolog No effect. In the original Korn shell, this prevents function definitions from being stored in the history file.

physical Causes the **cd** and **pwd** commands to use "physical" (i.e. the filesystem's) '..' directories instead of "logical" directories (i.e. the shell handles '..', which allows the user to be oblivious of symbolic links to directories). Clear by default. Note that setting this option does not affect the current value of the PWD parameter; only the **cd** command changes PWD. See the **cd** and **pwd** commands above for more details.

posix Enable POSIX mode. See POSIX mode above.

restricted The shell is a restricted shell. This option can only be used when the shell is invoked. See above for a description of what this means.

sh Enable strict Bourne shell mode (see Strict Bourne shell mode above).

vi Enable -like -- vi(1)Ns command-line editing (interactive shells only).

vi-esccomplete
In vi command-line editing, do command and file name completion when escape (^[) is entered in command mode.

vi-show8 Prefix characters with the eighth bit set with 'M-'. If this option is not set, characters in the range 128-160 are printed as is, which may cause problems.

vi-tabcomplete
In vi command-line editing, do command and file name completion when tab (^I) is entered in insert mode. This is the default.

viraw No effect. In the original Korn shell, unless **viraw** was set, the vi command-line mode would let the tty(4) driver do the work until ESC (^[) was entered. **ksh** is always in viraw mode.

These options can also be used upon invocation of the shell. The current set of options (with single letter names) can be found in the parameter '$-'. **set -o** with no option name will list all the options and whether each is on or off; **set +o** will print the current shell options in a form that can be reinput to the shell to achieve the same option settings.

Remaining arguments, if any, are positional parameters and are assigned, in order, to the positional parameters (i.e. $1, $2, etc.). If options end with '--' and there are no remaining arguments, all positional parameters are cleared. If no options or arguments are given, the values of all names are printed. For unknown historical reasons, a lone '-' option is treated

specially - it clears both the **-x** and **-v** options.

shift [*number*]
The positional parameters *number*+1, *number*+2, etc. are renamed to '1', '2', etc. *number* defaults to 1.

suspend
Stops the shell as if it had received the suspend character from the terminal. It is not possible to suspend a login shell unless the parent process is a member of the same terminal session but is a member of a different process group. As a general rule, if the shell was started by another shell or via su(1), it can be suspended.

test *expression*
[*expression* **]**
test evaluates the *expression* and returns zero status if true, 1 if false, or greater than 1 if there was an error. It is normally used as the condition command of **if** and **while** statements. Symbolic links are followed for all *file* expressions except **-h** and **-L**.

The following basic expressions are available:

-a *file*	*file* exists.
-b *file*	*file* is a block special device.
-c *file*	*file* is a character special device.
-d *file*	*file* is a directory.
-e *file*	*file* exists.
-f *file*	*file* is a regular file.
-G *file*	*file*'s group is the shell's effective group ID.
-g *file*	*file*'s mode has the setgid bit set.
-h *file*	*file* is a symbolic link.
-k *file*	*file*'s mode has the `sticky`(8) bit set.
-L *file*	*file* is a symbolic link.
-O *file*	*file*'s owner is the shell's effective user ID.
-o *option*	Shell *option* is set (see the **set** command above for a list of options). As a non-standard extension, if the option starts with a '**!**', the test is negated; the test always fails if *option* doesn't exist (so [-o foo -o -o !foo] returns true if and only if option *foo* exists).
-p *file*	*file* is a named pipe.
-r *file*	*file* exists and is readable.
-S *file*	*file* is a unix(4)Ns -domain socket.
-s *file*	*file* is not empty.
-t [*fd*]	File descriptor *fd* is a `tty`(4) device. If the **posix** option is not set, *fd* may be left out, in which case it is taken to be 1 (the behaviour differs due to the special POSIX rules described above).

-u *file* *file*'s mode has the setuid bit set.

-w *file* *file* exists and is writable.

-x *file* *file* exists and is executable.

file1 **-nt** *file2* *file1* is newer than *file2* or *file1* exists and *file2* does not.

file1 **-ot** *file2* *file1* is older than *file2* or *file2* exists and *file1* does not.

file1 **-ef** *file2* *file1* is the same file as *file2*.

string *string* has non-zero length.

-n *string* *string* is not empty.

-z *string* *string* is empty.

string = *string* Strings are equal.

string == *string*
Strings are equal.

string != *string* Strings are not equal.

number **-eq** *number*
Numbers compare equal.

number **-ne** *number*
Numbers compare not equal.

number **-ge** *number*
Numbers compare greater than or equal.

number **-gt** *number*
Numbers compare greater than.

number **-le** *number*
Numbers compare less than or equal.

number **-lt** *number*
Numbers compare less than.

The above basic expressions, in which unary operators have precedence over binary operators, may be combined with the following operators (listed in increasing order of precedence):

```
expr -o expr Logical OR.
expr -a expr Logical AND.
! expr Logical NOT.
( expr ) Grouping.
```

On operating systems not supporting `/dev/fd/`*n* devices (where *n* is a file descriptor number), the **test** command will attempt to fake it for all tests that operate on files (except the **-e** test). For example, [-w /dev/fd/2] tests if file descriptor 2 is writable.

Note that some special rules are applied (courtesy of POSIX) if the number of arguments to **test** or **[...]** is less than five: if leading '`!`' arguments can be stripped such that only one argument remains then a string length test is performed (again, even if the argument is a unary operator); if leading '`!`' arguments can be stripped such that three arguments remain and the second argument is a binary operator, then the binary operation is performed (even if the first argument is a unary operator, including an unstripped '`!`').

Note: A common mistake is to use "if [$foo = bar]" which fails if parameter "foo" is `NULL` or unset, if it has embedded spaces (i.e. `IFS` characters), or if it is a unary operator like '!' or '**-n**'. Use tests like "if [X$foo = Xbar]" instead.

time [**-p**] [*pipeline*]
If a *pipeline* is given, the times used to execute the pipeline are reported. If no pipeline is given, then the user and system time used by the shell itself, and all the commands it has run since it was started, are reported. The times reported are the real time (elapsed time from start to finish), the user CPU time (time spent running in user mode), and the system CPU time (time spent running in kernel mode). Times are reported to standard error; the format of the output is:

```
0m0.00s real 0m0.00s user 0m0.00s system
```

If the **-p** option is given the output is slightly longer:

```
real 0.00
user 0.00
sys 0.00
```

It is an error to specify the **-p** option unless *pipeline* is a simple command.

Simple redirections of standard error do not affect the output of the **time** command:

```
$ time sleep 1 2> afile
$ { time sleep 1; } 2> afile
```

Times for the first command do not go to "afile", but those of the second command do.

times Print the accumulated user and system times used both by the shell and by processes that the shell started which have exited. The format of the output is:

```
0m0.00s 0m0.00s
0m0.00s 0m0.00s
```

trap [*handler signal ...*]
Sets a trap handler that is to be executed when any of the specified signals are received. *handler* is either a `NULL` string, indicating the signals are to be ignored, a minus sign ('-'), indicating that the default action is to be taken for the signals (see signal(3)) , or a string containing shell commands to be evaluated and executed at the first opportunity (i.e. when the current command completes, or before printing the next `PS1` prompt) after receipt of one of the signals. *signal* is the name of a signal (e.g. `PIPE` or `ALRM`) or the number of the signal (see the **kill -l** command above).

There are two special signals: `EXIT` (also known as 0), which is executed when the shell is about to exit, and `ERR`, which is executed after an error occurs (an error is something that would cause the shell to exit if the **-e** or **errexit** option were set - see the **set** command above). `EXIT` handlers are executed in the environment of the last executed command. Note that for non-interactive shells, the trap handler cannot be changed for signals that were ignored when the shell started.

With no arguments, **trap** lists, as a series of **trap** commands, the current state of the traps that have been set since the shell started. Note that the output of **trap** cannot be usefully piped to another process (an artifact of the fact that traps are cleared when subprocesses are created).

The original Korn shell's `DEBUG` trap and the handling of `ERR` and `EXIT` traps in functions are not yet implemented.

true A command that exits with a zero value.

typeset [[**+-lprtUux**] [**-L**[*n*]] [**-R**[*n*]] [**-Z**[*n*]] [**-i**[*n*]] | **-f** [**-tux**]] [*name* [=*value*] . . .]
Display or set parameter attributes. With no *name* arguments, parameter attributes are displayed; if no options are used, the current attributes of all parameters are printed as **typeset** commands; if an option is given (or '**-**' with no option letter), all parameters and their values with the specified attributes are printed; if options are introduced with '**+**', parameter values are not printed.

If *name* arguments are given, the attributes of the named parameters are set (**-**) or cleared (**+**). Values for parameters may optionally be specified. If **typeset** is used inside a function, any newly created parameters are local to the function.

When **-f** is used, **typeset** operates on the attributes of functions. As with parameters, if no *name* arguments are given, functions are listed with their values (i.e. definitions) unless options are introduced with '**+**', in which case only the function names are reported.

-f Function mode. Display or set functions and their attributes, instead of parameters.

-i[*n*] Integer attribute. *n* specifies the base to use when displa ying the integer (if not specified, the base given in the first assignment is used). Parameters with this attribute may be assigned values containing arithmetic expressions.

-L[*n*] Left justify attribute. *n* specifies the field width. If *n* is not specified, the curren t width of a parameter (or the width of its first assigned value) is used. Leading whitespace (and zeros, if used with the **-Z** option) is stripped. If necessary, values are either truncated or space padded to fit the field width.

-l Lower case attribute. All upper case characters in values are converted to lower case. (In the original Korn shell, this parameter meant "long integer" when used with the **-i** option.)

-p Print complete **typeset** commands that can be used to re-create the attributes (but not the values) of parameters. This is the default action (option exists for ksh93 compatibility).

-R[*n*] Right justify attribute. *n* specifies the field width. If *n* is not specified, the curren t width of a parameter (or the width of its first assigned value) is used. Trailing whitespace is stripped. If necessary, values are either stripped of leading characters or space padded to make them fit the field width.

-r Read-only attribute. Parameters with this attribute may not be assigned to or unset. Once this attribute is set, it cannot be turned off.

-t Tag attribute. Has no meaning to the shell; provided for application use.

For functions, **-t** is the trace attribute. When functions with the trace attribute are executed, the **xtrace** (**-x**) shell option is temporarily turned on.

-U Unsigned integer attribute. Integers are printed as unsigned values (only useful when combined with the **-i** option). This option is not in the original Korn shell.

-u Upper case attribute. All lower case characters in values are converted to upper case. (In the original Korn shell, this parameter meant "unsigned integer" when used with the **-i** option, whic h meant upper case letters would never be used for bases greater than 10. See the **-U** option.)

For functions, **-u** is the undefined attribute. See Functions above for the implications of this.

-x Export attribute. Parameters (or functions) are placed in the environment of any executed commands. Exported functions are not yet implemented.

-Z[*n*] Zero fill attribute. If not combined with **-L**, this is the same as **-R**, except zero padding is used instead of space padding.

ulimit [**-acdfHlmnpSst** [*value*]] ...
Display or set process limits. If no options are used, the file size limit (**-f**) is assumed. *value*, if specified, may be either an arithmetic expression starting with a number or the word "unlimited". The limits affect the shell and any processes created by the shell after a limit is imposed; limits may not be increased once they are set.

-a Display all limits; unless **-H** is used, soft limits are displayed.

-c *n* Impose a size limit of *n* blocks on the size of core dumps.

-d *n* Impose a size limit of *n* kilobytes on the size of the data area.

-f *n* Impose a size limit of *n* blocks on files written by the shell and its child processes (files of any size may be read).

-H Set the hard limit only (the default is to set both hard and soft limits).

-l *n* Impose a limit of *n* kilobytes on the amount of locked (wired) physical memory.

-m *n* Impose a limit of *n* kilobytes on the amount of physical memory used. This limit is not enforced.

-n *n* Impose a limit of *n* file descriptors that can be open at once.

-p *n* Impose a limit of *n* processes that can be run by the user at any one time.

-S Set the soft limit only (the default is to set both hard and soft limits).

-s *n* Impose a size limit of *n* kilobytes on the size of the stack area.

-t *n* Impose a time limit of *n* CPU seconds spent in user mode to be used by each process.

As far as **ulimit** is concerned, a block is 512 bytes.

umask [**-S**] [*mask*]
Display or set the file permission creation mask, or umask (see . -- umask(2)) If the **-S** option is used, the mask displayed or set is symbolic; otherwise, it is an octal number.

Symbolic masks are like those used by chmod(1). When used, they describe what permissions may be made available (as opposed to octal masks in which a set bit means the corresponding bit is to be cleared). For example, "ug=rwx,o=" sets the mask so files will not be readable, writable, or executable by "others", and is equivalent (on most systems) to the octal mask "007".

unalias [**-adt**] [*name* ...]
The aliases for the given names are removed. If the **-a** option is used, all aliases are removed. If the **-t** or **-d** options are used, the indicated operations are carried out on tracked or directory aliases, respectively.

unset [**-fv**] *parameter* ...
Unset the named parameters (**-v**, the default) or functions (**-f**). The exit status is non-zero if any of the parameters have the read-only attribute set, zero otherwise.

wait [*job ...*]
Wait for the specified job(s) to finish. The exit status of **wait** is that of the last specified job; if the last job is killed by a signal, the exit status is 128 + the number of the signal (see **kill -l** *exit-status* above); if the last specified job can't be found (because it never existed, or had already finished), the exit status of **wait** is 127. See Job control below for the format of *job*. **wait** will return if a signal for which a trap has been set is received, or if a SIGHUP, SIGINT, or SIGQUIT signal is received.

If no jobs are specified, **wait** waits for all currently running jobs (if any) to finish and exits with a zero status. If job monitoring is enabled, the completion status of jobs is printed (this is not the case when jobs are explicitly specified).

whence [**-pv**] [*name ...*]
For each *name*, the type of command is listed (reserved word, built-in, alias, function, tracked alias, or executable). If the **-p** option is used, a path search is performed even if *name* is a reserved word, alias, etc. Without the **-v** option, **whence** is similar to **command -v** except that **whence** won't print aliases as alias commands. With the **-v** option, **whence** is the same as **command -V**. Note that for**whence**, the **-p** option does not affect the search path used, as it does for **command**. If the type of one or more of the names could not be determined, the exit status is non-zero.

Job control
Job control refers to the shell's ability to monitor and control jobs, which are processes or groups of processes created for commands or pipelines. At a minimum, the shell keeps track of the status of the background (i.e. asynchronous) jobs that currently exist; this information can be displayed using the **jobs** commands. If job control is fully enabled (using **set -m** or **set -o monitor**), as it is for interactive shells, the processes of a job are placed in their own process group. Foreground jobs can be stopped by typing the suspend character from the terminal (normally ^Z), jobs can be restarted in either the foreground or background using the **fg** and **bg** commands, and the state of the terminal is saved or restored when a foreground job is stopped or restarted, respectively.

Note that only commands that create processes (e.g. asynchronous commands, subshell commands, and non-built-in, non-function commands) can be stopped; commands like **read** cannot be.

When a job is created, it is assigned a job number. For interactive shells, this number is printed inside "[..]", followed by the process IDs of the processes in the job when an asynchronous command is run. A job may be referred to in the **bg**, **fg**, **jobs**, **kill**, and **wait** commands either by the process ID of the last process in the command pipeline (as stored in the **$!** parameter) or by prefixing the job number with a percent sign ('%'). Other percent sequences can also be used to refer to jobs:

%+ | %% | %	The most recently stopped job or, if there are no stopped jobs, the oldest running job.
%-	The job that would be the **%+** job if the latter did not exist.
%*n*	The job with job number *n*.
%?*string*	The job with its command containing the string *string* (an error occurs if multiple jobs are matched).
%*string*	The job with its command starting with the string *string* (an error occurs if multiple jobs are matched).

When a job changes state (e.g. a background job finishes or foreground job is stopped), the shell prints the following status information:

[*number*] *flag status command*

where...

number is the job number of the job;

flag is the '+' or '-' character if the job is the **%+** or **%-** job, respectively, or space if it is neither;

status indicates the current state of the job and can be:

Done [*number*]
The job exited. *number* is the exit status of the job, whic h is omitted if the status is zero.

Running The job has neither stopped nor exited (note that running does not necessarily mean consuming CPU time - the process could be blocked waiting for some event).

Stopped [*signal*]
The job was stopped by the indicated *signal* (if no signal is given, the job was stopped by SIGTSTP).

signal-description ["core dumped"]
The job was killed by a signal (e.g. memory fault, hangup); use **kill -l** for a list of signal descriptions. The "core dumped" message indicates the process created a core file.

command is the command that created the process. If there are multiple processes in the job, each process will have a line showing its *command* and possibly its *status*, if it is different from the status of the previous process.

When an attempt is made to exit the shell while there are jobs in the stopped state, the shell warns the user that there are stopped jobs and does not exit. If another attempt is immediately made to exit the shell, the stopped jobs are sent a SIGHUP signal and the shell exits. Similarly, if the **nohup** option is not set and there are running jobs when an attempt is made to exit a login shell, the shell warns the user and does not exit. If another attempt is immediately made to exit the shell, the running jobs are sent a SIGHUP signal and the shell exits.

Interactive input line editing

The shell supports three modes of reading command lines from a tty(4) in an interactive session, controlled by the **emacs**, **gmacs**, and **vi** options (at most one of these can be set at once). The default is **emacs**. Editing modes can be set explicitly using the**set** built-in, or implicitly via the EDITOR and VISUAL environment variables. If none of these options are enabled, the shell simply reads lines using the normal tty(4) driver. If the**emacs** or **gmacs** option is set, the shell allo ws emacs-like editing of the command; similarly, if the **vi** option is set, the shell allows vi-like editing of the command. These modes are described in detail in the following sections.

In these editing modes, if a line is longer than the screen width (see the COLUMNS parameter), a '>', '+', or '<' character is displayed in the last column indicating that there are more characters after, before and after, or before the current position, respectively. The line is scrolled horizontally as necessary.

Emacs editing mode

When the **emacs** option is set, interactive input line editing is enabled. Warning: This mode is slightly different from the emacs mode in the original Korn shell. In this mode, various editing commands (typically bound to one or more control characters) cause immediate actions without waiting

for a newline. Several editing commands are bound to particular control characters when the shell is invoked; these bindings can be changed using the **bind** command.

The following is a list of available editing commands. Each description starts with the name of the command, suffixed with a colon; an [*n*] (if the command can be prefixed with a count); and any keys the command is bound to by default, written using caret notation e.g. the ASCII ESC character is written as ^[. ^[A-Z] sequences are not case sensitive. A count prefix for a command is entered using the sequence ^[*n*, where *n* is a sequence of 1 or more digits. Unless otherwise specified, if a count is omitted, it defaults to 1.

Note that editing command names are used only with the **bind** command. Furthermore, many editing commands are useful only on terminals with a visible cursor. The default bindings were chosen to resemble corresponding Emacs key bindings. The user's `tty`(4) characters (e.g. `ERASE`) are bound to reasonable substitutes and override the default bindings.

abort: ^C, ^G
: Useful as a response to a request for a **search-history** pattern in order to abort the search.

auto-insert: [*n*]
: Simply causes the character to appear as literal input. Most ordinary characters are bound to this.

backward-char: [*n*] ^B, ^XD
: Moves the cursor backward *n* characters.

backward-word: [*n*] ^[b
: Moves the cursor backward to the beginning of the word; words consist of alphanumerics, underscore ('_'), and dollar sign ('$') characters.

beginning-of-history: ^[<
: Moves to the beginning of the history.

beginning-of-line: ^A
: Moves the cursor to the beginning of the edited input line.

capitalize-word: [*n*] ^[C, ^[c
: Uppercase the first character in the next *n* words, leaving the cursor past the end of the last word.

comment: ^[#
: If the current line does not begin with a comment character, one is added at the beginning of the line and the line is entered (as if return had been pressed); otherwise, the existing comment characters are removed and the cursor is placed at the beginning of the line.

complete: ^[^[
: Automatically completes as much as is unique of the command name or the file name containing the cursor. If the entire remaining command or file name is unique, a space is printed after its completion, unless it is a directory name in which case '/' is appended. If there is no command or file name with the current partial word as its prefix, a bell character is output (usually causing a beep to be sounded).

  Custom completions may be configured by creating an array named `complete_command`, optionally suffixed with an argument number to complete only for a single argument. So defining an array named `complete_kill` provides possible completions for any argument to the `kill`(1) command, but `complete_kill_1` only completes the first argument. For example, the following command makes **ksh** offer a selection of signal names for the first argument to kill(1):

```
set -A complete_kill_1 -- -9 -HUP -INFO -KILL -TERM
```

complete-command: ^X^[
Automatically completes as much as is unique of the command name having the partial word up to the cursor as its prefix, as in the **complete** command above.

complete-file: ^[^X
Automatically completes as much as is unique of the file name having the partial word up to the cursor as its prefix, as in the **complete** command described above.

complete-list: ^I, ^[=
Complete as much as is possible of the current word, and list the possible completions for it. If only one completion is possible, match as in the **complete** command above.

delete-char-backward: [*n*] ERASE, ^?, ^H
Deletes *n* characters before the cursor.

delete-char-forward: [*n*] Delete
Deletes *n* characters after the cursor.

delete-word-backward: [*n*] ERASE, ^[^?, ^[^H, ^[h
Deletes *n* words before the cursor.

delete-word-forward: [*n*] ^[d
Deletes characters after the cursor up to the end of *n* words.

down-history: [*n*] ^N, ^XB
Scrolls the history buffer forward *n* lines (later). Each input line originally starts just after the last entry in the history buffer, so **down-history** is not useful until either **search-history** or **up-history** has been performed.

downcase-word: [*n*] ^[L, ^[l
Lowercases the next *n* words.

end-of-history: ^[>
Moves to the end of the history.

end-of-line: ^E
Moves the cursor to the end of the input line.

eot: ^_ Acts as an end-of-file; this is useful because edit-mode input disables normal terminal input canonicalization.

eot-or-delete: [*n*] ^D
Acts as **eot** if alone on a line; otherwise acts as **delete-char-forward**.

error: Error (ring the bell).

exchange-point-and-mark: ^X^X
Places the cursor where the mark is and sets the mark to where the cursor was.

expand-file: ^[*
Appends a '*' to the current word and replaces the word with the result of performing file globbing on the word. If no files match the pattern, the bell is rung.

forward-char: [*n*] ^F, ^XC
Moves the cursor forward *n* characters.

forward-word: [*n*] ^[f
Moves the cursor forward to the end of the *n*th word.

goto-history: [*n*] ^[g
Goes to history number *n*.

kill-line: KILL
Deletes the entire input line.

kill-region: ^W
Deletes the input between the cursor and the mark.

kill-to-eol: [*n*] ^K
Deletes the input from the cursor to the end of the line if *n* is not specified; otherwise deletes characters between the cursor and column *n*.

list: ^[?
Prints a sorted, columnated list of command names or file names (if any) that can complete the partial word containing the cursor. Directory names have '/' appended to them.

list-command: ^X?
Prints a sorted, columnated list of command names (if any) that can complete the partial word containing the cursor.

list-file: ^X^Y
Prints a sorted, columnated list of file names (if any) that can complete the partial word containing the cursor. File type indicators are appended as described under **list** above.

newline: ^J, ^M
Causes the current input line to be processed by the shell. The current cursor position may be anywhere on the line.

newline-and-next: ^O
Causes the current input line to be processed by the shell, and the next line from history becomes the current line. This is only useful after an **up-history** or **search-history**.

no-op: QUIT
This does nothing.

prev-hist-word: [*n*] ^[., ^[_
The last (*n*th) word of the previous command is inserted at the cursor.

quote: ^^
The following character is taken literally rather than as an editing command.

redraw: ^L
Reprints the prompt string and the current input line.

search-character-backward: [*n*] ^[^]
Search backward in the current line for the *n*th occurrence of the next character typed.

search-character-forward: [*n*] ^]
Search forward in the current line for the *n*th occurrence of the next character typed.

search-history: ^R
Enter incremental search mode. The internal history list is searched backwards for commands matching the input. An initial '' in the search string anchors the search. The abort key will leave search mode. Other commands will be executed after leaving search mode. Successive **search-history** commands continue searching backward to the next previous occurrence of the pattern. The history buffer retains only a finite number of lines; the oldest are discarded as necessary.

set-mark-command: ^[⟨space⟩
Set the mark at the cursor position.

stuff: On systems supporting it, pushes the bound character back onto the terminal input where it may receive special processing by the terminal handler. This is useful for the BRL ^T mini-systat feature, for example.

stuff-reset:
Acts like **stuff**, then aborts input the same as an interrupt.

transpose-chars: ^T
If at the end of line, or if the **gmacs** option is set, this exchanges the two previous characters; otherwise, it exchanges the previous and current characters and moves the cursor one character to the right.

up-history: [*n*] ^P, ^XA
Scrolls the history buffer backward *n* lines (earlier).

upcase-word: [*n*] ^[U, ^[u
Uppercase the next *n* words.

quote: ^V
Synonym for ^^.

yank: ^Y
Inserts the most recently killed text string at the current cursor position.

yank-pop: ^[y
Immediately after a **yank**, replaces the inserted text string with the next previously killed text string.

Vi editing mode

The vi command-line editor in **ksh** has basically the same commands as the vi(1) editor with the following exceptions:

- You start out in insert mode.
- There are file name and command completion commands: =, , *, ^X, ^E, ^F, and, optionally, ⟨tab⟩ and ⟨esc⟩.
- The _ command is different (in **ksh** it is the last argument command; in vi(1) it goes to the start of the current line).
- The **/** and **G** commands move in the opposite direction to the **j** command.
- Commands which don't make sense in a single line editor are not available (e.g. screen movement commands and -style -- ex(1)Ns colon (**:**) commands).

Note that the ^X stands for control-X; also ⟨esc⟩, ⟨space⟩, and ⟨tab⟩ are used for escape, space, and tab, respectively (no kidding).

Like vi(1), there are two modes: "insert" mode and "command" mode. In insert mode, most characters are simply put in the buffer at the current cursor position as they are typed; however, some characters are treated specially. In particular, the following characters are taken from current tty(4) settings (see stty(1)) and have their usual meaning (normal values are in parentheses): kill (^U), erase (^?), werase (^W), eof (^D), intr (^C), and quit (^). In addition to the above, the following characters are also treated specially in insert mode:

^E Command and file name enumeration (see below).

^F Command and file name completion (see below). If used twice in a row, the list of possible completions is displayed; if used a third time, the completion is undone.

^H Erases previous character.

^J | ^M End of line. The current line is read, parsed, and executed by the shell.

^V Literal next. The next character typed is not treated specially (can be used to insert the characters being described here).

^X Command and file name expansion (see below).

⟨esc⟩ Puts the editor in command mode (see below).

⟨tab⟩ Optional file name and command completion (see **^F** above), enabled with **set -o vi-tabcomplete**.

In command mode, each character is interpreted as a command. Characters that don't correspond to commands, are illegal combinations of commands, or are commands that can't be carried out, all cause beeps. In the following command descriptions, an [*n*] indicates the command may be prefixed by a number (e.g. **10l** moves right 10 characters); if no number prefix is used, *n* is assumed to be 1 unless otherwise specified. The term "current position" refers to the position between the cursor and the character preceding the cursor. A "word" is a sequence of letters, digits, and underscore characters or a sequence of non-letter, non-digit, non-underscore, and non-whitespace characters (e.g. "ab2*&^" contains two words) and a "big-word" is a sequence of non-whitespace characters.

Special **ksh** vi commands:

The following commands are not in, or are different from, the normal vi file editor:

[*n*]_ Insert a space followed by the *n*th big-word from the last command in the history at the current position and enter insert mode; if *n* is not specified, the last word is inserted.

Insert the comment character ('#') at the start of the current line and return the line to the shell (equivalent to **I#^J**).

[*n*]g Like **G**, except if *n* is not specified, it goes to the most recent remembered line.

[*n*]v Edit line *n* using the vi(1) editor; if *n* is not specified, the current line is edited. The actual command executed is **fc -e ${VISUAL:-${EDITOR:-vi}}** *n*.

* and ^X Command or file name expansion is applied to the current big-word (with an appended '*' if the word contains no file globbing characters) - the big-word is replaced with the resulting words. If the current big-word is the first on the line or follows one of the characters '**;**', '**|**', '**&**', '**(**', or '**)**', and does not contain a slash ('/'), then command expansion is done; otherwise file name expansion is done. Command expansion will match the big-word against all aliases, functions, and built-in commands as well as any executable files found by searching the directories in the **PATH** parameter. File name expansion matches the big-word against the files in the current directory. After expansion, the cursor is placed just past the last word and the editor is in insert mode.

[*n*], [*n*]^F, [*n*]⟨tab⟩, and [*n*]⟨esc⟩
Command/file name completion. Replace the current big-word with the longest unique match obtained after performing command and file name expansion. ⟨tab⟩ is only recognized if the **vi-tabcomplete** option is set, while ⟨esc⟩ is only recognized if the **vi-esccomplete** option is set (see **set -o**). If *n* is specified, the *n*th possible completion is selected (as reported by the command/file name enumeration command).

= and ^E Command/file name enumeration. List all the commands or files that match the current big-word.

@*c* Macro expansion. Execute the commands found in the alias _*c*.

^L Clear the screen leaving the current line at the top of the screen.

Intra-line movement commands:

[*n*]h and [*n*]^H
Move left *n* characters.

[*n*]l and [*n*]⟨space⟩
Move right *n* characters.

0 Move to column 0.

^ Move to the first non-whitespace character.

[*n*]| Move to column *n*.

$ Move to the last character.

[*n*]b Move back *n* words.

[*n*]B Move back *n* big-words.

[*n*]e Move forward to the end of the word, *n* times.

[*n*]E Move forward to the end of the big-word, *n* times.

[*n*]w Move forward *n* words.

[*n*]W Move forward *n* big-words.

% Find match. The editor looks forward for the nearest parenthesis, bracket, or brace and then moves the cursor to the matching parenthesis, bracket, or brace.

[*n*]f*c* Move forward to the *n*th occurrence of the character *c*.

[*n*]F*c* Move backward to the *n*th occurrence of the character *c*.

[*n*]t*c* Move forward to just before the *n*th occurrence of the character *c*.

[*n*]T*c* Move backward to just before the *n*th occurrence of the character *c*.

[*n*]; Repeats the last **f**, **F**, **t**, or **T** command.

[*n*], Repeats the last **f**, **F**, **t**, or **T** command, but moves in the opposite direction.

Inter-line movement commands:

[*n*]j, [*n*]+, and [*n*]^N
Move to the *n*th next line in the history.

[*n*]k, [*n*]-, and [*n*]^P
Move to the *n*th previous line in the history.

[*n*]G Move to line *n* in the history; if *n* is not specified, the number of the first remembered line is used.

[*n*]g Like **G**, except if *n* is not specified, it goes to the most recent remembered line.

[*n*]/*string*
Search backward through the history for the *n*th line containing *string*; if *string* starts with ‘’, the remainder of the string must appear at the start of the history line for it to

match.

[*n*]?*string*
Same as **/**, except it searches forward through the history.

[*n*]n Search for the *n*th occurrence of the last search string; the direction of the search is the same as the last search.

[*n*]N Search for the *n*th occurrence of the last search string; the direction of the search is the opposite of the last search.

Edit commands

[*n*]a Append text *n* times; goes into insert mode just after the current position. The append is only replicated if command mode is re-entered i.e. ⟨esc⟩ is used.

[*n*]A Same as **a**, except it appends at the end of the line.

[*n*]i Insert text *n* times; goes into insert mode at the current position. The insertion is only replicated if command mode is re-entered i.e. ⟨esc⟩ is used.

[*n*]I Same as **i**, except the insertion is done just before the first non-blank character.

[*n*]s Substitute the next *n* characters (i.e. delete the characters and go into insert mode).

S Substitute whole line. All characters from the first non-blank character to the end of the line are deleted and insert mode is entered.

[*n*]c*move-cmd*
Change from the current position to the position resulting from *n move-cmd*s (i.e. delete the indicated region and go into insert mode); if *move-cmd* is **c**, the line starting from the first non-blank character is changed.

C Change from the current position to the end of the line (i.e. delete to the end of the line and go into insert mode).

[*n*]x Delete the next *n* characters.

[*n*]X Delete the previous *n* characters.

D Delete to the end of the line.

[*n*]d*move-cmd*
Delete from the current position to the position resulting from *n move-cmd*s; *move-cmd* is a movement command (see above) or **d**, in which case the current line is deleted.

[*n*]r*c* Replace the next *n* characters with the character *c*.

[*n*]R Replace. Enter insert mode but overwrite existing characters instead of inserting before existing characters. The replacement is repeated *n* times.

[*n*]~ Change the case of the next *n* characters.

[*n*]y*move-cmd*
Yank from the current position to the position resulting from *n move-cmd*s into the yank buffer; if *move-cmd* is **y**, the whole line is yanked.

Y Yank from the current position to the end of the line.

[*n*]p Paste the contents of the yank buffer just after the current position, *n* times.

[*n*]P Same as **p**, except the buffer is pasted at the current position.

Miscellaneous vi commands

^J and ^M
The current line is read, parsed, and executed by the shell.

^L and ^R
Redraw the current line.

[*n*]. Redo the last edit command *n* times.

u Undo the last edit command.

U Undo all changes that have been made to the current line.

intr and *quit*
The interrupt and quit terminal characters cause the current line to be deleted and a new prompt to be printed.

FILES

`/.profile`	User's login profile.
`/etc/ksh.kshrc`	Global configuration file. Not sourced by default.
`/etc/profile`	System login profile.
`/etc/shells`	Shell database.
`/etc/suid_profile`	Privileged shell profile.

SEE ALSO

ed(1), `ksh-sh`(1), stty(1), vi(1), shells(5), environ(7), `script`(7)

Morris Bolsky and David Korn, *The KornShell Command and Programming Language, 2nd Edition*, *Prentice Hall*, 1995, ISBN 0131827006.

Stephen G. Kochan and Patrick H. Wood, *UNIX Shell Programming, 3rd Edition*, *Sams*, 2003, ISBN 0672324903.

IEEE Inc., *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities*, 1993, ISBN 1-55937-266-9.

VERSION

This page documents version @(#)PD KSH v5.2.14 99/07/13.2 of the public domain Korn shell.

AUTHORS

This shell is based on the public domain 7th edition Bourne shell clone by Charles Forsyth and parts of the BRL shell by Doug A. Gwyn, Doug Kingston, Ron Natalie, Arnold Robbins, Lou Salkind, and others. The first release of **pdksh** was created by Eric Gisin, and it was subsequently maintained by John R. MacMillan<`change!john@sq.sq.com`>, Simon J. Gerrat y <`sjg@zen.void.oz.au`>, and Michael Rendell <`michael@cs.mun.ca`>. The`CONTRIBUTORS` file in the source distribution contains a more complete list of people and their part in the shell's development.

BUGS

$(*command*) expressions are currently parsed by finding the closest matching (unquoted) parenthesis. Thus constructs inside $(*command*) may produce an error. For example, the parenthesis in `x);;` is interpreted as the closing parenthesis in `$(case x in x);; *);; esac)`.