NAME

mksh, **sh**—MirBSD Korn shell

SYNOPSIS

mksh [**-+abCefhiklmnprUuvXx**][**-T** [*!*]*tty*|*-*][**-+o** *option*][**-c** *string* | **-s** |*file*[*argument ...*]]
builtin-name [*argument ...*]

DESCRIPTION

mksh is a command interpreter intended for both interactive and shell script use. Its command language is a superset of the sh(C) shell language and largely compatible to the original Korn shell. At times, this manual page may give scripting advice; while it sometimes does take portable shell scripting or various standards into account all information is first and foremost presented with **mksh** in mind and should be taken as such.

I use Android, OS/2, etc. so what...?

Please refer to: http://www.mirbsd.org/mksh-faq.htm#sowhatismksh

Invocation

Most builtins can be called directly, for example if a link points from its name to the shell; not all make sense, have been tested or work at all though.

The options are as follows:

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

-i
: Interactive shell. A shell that reads commands from standard input 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). It also processes the ENV parameter or the mkshrc file (see below). For non-interactive shells, the **trackall** option is on by default (see the **set** command below).

-l
: Login shell. If the name or 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; see Startup files below.

-p
: Privileged shell. A shell is "privileged" if the real user ID or group ID does not match the effective user ID or group ID (see getuid(2) and getgid(2)). Clearing the privileged option causes the shell to set its effective user ID (group ID) to its initial real user ID (group ID). For further implications, see Startup files. If the shell is privileged and this flag is not explicitly set, the "privileged" option is cleared automatically after processing the startup files.

-r
: Restricted shell. A shell is "restricted" if the basename the shell is called with, after '-' processing, starts with 'r' or if this option is used. The following restrictions come into effect after the shell processes any profile and ENV files:

  - The **cd** (and **chdir**) 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.

-T *name* Spawn **mksh** on the tty(4) device given. The paths *name*, /dev/tyC*name* and /dev/tty*name* are attempted in order. Unless *name* begins with an exclamation mark ('!'), this is done in a subshell and returns immediately. If *name* is a dash ('-'), detach from controlling terminal (daemonise) instead.

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 name the shell was called with (i.e. argv[0]) is used.

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.

Startup files

For the actual location of these files, see FILES. A login shell processes the system profile first. A privileged shell then processes the suid profile. A non-privileged login shell processes the user profile next. A non-privileged interactive shell checks the value of the ENV parameter after subjecting it to parameter, command, arithmetic and tilde ('~') substitution; if unset or empty, the user mkshrc profile is processed; otherwise, if a file whose name is the substitution result exists, it is processed; non-existence is silently ignored. A privileged shell then drops privileges if neither was the **-p** option given on the command line nor set during execution of the startup files.

Command syntax

The shell begins parsing its input by removing any backslash-newline combinations, then 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; ";;", ";&" and ";|" are used in **case** statements; "((...))" 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. Note that 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)-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 programmes that are executed, and *compound-commands*, such as **for** and **if** state-

ments, 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). Note that 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, unless the **pipefail** option is set (see there). All commands of a pipeline are executed in separate subshells; this is allowed by POSIX but differs from both variants of AT&T UNIX **ksh**, where all but the last command were executed in subshells; see the **read** builtin's description for implications and workarounds. 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. Note that 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). Note that 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 recognised 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    else    function    then    !    (
do      esac    if          time    [[   ((
done    fi      in          until   {
elif    for     select      while   }
```

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 }
```

case *word* **in** [[(] *pattern* [| *pattern*] ...) *list* <terminator>] ... **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**, for example: "`case $foo { (ba[rz]|blah) date ;; }`"

The list <terminator>s are:

"`;;`" Terminate after the list.

"`;&`" Fall through into the next list.

"`;|`" Evaluate the remaining pattern-list tuples.

The exit status of a **case** statement 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. The exit status of a **for** statement is the last exit status of *list*; if *list* is never executed, the exit status is zero. If **in** is not used to specify a word list, the positional parameters ($1, $2, etc.) are used instead; in this case, use a newline instead of the semicolon ('`;`') for portability. For historical reasons, open and close braces may be used instead of **do** and **done**, as in "`for i; { echo $i; }`" (not portable).

function *name* { *list*; }
Defines the function *name* (see Functions below). All 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 above and Functions below). Most amounts of space and tab after *name* will be ignored.

function *name*() { *list*; }
bashism for *name*`() { ` *list*`; }` (the **function** keyword is ignored).

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** statement is that of whatever non-conditional (not the first) *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 number 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 and trailing space is stripped), and *list* is executed. If a blank line (i.e. zero or more IFS octets) is entered, the menu is reprinted without executing *list*.

When *list* completes, the enumerated list is printed if REPLY is empty, 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. The exit status of a **select** statement is zero if a **break** statement is used to exit the loop, non-zero otherwise. If "**in** *word ...*" is omitted, the positional parameters are used. For historical reasons, open and close braces may be used instead of **do** and **done**, as in: "`select i; { echo $i; }`"

time [**-p**] [*pipeline*]
The Command execution section describes the **time** reserved word.

until *list*; **do** *list*; **done**
This works like **while** (see below), except that the body *list* 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 *list* 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.

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

- Field splitting and globbing are not performed on arguments.
- The **-a** (AND) and **-o** (OR) operators are replaced, respectively, with "**&&**" and "||".
- Operators (e.g. "**-f**", "**=**", "**!**") must be unquoted.
- 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 ]]
```

- The second operand of the "**=**" and "**!=**" expressions is a pattern (e.g. the comparison **[[foobar = f*r]]** succeeds). This even works indirectly, while quoting forces literal interpretation:

```
$ bar=foobar; baz='f*r'          # or: baz='f+(o)b?r'
$ [[ $bar = $baz ]]; echo $?     # 0
$ [[ $bar = "$baz" ]]; echo $?   # 1
```

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

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

((*expression*))
The arithmetic expression *expression* is evaluated; equivalent to '`let "`*expression*`"`' in a compound construct.
See the **let** command and Arithmetic expressions below.

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 ‘$’, ‘\’ and ‘`’, up to the next unescaped double quote. ‘$’ and ‘`’ inside double quotes have their usual meaning (i.e. parameter, arithmetic or command substitution) except no field splitting is carried out on the results of double-quoted substitutions, and the old-style form of command substitution has backslash-quoting for double quotes enabled. If a ‘\’ inside a double-quoted string is followed by ‘"’, ‘$’, ‘\’ or ‘`’, only the ‘\’ is removed, i.e. the combination 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.

If a single-quoted string is preceded by an unquoted ‘$’, C style backslash expansion (see below) is applied (even single quote characters inside can be escaped and do not terminate the string then); the expanded result is treated as any other single-quoted string. If a double-quoted string is preceded by an unquoted ‘$’, the ‘$’ is simply ignored.

Backslash expansion

In places where backslashes are expanded, certain C and AT&T UNIX **ksh** or GNU **bash** style escapes are translated. These include “\a”, “\b”, “\f”, “\n”, “\r”, “\t”, “\U########”, “\u####” and “\v”. For “\U########” and “\u####”, ‘#’ means a hexadecimal digit (up to 4 or 8); these translate a Universal Coded Character Set codepoint to UTF-8 (see CAVEATS on UCS limitations). Furthermore, “\E” and “\e” expand to the escape character.

In the **print** builtin mode, octal sequences must have the optional up to three octal digits ‘#’ prefixed with the digit zero (“\0###”); hexadecimal sequences “\x##” are limited to up to two hexadecimal digits ‘#’; both octal and hexadecimal sequences convert to raw octets; “\%”, where ‘%’ is none of the above, translates to \% (backslashes are retained).

In C style mode, raw octet-yielding octal sequences “\###” must not have the one up to three octal digits prefixed with the digit zero; hexadecimal sequences “\x##” greedily eat up as many hexadecimal digits ‘#’ as they can and terminate with the first non-xdigit; below \x100 these produce raw octets; above, they are equivalent to “\U#”. The sequence “\c%”, where ‘%’ is any octet, translates to **Ctrl-%**, that is, “\c?” becomes DEL, everything else is bitwise ANDed with 0x9F. “\%”, where ‘%’ is none of the above, translates to %: backslashes are trimmed even before newlines.

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. Aliases are specifically an interactive feature: while they do happen to work in scripts and on the command line in some cases, aliases are expanded during lexing, so their use must be in a separate command tree from their definition; otherwise, the alias will not be found. Noticeably, command lists (separated by semicolon, in command substitutions also by newline) may be one same parse tree.

The following command aliases are defined automatically by the shell:

```
autoload='\\builtin typeset -fu'
functions='\\builtin typeset -f'
hash='\\builtin alias -t'
history='\\builtin fc -l'
```

```
integer='\\builtin typeset -i'
local='\\builtin typeset'
login='\\builtin exec login'
nameref='\\builtin typeset -n'
nohup='nohup '
r='\\builtin fc -e -'
type='\\builtin 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), 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 (deprecated) `*command*` or (executed in the current environment) ${ *command*;} and strip trailing newlines; and arithmetic substitutions take the form $((*expression*)). Parsing the current-environment command substitution requires a space, tab or newline after the opening brace and that the closing brace be recognised as a keyword (i.e. is preceded by a newline or semicolon). They are also called funsubs (function substitutions) and behave like functions in that **local** and **return** work, and in that **exit** terminates the parent shell; shell options are shared.

Another variant of substitution are the valsubs (value substitutions) ${|*command*;} which are also executed in the current environment, like funsubs, but share their I/O with the parent; instead, they evaluate to whatever the, initially empty, expression-local variable REPLY is set to within the *command*s.

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 octets which are used to break a string up into several words; any octets from the set space, tab and newline that appear in the IFS octets are called "IFS whitespace". Sequences of one or more IFS whitespace octets, in combination with zero or one non-IFS whitespace octets, 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 or trailing 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 empty string, no field splitting is done; if it 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*) and ${|*command*;} and ${ *command*;} substitutions, normal quoting rules are used when *command* is parsed; however, for the deprecated `*command*` form, a '\' followed by any of '$', '`' or '\' is stripped (as is '"' when the substitution is part of a double-quoted string); a backslash '\' 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)**.

Note that some shells do not use a recursive parser for command substitutions, leading to failure for certain constructs; to be portable, use as workaround "`x=$(cat) <<\EOF`" (or the newline-keeping "`x=<<\EOF`" extension) instead to merely slurp the string. IEEE Std 1003.1 ("POSIX.1") recommends using case statements of the form `x=$(case $foo in (bar) echo $bar ;; (*) echo $baz ;; esac)` instead, which would work but not serve as example for this portability issue.

```
x=$(case $foo in bar) echo $bar ;; *) echo $baz ;; esac)
# above fails to parse on old shells; below is the workaround
x=$(eval $(cat)) <<\EOF
case $foo in bar) echo $bar ;; *) echo $baz ;; esac
EOF
```

Arithmetic substitutions are replaced by the value of the specified expression. For example, the command **print $((2+3*4))** displays 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. Array indices in **mksh** are limited to the range 0 through 4294967295, inclusive. That is, they are a 32-bit unsigned integer.

Parameter substitutions take the form $*name*, ${*name*} or ${*name*[*expr*]} where *name* is a parameter name. Substitutions of an array in scalar context, i.e. without an *expr* in the latter form mentioned above, expand the element with the key "0". Substitution of all array elements with ${*name*[*]} and ${*name*[@]} works equivalent to $* and $@ for positional parameters. If substitution is performed on a parameter (or an array parameter element) that is not set, an empty string is substituted unless the **nounset** option (**set -u**) is set, in which 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 recognise a parameter assignment. The construct **FOO+=baz** is also recognised; the old and new values are string-concatenated with no separator. 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 empty, it is substituted; otherwise, *word* is substituted.

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

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

${*name*:?*word*}
: If *name* is set and not empty, 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 a script sourced using the "`.`" built-in). If *word* is omitted, the string "`parameter null or not set`" is used instead.

Note that, for all of the above, *word* is actually considered quoted, and special parsing rules apply. The parsing rules also differ on whether the expression is double-quoted: *word* then uses double-quoting rules, except for the double quote itself (‘"’) and the closing brace, which, if backslash escaped, gets quote removal applied.

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 empty). 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 (in characters) of the string value of parameter *name*.

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

${%*name*}
: The width (in screen columns) of the string value of parameter *name*, or -1 if ${*name*} contains a control character.

${!*name*}
: The name of the variable referred to by *name*. This will be *name* except when *name* is a name reference (bound variable), created by the **nameref** command (which is an alias for **typeset -n**). *name* cannot be one of most special parameters (see below).

${!*name*[*]}
${!*name*[@]}
: The names of indices (keys) 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 ${...#...} but deletes from the end of the value.

${*name*/*pattern*/*string*}
${*name*/#*pattern*/*string*}
${*name*/%*pattern*/*string*}
${*name*//*pattern*/*string*}
: The longest match of *pattern* in the value of parameter *name* is replaced with *string* (deleted if *string* is empty; the trailing slash ('/') may be omitted in that case). A leading slash followed by '#' or '%' causes the pattern to be anchored at the beginning or end of the value, respectively; empty unanchored *pattern*s cause no replacement; a single leading slash or use of a *pattern* that matches the empty string causes the replacement to happen only once; two leading slashes cause all occurrences of matches in the value to be replaced. May be slow on long strings.

${*name*@/*pattern*/*string*}
: The same as ${*name*//*pattern*/*string*}, except that both *pattern* and *string* are expanded anew for each iteration. Use with KSH_MATCH.

${*name*:*pos*:*len*}
: The first *len* characters of *name*, starting at position *pos*, are substituted. Both *pos* and :*len* are optional. If *pos* is negative, counting starts at the end of the string; if it is omitted, it defaults to 0. If *len* is omitted or greater than the length of the remaining string, all of it is substituted. Both *pos* and *len* are evaluated as arithmetic expressions.

${*name*@#}
: The hash (using the BAFH algorithm) of the expansion of *name*. This is also used internally for the shell's hashtables.

${*name*@Q}
: A quoted expression safe for re-entry, whose value is the value of the *name* parameter, is substituted.

Note that *pattern* may need extended globbing pattern (@(...)), single ('...') or double ("...") quote escaping unless **-o sh** is set.

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, if it is a subshell, the PID of the original shell. 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, but at most 255.

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

@ 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 empty arguments or splitting arguments with spaces (IFS, actually).

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.

BASHPID The PID of the shell or subshell.

CDPATH Like PATH, but used to resolve the argument to the **cd** built-in command. Note that if CDPATH is set and does not contain "." or an empty string element, 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. If never unset and not imported, always set dynamically; unless the value as reported by stty(1) is non-zero and sane enough (minimum is 12x3), defaults to 80; similar for LINES. This parameter is used by the interactive line editing modes and by the **select**, **set -o** and **kill -l** commands to format information columns. Importing from the environment or unsetting this parameter removes the binding to the actual terminal size in favour of the provided value.

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.

EPOCHREALTIME
Time since the epoch, as returned by gettimeofday(2), formatted as decimal *tv_sec* followed by a dot ('.') and *tv_usec* padded to exactly six decimal digits.

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 or unset, the file is opened, history is truncated then loaded from the file; subsequent new commands (possibly consisting of several lines) are appended once they successfully compiled. Also, several invocations of the shell will share history if their HISTFILE parameters all point to the same file.

Note: If HISTFILE is unset or empty, no history file is used. This is different from AT&T UNIX **ksh**.

HISTSIZE The number of commands normally stored for history. The default is 2047. The maximum is 65535.

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.

KSHEGID The effective group id of the shell at startup.

KSHGID The real group id of the shell at startup.

KSHUID The real user id of the shell at startup.

KSH_MATCH The last matched string. In a future version, this will be an indexed array, with indexes 1 and up capturing matching groups. Set by string comparisons (= and !=) in double-bracket test expressions when a match is found (when != returns false), by **case** when a match is encountered, and by the substitution operations ${*x*#*pat*}, ${*x*##*pat*}, ${*x*%*pat*}, ${*x*%%*pat*}, ${*x*/*pat*/*rpl*}, ${*x*/#*pat*/*rpl*}, ${*x*/%*pat*/*rpl*}, ${*x*//*pat*/*rpl*}, and ${*x*@/*pat*/*rpl*}. See the end of the Emacs editing mode documentation for an example.

KSH_VERSION The name (self-identification) and version of the shell (read-only). See also the version commands in Emacs editing mode and Vi editing mode sections, below.

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. Defaults to 24; always set, unless imported or unset. See COLUMNS.

OLDPWD The previous working directory. Unset if **cd** has not successfully changed 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 (semicolon on OS/2) 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 (semi)colon, or two adjacent ones, is treated as a "." (the current directory).

PATHSEP A colon (semicolon on OS/2), for the user's convenience.

PGRP The process ID of the shell's process group leader.

PIPESTATUS An array containing the errorlevel (exit status) codes, one by one, of the last pipeline run in the foreground.

PPID The process ID of the shell's parent.

PS1 The primary prompt for interactive shells. Parameter, command and arithmetic substitutions are performed, and '!' is replaced with the current command number (see the **fc** command below). A literal '!' can be put in the prompt by placing "!!" in PS1.

The default prompt is "$ " for non-root users, "# " for root. If **mksh** 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 **mksh** distribution comes with a sample `dot.mkshrc` containing a sophisticated example, but you might like the following one (note that ${HOSTNAME:=$(hostname)} and the root-vs-user distinguishing clause are (in this example) executed at PS1 assignment time, while the $USER and $PWD are escaped and thus will be evaluated each time a prompt is displayed):

```
PS1='${USER:=$(id -un)}'"@${HOSTNAME:=$(hostname)}:\$PWD $(
        if (( USER_ID )); then print \$; else print \#; fi) "
```

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 prefixing your prompt with a character (such as Ctrl-A) followed by a carriage return and then delimiting the escape codes with this character. Any occurrences of that character in the prompt are not printed. By the way, don't blame me for this hack; it's derived from the original ksh88(1), which did print the delimiter character so you were out of luck if you did not have any non-printing characters.

Since backslashes and other special characters may be interpreted by the shell, to set PS1 either escape the backslash itself or use double quotes. The latter is more practical. This is a more complex example, avoiding to directly enter special characters (for example with **^V** in the emacs editing mode), which embeds the current working directory, in reverse video (colour would work, too), in the prompt string:

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

Due to a strong suggestion from David G. Korn, **mksh** now also supports the following form:

```
PS1=$'\1\r\1\e[7m\1$PWD\1\e[0m\1> '
```

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 "+ ". You may want to set it to "`[$EPOCHREALTIME] `" instead, to include timestamps.

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

RANDOM Each time RANDOM is referenced, it is assigned a number between 0 and 32767 from a Linear Congruential PRNG first.

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`.

`USER_ID` The effective user id of the shell at startup.

Tilde expansion

Tilde expansion, which is done in parallel with parameter substitution, is applied to words starting with an unquoted '~'. In parameter assignments (such as those preceding a simple-command or those occurring in the arguments of a declaration utility), 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 Korn shell, except in POSIX mode, always expands tildes after unquoted equals signs, not just in assignment context (see below), and enables tab completion for tildes after all unquoted colons during command line editing.

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 simplified 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.

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 octets.

[...] Matches any of the octets inside the brackets. Ranges of octets can be specified by separating two octets 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 octet in the octet list. Similarly, a ']' must be quoted or the first octet 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.

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

*(*pattern*|...|*pattern*)

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

+(*pattern*|...|*pattern*)
: Matches any string of octets that matches one or more occurrences of the specified patterns. Example: The pattern **+(foo|bar)** matches 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).

Note that complicated globbing, especially with alternatives, is slow; using separate comparisons may (or may not) be faster.

Note that **mksh** (and **pdksh**) never matches "." and "..", but AT&T UNIX **ksh**, Bourne **sh** and GNU **bash** do.

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 /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", but not for '"'. If multiple here documents are used on the same command line, they are saved in order.

If no *marker* is given, the here document ends at the next **<<** and substitution will be performed. If *marker* is only a set of either single "' '" or double '""' quotes with nothing in between, the here document ends at the next empty line and substitution will not be performed.

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

<<<*word* Same as **<<**, except that *word is* the here document. This is called a here string.

<&*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.

&>*file* Same as **>***file 2>&1*. This is a deprecated (legacy) GNU **bash** extension supported by **mksh** which also supports the preceding explicit fd digit, for example, **3&>***file* is the same as **3>***file 2>&3* in **mksh** but a syntax error in GNU **bash**.

&>|*file*, &>>*file*, &>&*fd*
Same as **>|***file*, **>>***file* or **>&***fd*, followed by **2>&1**, as above. These are **mksh** extensions.

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 | pr -n -t
```

File descriptors created by I/O redirections are private to the shell.

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. *Warning*: This also affects implicit conversion to integer, for example as done by the **let** command. *Never* use unchecked user input, e.g. from the environment, in an arithmetic context!

Expressions are calculated using signed arithmetic and the `mksh_ari_t` type (a 32-bit signed integer), unless they begin with a sole '#' character, in which case they use `mksh_uari_t` (a 32-bit unsigned integer).

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:

```
( )
```

Integer constants and expressions are calculated using an exactly 32-bit wide, signed or unsigned, type with silent wraparound on integer overflow. Integer constants may be specified with arbitrary bases using the notation *base*#*number*, where *base* is a decimal integer specifying the base (up to 36), and *number* is a number in the specified base. Additionally, base-16 integers may be specified by prefixing them with "0x" (case-insensitive) in all forms of arithmetic expressions, except as numeric arguments to the **test** built-in utility. Prefixing numbers with a sole digit zero ("0") does not cause interpretation as octal (except in POSIX mode, as required by the standard), as that's unsafe to do.

As a special **mksh** extension, numbers to the base of one are treated as either (8-bit transparent) ASCII or Universal Coded Character Set codepoints, depending on the shell's **utf8-mode** flag (current setting). The AT&T UNIX **ksh93** syntax of "'x'" instead of "1#x" is also supported. Note that NUL bytes (integral value of zero) cannot be used. An unset or empty parameter evaluates to 0 in integer context. If 'x' isn't comprised of exactly one valid character, the behaviour is undefined (usually, the shell aborts with a parse error, but rarely, it succeeds, e.g. on the sequence C2 20); users of this feature (as opposed to **read -a**) must validate the input first. See CAVEATS for UTF-8 mode handling.

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, greater than or equal. See **<**.

<< >>
: Shift left (right); the result is the left argument with its bits arithmetically (signed operation) or logically (unsigned expression) shifted left (right) by the amount given in the right argument.

^< ^>
: Rotate left (right); the result is similar to shift, except that the bits shifted out at one end are shifted in at the other end, instead of zero or sign bits.

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

%
: Remainder; the result is the symmetric remainder of the division of the left argument by the right. To get the mathematical modulus of "a **mod** b", use the formula "(a % b + b) % b".

<*arg1*>?<*arg2*>:<*arg3*>
: If <*arg1*> is non-zero, the result is <*arg2*>; otherwise the result is <*arg3*>. The non-result argument is not evaluated.

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: **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 command, functions are searched after special built-in commands, before builtins and 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 using PATH, 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. The "export" attribute of functions is currently not used.

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 AT&T UNIX **ksh93** uses static scoping (one global scope, one local scope per function) and allows local variables only on Korn style functions, whereas **mksh** uses dynamic scoping (nested scopes of varying locality). 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. Note that when called in a subshell, **return** will only exit that subshell and will not cause the original shell to exit a running function (see the **while**...**read** loop FAQ).

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).
- 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).
- Shell options (**set -o**) have local scope, i.e. changes inside a function are reset upon its exit.

In the future, the following differences may also be added:

- A separate trap/signal environment will be used during the execution of functions. This will mean that traps set inside a function will not affect the shell's traps and signals that are not ignored in the shell (but may be trapped) will have their default effect in a function.
- The EXIT trap, if set in a function, will be executed after the function returns.

Command execution

After evaluation of command-line arguments, redirections and parameter assignments, the type of command is determined: a special built-in command, a function, a normal builtin 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. Regular built-in commands are different only in that the PATH parameter is not used to find them.

POSIX special built-in utilities:

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

Additional **mksh** commands keeping assignments:

source, **typeset**

All other builtins are not special; these are at least:

[, **alias**, **bg**, **bind**, **builtin**, **cat**, **cd**, **command**, **echo**, **false**, **fc**, **fg**, **getopts**, **jobs**, **kill**, **let**, **print**, **pwd**, **read**, **realpath**, **rename**, **sleep**, **suspend**, **test**, **true**, **ulimit**, **umask**, **unalias**, **wait**, **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 and builtin-like reserved words, as well as some optional utilities:

. *file* [*arg ...*]
: (keeps assignments, special) This is called the "dot" command. 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.

: [...]
: (keeps assignments, special) The null command.
Exit status is set to zero.

Lb64decode [*string*]
: (dot.mkshrc function) Decode *string* or standard input to binary.

Lb64encode [*string*]
: (dot.mkshrc function) Encode *string* or standard input as base64.

Lbafh_init
Lbafh_add [*string*]
Lbafh_finish
: (dot.mkshrc functions) Implement the Better Avalance for the Jenkins Hash. This is the same hash **mksh** currently uses internally. After calling **Lbafh_init**, call **Lbafh_add** multiple times until all input is read, then call **Lbafh_finish**, which writes the result to the unsigned integer *Lbafh_v* variable for your consumption.

Lstripcom [*file ...*]
(dot.mkshrc function) Same as **cat** but strips any empty lines and comments (from any '#' character onwards, no escapes) and reduces any amount of whitespace to one space character.

[*expression* **]**
(regular) See **test**.

alias [**-d** | **-t** [**-r**] | **-+x**] [**-p**] [**+**] [*name*[=*value*] ...]
(regular) 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. `[][A-Za-z0-9_!%+,.@:-]` are valid in names, except they may not begin with a plus or hyphen-minus, and **[[** is not a valid alias name.

When listing aliases, one of two formats is used. Normally, aliases are listed as *name*=*value*, where *value* is quoted as necessary. 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).

With **-p**, each alias is listed with the string "`alias `" prefixed.

The **-t** option indicates that tracked aliases are to be listed/set (values given with the command 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).

autoload
(built-in alias) See Functions above.

bg [*job ...*]
(regular, needs job control) 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
(regular) The names of editing commands strings can be bound to are listed. See Emacs editing mode for more information.

bind [*string ...*]
The current bindings, for *string*, if given, else all, are listed. *Note:* Default prefix bindings (1=Esc, 2=^X, 3=NUL) assumed.

bind *string*=[*editing-command*] [*...*]
bind -m *string*=*substitute* [*...*]
To *string*, which should consist of a control character optionally preceded by one of the three prefix characters and optionally succeeded by a tilde character, the *editing-command* is bound so that future input of the *string* will immediately invoke that editing command. If a tilde postfix is given, a tilde trailing the control character is ignored. If **-m** (macro) is given, future input of the *string* will be replaced by the given NUL-terminated *substitute* string, wherein prefix/control/tilde characters mapped to editing commands (but not those mapped to other macros) will be processed.

Prefix and control characters may be written using caret notation, i.e. ^Z represents Ctrl-Z. Use a backslash to escape the caret, an equals sign or another backslash. Note that, although only three prefix characters (usually Esc, ^X and NUL) are supported, some multi-character sequences can be supported.

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

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

\builtin *command* [*arg ...*]
(regular, decl-forwarder) Same as **builtin**. Additionally acts as declaration utility forwarder, i.e. this is a declaration utility (see Tilde expansion) iff *command* is a declaration utility.

cat [**-u**] [*file ...*]
(defer with flags) Copy files in command line order to standard output. If a *file* is a single dash ("-") or absent, read from standard input. For direct builtin calls, the POSIX **-u** option is supported as a no-op. For calls from shell, if any options are given, an external cat(1) utility is preferred over the builtin.

cd [**-L**] [*dir*]
cd -P [**-e**] [*dir*]
chdir [**-eLP**] [*dir*]
(regular) Set the working directory to *dir*. If the parameter CDPATH is set, it lists the search path for the directory containing *dir*. An unset or empty path means the current directory. If *dir* is found in any component of the CDPATH search path other than an unset or empty 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 directory. 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. If the **-e** option is set for physical filesystem traversal and PWD could not be set, the exit code is 1; greater than 1 if an error occurred, 0 otherwise.

cd [**-eLP**] *old new*
chdir [**-eLP**] *old new*
(regular) The string *new* is substituted for *old* in the current directory, and the shell attempts to change to the new directory.

cls (dot.mkshrc alias) Reinitialise the display (hard reset).

command [**-pVv**] *cmd* [*arg ...*]
(regular, decl-forwarder) 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 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, whose actual value is system-dependent, is used instead of the current PATH.

If the **-v** option is given, instead of executing *cmd*, information about what would be executed is given for each argument. For builtins, functions and keywords, their names are simply printed; for aliases, a command that defines them is printed; for utilities 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, but more verbose.

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

dirs [**-lnv**]
(dot.mkshrc function) Print the directory stack. **-l** causes tilde expansion to occur in the output. **-n** causes line wrapping before 80 columns, whereas **-v** causes numbered vertical output.

doch (dot.mkshrc alias) Execute the last command with sudo(8).

echo [**-Een**] [*arg ...*]
(regular) *Warning:* this utility is not portable; use the standard Korn shell built-in utility **print** in new code instead.

Print arguments, separated by spaces, followed by a newline, to 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 recognised.

The options are provided for compatibility with BSD shell scripts. The **-E** option suppresses backslash interpretation, **-e** enables it (normally default), **-n** suppresses the trailing newline, and anything else causes the word to be printed as argument instead.

If the **posix** or **sh** option is set or this is a direct builtin call or **print -R**, only the first argument is treated as an option, and only if it is exactly "-n". Backslash interpretation is disabled.

enable [**-anps**] [*name ...*]
(dot.mkshrc function) Hide and unhide built-in utilities, aliases and functions and those defined in dot.mkshrc.

If no *name* is given or the **-p** option is used, builtins are printed (behind the string "enable ", followed by "-n " if the builtin is currently disabled), otherwise, they are disabled (if **-n** is given) or re-enabled.

When printing, only enabled builtins are printed by default; the **-a** options prints all builtins, while **-n** prints only disabled builtins instead; **-s** limits the list to POSIX special builtins.

eval *command ...*
(keeps assignments, special) The arguments are concatenated, with a space between each, to form a single string which the shell then parses and executes in the current execution environment.

exec [**-a** *argv0*] [**-c**] [*command* [*arg ...*]]
(keeps assignments, special) The command (with arguments) is executed without forking, fully replacing the shell process; this is absolute, i.e. **exec** never returns, even if the *command* is not found. The **-a** option permits setting a different argv[0] value, and **-c** clears the environment before executing the child process, except for the _ parameter and direct assignments.

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 dup(2)'d 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*]
(keeps assignments, special) The shell or subshell exits with the specified errorlevel (or the current value of the *$?* parameter).

export [**-p**] [*parameter*[=*value*]]
(keeps assignments, special, decl-util) 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. This is a declaration utility.

If no parameters are specified, all parameters with the export attribute set are printed one per line: either their names, or, if a "−" with no option letter is specified, name=value pairs, or, with the **−p** option, **export** commands suitable for re-entry.

extproc
(OS/2) Null command required for shebang-like functionality.

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

fc [**−e** *editor* | **−l** [**−n**]] [**−r**] [*first* [*last*]]
(regular) *first* and *last* select commands from the history. Commands can be selected by history number (negative numbers go backwards from the current, most recent, line) 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 the result is executed by the shell.

fc −e − | **−s** [**−g**] [*old*=*new*] [*prefix*]
(regular) Re-execute the selected command (the previous command by default) after performing the optional substitution of *old* with *new*. If **−g** is specified, all occurrences of *old* are replaced with *new*. The meaning of **−e −** and **−s** is identical: re-execute the selected command without invoking an editor. This command is usually accessed with the predefined: **alias r='fc −e −'**

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

functions [*name ...*]
(built-in alias) Display the function definition commands corresponding to the listed, or all defined, functions.

getopts *optstring name* [*arg ...*]
(regular) Used by shell procedures to parse the specified arguments (or positional parameters, if no arguments are given) and to check for legal options. 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 word it is found in, the remainder of the word is taken to be the option's argument; otherwise, the next word is the option's argument.

optstring contains the option letters to be recognised. If a letter is followed by a colon, the option takes an argument.

Each time **getopts** is invoked, it places the next option in the shell parameter *name*. If the option was introduced with a '+', the character placed in *name* is prefixed with a '+'. If the option takes an argument, it is placed 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 letter that caused the problem. Furthermore, unless *optstring* begins with a colon, a question mark is placed in *name*, OPTARG is unset and a diagnostic is shown on standard error.

getopts records the index of the argument to be processed by the next call in OPTIND. When the end of the options is encountered, **getopts** returns a non-zero exit status. Options end at the first argument that does not start with a '−' (non-option argument) 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 ...*]
(built-in alias) Without arguments, any hashed executable command paths are listed. The **-r** option causes all hashed commands to be removed from the cache. Each *name* is searched as if it were a command name and added to the cache if it is an executable command.

hd [*file ...*]
(dot.mkshrc alias or function) Hexdump stdin or arguments legibly.

history [**-nr**] [*first* [*last*]]
(built-in alias) Same as **fc -l** (see above).

integer [flags] [*name*[=*value*] *...*]
(built-in alias) Same as **typeset -i** (see below).

jobs [**-lnp**] [*job ...*]
(regular) 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* } *...*
(regular) 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 ...*]
(regular) Print the signal name corresponding to *exit-status*. If no arguments are specified, a list of all the signals with their numbers and a short description of each are printed.

let [*expression ...*]
(regular) Each expression is evaluated (see Arithmetic expressions above). If all expressions evaluate successfully, 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:

```
{ \\builtin let 'expr'; }
```

local [flags] [*name*[=*value*] *...*]
(built-in alias) Same as **typeset** (see below).

mknod [**-m** *mode*] *name* **b**|**c** *major minor*
mknod [**-m** *mode*] *name* **p**
(optional) Create a device special file. The file type may be one of **b** (block type device), **c** (character type device) or **p** (named pipe, FIFO). The file created may be modified according to its *mode* (via the **-m** option), *major* (major device number), and *minor* (minor device number). This is not normally part of **mksh**; however, distributors may have added this as builtin as a speed hack.

nameref [flags] [*name*[=*value*] *...*]
(built-in alias) Same as **typeset -n** (see below).

popd [**-lnv**] [**+***n*]
(dot.mkshrc function) Pops the directory stack and returns to the new top directory. The flags are as in **dirs** (see above). A numeric argument **+***n* selects the entry in the stack to discard.

print [**-AcelNnprsu**[*n*]| **-R** [**-n**]] [*argument ...*]
(regular) Print the specified argument(s) on the standard output, separated by spaces, terminated with a newline. The escapes mentioned in Backslash expansion above, as well as "\c", which is equivalent to using the **-n** option, are interpreted.

The options are as follows:

-A Each *argument* is arithmetically evaluated; the character corresponding to the resulting value is printed. Empty *argument*s separate input words.

-c The output is printed columnised, line by line, similar to how the rs(1) utility, tab completion, the **kill -l** built-in utility and the **select** statement do.

-e Restore backslash expansion after a previous **-r**.

-l Change the output word separator to newline.

-N Change the output word and line separator to ASCII NUL.

-n Do not print the trailing line separator.

-p Print to the co-process (see Co-processes above).

-r Inhibit backslash expansion.

-s Print to the history file instead of standard output.

-u[*n*]
Print to the file descriptor *n* (defaults to 1 if omitted) instead of standard output.

The **-R** option mostly emulates the BSD echo(1) command which does not expand backslashes and interprets its first argument as option only if it is exactly "-n" (to suppress the trailing newline).

printf *format* [*arguments ...*]
(optional, defer always) If compiled in, format and print the arguments, supporting the bare POSIX-mandated minimum. If an external utility of the same name is found, it is deferred to, unless run as direct builtin call or from the **builtin** utility.

pushd [**-lnv**]
(dot.mkshrc function) Rotate the top two elements of the directory stack. The options are the same as for **dirs** (see above), and **pushd** changes to the topmost directory stack entry after acting.

pushd [**-lnv**] **+***n*
(dot.mkshrc function) Rotate the element number *n* to the top.

pushd [**-lnv**] *name*
(dot.mkshrc function) Push *name* on top of the stack.

pwd [**-LP**]
(regular) Print the present working directory. If no options are given, **pwd** behaves as if the **-P** option (print physical path) was used if the **physical** shell option is set, the **-L** option (print logical path) otherwise. The logical path is the path used to **cd** to the current directory; the physical path is determined from the filesystem (by following ". ." directories to the root directory).

r [**-g**] [*old*=*new*] [*prefix*]
(built-in alias) Same as **fc -e -** (see above).

read [**-A** | **-a**] [**-d** *x*] [**-N** *z* | **-n** *z*] [**-p** | **-u**[*n*]] [**-t** *n*] [**-rs**] [*p* ...]
(regular) Reads a line of input, separates the input into fields using the IFS parameter (see Substitution above) or other specified means, and assigns each field to the specified parameters *p*. If no parameters are specified, the REPLY parameter is used to store the result. If there are more parameters than fields, the extra parameters are set to the empty string or 0; if there are more fields than parameters, the last parameter is assigned the remaining fields (including the word separators).

The options are as follows:

-A Store the result into the parameter *p* (or REPLY) as array of words. Only no or one parameter is accepted.

-a Store the result, without applying IFS word splitting, into the parameter *p* (or REPLY) as array of characters (wide characters if the **utf8-mode** option is enacted, octets otherwise); the codepoints are encoded as decimal numbers by default. Only no or one parameter is accepted.

-d *x* Use the first byte of *x*, NUL if empty, instead of the ASCII newline character to delimit input lines.

-N *z* Instead of reading till end-of-line, read exactly *z* bytes. Upon EOF, a partial read is returned with exit status 1. After timeout, a partial read is returned with an exit status as if SIGALRM were caught.

-n *z* Instead of reading till end-of-line, read up to *z* bytes but return as soon as any bytes are read, e.g. from a slow terminal device, or if EOF or a timeout occurs.

-p Read from the currently active co-process (see Co-processes above for details) instead of from a file descriptor.

-u[*n*]
Read from the file descriptor number *n* (defaults to 0, i.e. standard input).
The argument must immediately follow the option character.

-t *n* Interrupt reading after *n* seconds (specified as positive decimal value with an optional fractional part). The exit status of **read** is the same as if SIGALRM were caught if the timeout occurred, but partial reads may still be returned.

-r Normally, **read** strips backslash-newline sequences and any remaining backslashes from input. This option enables raw mode, in which backslashes are retained and ignored.

-s The input line is saved to the history.

If the input is a terminal, both the **-N** and **-n** options set it into raw mode; they read an entire file if -1 is passed as *z* argument.

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: '**).

If no input is read or a timeout occurred, **read** exits with a non-zero status.

readonly [**-p**] [*parameter*[=*value*] ...]
(keeps assignments, special, decl-util) Sets the read-only attribute of the named parameters. If values are given, parameters are assigned these before disallowing writes. 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.

realpath [**--**] *name*
(defer with flags) Resolves an absolute pathname corresponding to *name*. If the resolved pathname either exists or can be created immediately, **realpath** returns 0 and prints the resolved pathname, otherwise or if an error occurs, it issues a diagnostic and returns nonzero. If *name* ends with a slash ('/'), resolving to an extant non-directory is also treated as error.

rename [**--**] *from to*
(defer always) Renames the file *from* to *to*. Both must be complete pathnames and on the same device. Intended for emergency situations (where `/bin/mv` becomes unusable); directly calls `rename`(2).

return [*status*]
(keeps assignments, special) Returns from a function or **.** script with errorlevel *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 **mksh** treats both profile and `ENV` files as **.** scripts, while the original Korn shell only treated profiles as **.** scripts.

rot13 (`dot.mkshrc` alias) ROT13-encrypts/-decrypts stdin to stdout.

set [**-+abCefhiklmnprsUuvXx**] [**-+o** *option*] [**-+A** *name*] [**--**] [*arg ...*]
(keeps assignments, special) The **set** command can be used to show all shell parameters (like **typeset -**), set (**-**) or clear (**+**) shell options, set an array parameter or the positional parameters.

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 short (if extant) and long names along with a description of what each 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. If *name* ends with a '+', the array is appended to instead.

An alternative syntax for the command **set -A foo -- a b c; set -A foo+ -- d e** which is compatible to GNU **bash** and also supported by AT&T UNIX **ksh93** is: **foo=(a b c); foo+=(d e)**

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

-b | **-o notify**
Print job notification messages asynchronously instead of just before the prompt. Only used with job control (**-m**).

-C | **-o noclobber**
Prevent > redirection from overwriting existing files. Instead, >| must be used to force an overwrite. *Note:* This is not safe to use for creation of temporary files or lockfiles due to a TOCTOU in a check allowing one to redirect output to `/dev/null` or other device files even in **noclobber** mode.

-e | **-o 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** or **while** statements. For **&&**, || and pipelines (but mind **-o pipefail**), only the status of the last command is tested.

-f | **-o noglob**
Do not expand file name patterns.

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

-i | **-o interactive**
The shell is an interactive shell. This option can only be used when the shell is invoked. See above for details.

-k | **-o keyword**
Parameter assignments are recognised anywhere in a command.

-l | **-o login**
The shell is a login shell. This option can only be used when the shell is invoked. See above for what this means.

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

-n | **-o noexec**
Do not execute any commands. Useful for checking the syntax of scripts. Ignored if reading commands from a tty.

-p | **-o 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.

If the shell is privileged, setting this flag after startup files have been processed let it go full setuid and/or setgid. Clearing this flag makes the shell drop privileges. Changing this flag resets the groups vector.

-r | **-o restricted**
The shell is a restricted shell. This option can only be used when the shell is invoked. See above for what this means.

-s | **-o 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 ASCIIbetically before assigning them to the positional parameters (or to array *name*, with **-A**).

-U | **-o utf8-mode**
Enable UTF-8 support in the Emacs editing mode and internal string handling functions. This flag is disabled by default, but can be enabled by setting it on the shell command line; is enabled automatically for interactive shells if requested at compile time, your system supports **setlocale**(*LC_CTYPE*, *""*) and optionally **nl_langinfo**(*CODESET*), or the LC_ALL, LC_CTYPE or LANG environment variables, and at least one of these returns something that matches "UTF-8" or "utf8" case-insensitively; for direct builtin calls depending on the aforementioned environ-

ment variables; or for stdin or scripts, if the input begins with a UTF-8 Byte Order Mark.

In near future, locale tracking will be implemented, which means that **set −+U** is changed whenever one of the POSIX locale-related environment variables changes.

−u | **−o nounset**
Referencing of an unset parameter, other than "$@" or "$*", is treated as an error, unless one of the '−', '+' or '=' modifiers is used.

−v | **−o verbose**
Write shell input to standard error as it is read.

−X | **−o markdirs**
Mark directories with a trailing '/' during globbing.

−x | **−o xtrace**
Print commands when they are executed, preceded by PS4.

−o bgnice
Background jobs are run with lower priority.

−o braceexpand
Enable brace expansion. This is enabled by default.

−o emacs
Enable BRL emacs-like command-line editing (interactive shells only); see Emacs editing mode. Enabled by default.

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

−o 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.

−o inherit-xtrace
Do not reset **−o xtrace** upon entering functions (default).

−o nohup
Do not kill running jobs with a SIGHUP signal when a login shell exits. Currently set by default, but this may change in the future to be compatible with AT&T UNIX **ksh**, which doesn't have this option, but does send the SIGHUP signal.

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

−o 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 **cd** and **pwd** above for more details.

−o pipefail
Make the exit status of a pipeline the rightmost non-zero errorlevel, or zero if all commands exited with zero.

-o posix
Behave closer to the standards (see POSIX mode for details). Automatically enabled if the shell invocation basename, after '-' and 'r' processing, begins with "sh" and (often used for the **lksh** binary) this autodetection feature is compiled in. As a side effect, setting this flag turns off the **braceexpand** and **utf8-mode** flags, which can be turned back on manually, and (unless both are set in the same command) **sh** mode.

-o sh
Enable kludge `/bin/sh` compatibility mode (see SH mode below for details). Automatically enabled if the basename of the shell invocation, after '-' and 'r' processing, begins with "sh" and this autodetection feature is compiled in (rather uncommon). As a side effect, setting this flag turns off the **braceexpand** flag, which can be turned back on manually, and **posix** mode (unless both are set in the same command).

-o vi
Enable `vi`(1)-like command-line editing (interactive shells only). See Vi editing mode for documentation and limitations.

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

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

-o 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. **mksh** 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** prints a command to restore the current option set, using the internal **set -o .reset** construct, which is an implementation detail; these commands are transient (only valid within the current shell session).

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. For unknown historical reasons, a lone "-" option is treated specially—it clears both the **-v** and **-x** options. If no options or arguments are given, the values of all parameters are printed (suitably quoted).

setenv [*name* [*value*]]
(`dot.mkshrc` function) Without arguments, display the names and values of all exported parameters. Otherwise, set *name*'s export attribute, and its value to *value* (empty string if none given).

shift [*number*]
(keeps assignments, special) The positional parameters *number*+1, *number*+2, etc. (*number* defaults to 1) are renamed to 1, 2, etc.

sleep *seconds*
(regular, needs `select`(2)) Suspends execution for a minimum of the *seconds* (specified as positive decimal value with an optional fractional part). Signal delivery may continue execution earlier.

smores [*file ...*]
(`dot.mkshrc` function) Simple pager: <Enter> next; 'q'+<Enter> quit

source *file* [*arg ...*]
(keeps assignments) Like `.` ("dot"), except that the current working directory is appended to the search path. (GNU **bash** extension)

suspend
(needs job control and `getsid`(2)) 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* **]**
(regular) **test** evaluates the *expression* and exits with status code 0 if true, 1 if false, or greater than 1 if there was an error. It is often used as the condition command of **if** and **while** statements. All *file* expressions, except **-h** and **-L**, follow symbolic links.

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 context dependent directory (only useful on HP-UX).
-h *file*	*file* is a symbolic link.
-k *file*	*file*'s mode has the `sticky`(7) bit set.
-L *file*	*file* is a symbolic link.
-O *file*	*file*'s owner is the shell's effective user ID.
-p *file*	*file* is a named pipe (FIFO).
-r *file*	*file* exists and is readable.
-S *file*	*file* is a `unix`(4)-domain socket.
-s *file*	*file* is not empty.
-t *fd*	File descriptor *fd* is a `tty`(4) device.
-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.

-v *name* The shell parameter *name* is set.

-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). The same can be achieved with [-o ?foo] like in AT&T UNIX **ksh93**. *option* can also be the short flag prefixed with either '–' or '+' (no logical negation), for example "–x" or "+x" instead of "`xtrace`".

string = *string* Strings are equal. In double brackets, pattern matching (R59+ using extglobs) occurs if the right-hand string isn't quoted.

string == *string*
Same as '=' (deprecated).

string != *string*
Strings are not equal. See '=' regarding pattern matching.

string > *string* First string operand is greater than second string operand.

string < *string* First string operand is less than second string operand.

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.
```

Note that a number actually may be an arithmetic expression, such as a mathematical term or the name of an integer variable:

```
x=1; [ "x" -eq 1 ]        evaluates to true
```

Note that some special rules are applied (courtesy of POSIX) if the number of arguments to **test** or inside the brackets **[...]** is less than five: if leading "!" arguments can be stripped such that only one to three arguments remain, then the lowered comparison is executed; (thanks to XSI) parentheses **\(... \)** lower four- and three-argument forms to two- and one-argument forms, respectively; three-argument forms ultimately prefer binary operations, followed by negation and parenthesis lowering; two- and four-argument forms prefer negation followed by parenthesis; the one-argument form always implies **-n**. To assume this is not necessarily portable.

Note: A common mistake is to use "`if [ $foo = bar ]`" which fails if parameter "foo" is empty or unset, if it has embedded spaces (i.e. IFS octets) or if it is a unary operator like "!" or "-n". Use tests like "`if [ x"$foo" = x"bar" ]`" instead, or the double-bracket operator (see **[[** above): "`if [[ $foo = bar ]]`" or, to avoid pattern matching, "`if [[ $foo = "$bar" ]]`"; the **[[...]]** construct is not only more secure to use but also often faster.

time [**-p**] [*pipeline*]
(reserved word) 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:

```
0m0.03s real     0m0.02s user     0m0.01s system
```

If the **-p** option is given (which is only permitted if *pipeline* is a simple command), the output is slightly longer:

```
real     0.03
user     0.02
sys      0.01
```

Simple redirections of standard error do not affect **time**'s output:

```
$ 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 (keeps assignments, special) Print the accumulated user and system times (see above) used both by the shell and by processes that the shell started which have exited. The format of the output is:

```
0m0.01s 0m0.00s
0m0.04s 0m0.02s
```

trap *n* [*signal ...*]
(keeps assignments, special) If the first operand is a decimal unsigned integer, this resets all specified signals to the default action, i.e. is the same as calling **trap** with a dash ("-") as *handler*, followed by the arguments (interpreted as signals).

trap [*handler signal ...*]
(keeps assignments, special) Sets a trap handler that is to be executed when any of the specified *signal*s are received. *handler* is either an empty string, indicating the signals are to be ignored, a dash ("-"), indicating that the default action is to be taken for the signals (see signal(3)), or a string comprised of shell commands to be 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, possibly prefixed with "SIG", of a signal (e.g. PIPE, ALRM or SIGINT) 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 **set -e** or **set -o errexit** option were set. EXIT handlers are executed in the environment of the last executed command. The original Korn shell's DEBUG trap and handling of ERR and EXIT in functions are not yet implemented.

Note that, for non-interactive shells, the trap handler cannot be changed for signals that were ignored when the shell started.

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

true (regular) A command that exits with a zero status.

type *name ...*
(built-in alias) Reveal how *name* would be interpreted as command.

typeset [**-+aglpnrtUux**][**-L**[*n*]|**-R**[*n*]|**-Z**[*n*]][**-i**[*n*]][*name*[=*value*] ...]
typeset -f [**-tux**][*name ...*]
(keeps assignments, decl-util) Display or set attributes of shell parameters or functions. With no *name* arguments, parameter attributes are shown; 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 '+' (or "+" alone), only names are printed.

If any *name* arguments are given, the attributes of the so named parameters are set (**-**) or cleared (**+**); inside a function, this will cause the parameters to be created (and set to "" if no value is given) in the local scope (except if **-g** is used). Values for parameters may optionally be specified. For *name*[*], the change affects all elements of the array, and no value may be specified.

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 names are displayed.

-a Indexed array attribute.

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

-g "global" mode. Do not cause named parameters to be created in the local scope when called inside a function.

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

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

padded with space to fit the field width.

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

-n Create a bound variable (name reference): any access to the variable *name* will access the variable *value* in the current scope (this is different from AT&T UNIX **ksh93**!) instead. Also different from AT&T UNIX **ksh93** is that *value* is lazily evaluated at the time *name* is accessed. This can be used by functions to access variables whose names are passed as parameters, instead of resorting to **eval**.

-p Print complete **typeset** commands that can be used to re-create the attributes and values of parameters.

-R[*n*] Right justify attribute. *n* specifies the field width. If *n* is not specified, the current width of the 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 padded with space to 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. This attribute has no meaning to the shell for parameters and is provided for application use.

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

-U Unsigned integer attribute. Integers are printed as unsigned values (combined with the **-i** option).

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

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

-x Export attribute. Parameters are placed in the environment of any executed commands. Functions cannot be exported for security reasons ("shellshock").

-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. For integers, the number is padded, not the base.

If any of the **-i**, **-L**, **-l**, **-R**, **-U**, **-u** or **-Z** options are changed, all others from this set are cleared, unless they are also given on the same command line.

ulimit [**-aBCcdefHilMmnOPpqrSsTtVvwx**] [*value*]
(regular) 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 or the word "unlimited". The limits affect the shell and any processes created by the shell after a limit is imposed. Note that systems may not allow some limits to be increased once they are set. Also note that the types of limits available are system dependent—some systems have only the **-f** limit, or not even that, or can set only the soft limits, etc.

-a Display all limits (soft limits unless **-H** is used).

-B *n* Set the socket buffer size to *n* kibibytes.

-C *n* Set the number of cached threads to *n*.

-c *n* Impose a size limit of *n* blocks on the size of core dumps. Silently ignored if the system does not support this limit.

-d *n* Limit the size of the data area to *n* kibibytes.
On some systems, read-only maximum brk(2) size minus `etext`.

-e *n* Set the maximum niceness to *n*.

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

-H Set the hard limit only (the default is to set both hard and soft limits). With **-a**, display all hard limits.

-i *n* Set the number of pending signals to *n*.

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

-M *n* Set the AIO locked memory to *n* kibibytes.

-m *n* Impose a limit of *n* kibibytes on the amount of physical memory used.

-n *n* Impose a limit of *n* file descriptors that can be open at once. On some systems attempts to set are silently ignored.

-O *n* Set the number of AIO operations to *n*.

-P *n* Limit the number of threads per process to *n*.

This option mostly matches AT&T UNIX **ksh93**'s **-T**;
on AIX, see **-r** as used by its **ksh** though.

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

-q *n* Limit the size of POSIX message queues to *n* bytes.

-r *n* (**AIX**) Limit the number of threads per process to *n*.
(**Linux**) Set the maximum real-time priority to *n*.

-S Set the soft limit only (the default is to set both hard and soft limits). With **-a**, display soft limits (default).

-s *n* Limit the size of the stack area to *n* kibibytes.

-T *n* Impose a time limit of *n* real seconds ("humantime") to be used by each process.

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

-V *n* Set the number of vnode monitors on Haiku to *n*.

-v *n* Impose a limit of *n* kibibytes on the amount of virtual memory (address space) used.

-w *n* Limit the amount of swap space used to at most *n* kibibytes.

-x *n* Set the maximum number of file locks to *n*.

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

umask [**-S**] [*mask*]
(regular) 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 ...*]
(regular) 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 ...*
(keeps assignments, special) Unset the named parameters (**-v**, the default) or functions (**-f**). With *parameter*[*], attributes are retained, only values are unset. The exit status is non-zero if any of the parameters are read-only, zero otherwise (not portable).

wait [*job ...*]
(regular) 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 signal number (see **kill -l** *exit-status* above); if the last specified job cannot be found (because it never existed or had already finished), the exit status 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 ...*]
(regular) Without the **-v** option, it is the same as **command -v**, except aliases are printed as their definition only. With the **-v** option, it is exactly identical to **command -V**. In either case, with the **-p** option the search is restricted to the (current) PATH.

which [**-a**] [*name ...*]
(dot.mkshrc function) Without **-a**, behaves like **whence -p** (does a PATH search for each *name* printing the resulting pathname if found); with **-a**, matches in all PATH components are printed, i.e. the search is not stopped after a match. If no *name* was matched, the exit status is 2; if every name was matched, it is zero, otherwise it is 1. No diagnostics are produced on failure to match.

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 commands **fg** and **bg**.

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 which 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.

Terminal state
The state of the controlling terminal can be modified by a command executed in the foreground, whether or not job control is enabled, but the modified terminal state is only kept past the job's lifetime and used for later command invocations if the command exits successfully (i.e. with an exit status of 0). When such a job is momentarily stopped or restarted, the terminal state is saved and restored, respectively, but it will not be kept afterwards. In interactive mode, when line editing is enabled, the terminal state is saved before being reconfigured by the shell for the line editor, then restored before running a command.

POSIX mode
Entering **set -o posix** mode will cause **mksh** to behave even more POSIX compliant in places where the defaults or opinions differ. Note that **mksh** will still operate with unsigned 32-bit arithmetic; use **lksh** if arithmetic on the host `long` data type, complete with ISO C Undefined Behaviour, is required; refer to the `lksh`(1) manual page for details. Most other historic, AT&T UNIX **ksh**-compatible or opinionated differences can be disabled by using this mode; these are:

- The incompatible GNU **bash** I/O redirection **&>***file* is not supported.
- File descriptors created by I/O redirections are inherited by child processes.
- Numbers with a leading digit zero are interpreted as octal.
- The **echo** builtin does not interpret backslashes and only supports the exact option **-n**.
- Alias expansion with a trailing space only reruns on command words.
- Tilde expansion follows POSIX instead of Korn shell rules.
- The exit status of **fg** is always 0.
- **kill -l** only lists signal names, all in one line.
- **getopts** does not accept options with a leading '+'.
- **exec** skips builtins, functions and other commands and uses a `PATH` search to determine the utility to execute.

SH mode
Compatibility mode; intended for use with legacy scripts that cannot easily be fixed; the changes are as follows:

- The incompatible GNU **bash** I/O redirection **&>***file* is not supported.
- File descriptors created by I/O redirections are inherited by child processes.
- The **echo** builtin does not interpret backslashes and only supports the exact option **-n**, unless built with `-DMKSH_MIDNIGHTBSD01ASH_COMPAT`.
- The substitution operations ${*x*#*pat*}, ${*x*##*pat*}, ${*x*%*pat*}, and ${*x*%%*pat*} wrongly do not require a parenthesis to be escaped and do not parse extglobs.
- The getopt construct from `lksh`(1) passes through the errorlevel.
- **sh -c** eats a leading **--** if built with `-DMKSH_MIDNIGHTBSD01ASH_COMPAT`.

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. 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 allows 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.

Completed lines are pushed into the history, unless they begin with an IFS octet or IFS white space or are the same as the previous line.

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 ^[. These control 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 user's tty(4) characters (e.g. ERASE) are bound to reasonable substitutes and override the default bindings; their customary values are shown in parentheses below. The default bindings were chosen to resemble corresponding Emacs key bindings:

abort: INTR (^C), ^G
: Abort the current command, save it to the history, empty the line buffer and set the exit state to interrupted.

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

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

backward-word: [*n*] ^[b, ANSI-Ctrl-CurLeft, ANSI-Alt-CurLeft
: 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, ANSI-Home, PC-Home
: Moves the cursor to the beginning of the edited input line.

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

clear-screen: ^[^L
: Prints a compile-time configurable sequence to clear the screen and home the cursor, redraws the last line of the prompt string and the currently edited input line. The default sequence works for almost all standard terminals.

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).

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. Note that ^I is usually generated by the Tab (tabulator) key.

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

delete-char-forward: [*n*] ANSI-Del, PC-Del
: Deletes *n* characters after the cursor.

delete-word-backward: [*n*] Pfx1+ERASE (^[^H) , WERASE (^W) , ^[^?, ^[^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, ANSI-CurDown, PC-CurDown
: 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**, **search-history-up** or **up-history** has been performed.

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

edit-line: [*n*] ^Xe
: Internally run the command **fc -e "${VISUAL:-${EDITOR:-vi}}" --** *n*
on a temporary script file to interactively edit line *n* (if *n* is not specified, the current line); then, unless the editor invoked exits nonzero but even if the script was not changed, execute the resulting script as if typed on the command line; both the edited (resulting) and original lines are added onto history.

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

end-of-line: ^E, ANSI-End, PC-End
: 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 canonicalisation.

eot-or-delete: [*n*] EOF (^D)
If alone on a line, same as **eot**, otherwise, **delete-char-forward**.

error: (not bound)
Error (ring the bell).

evaluate-region: ^[^E
Evaluates the text between the mark and the cursor position (the entire line if no mark is set) as function substitution (if it cannot be parsed, the editing state is unchanged and the bell is rung to signal an error); $? is updated accordingly.

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, ANSI-CurRight, PC-CurRight
Moves the cursor forward *n* characters.

forward-word: [*n*] ^[f, ANSI-Ctrl-CurRight, ANSI-Alt-CurRight
Moves the cursor forward to the end of the *n*th word.

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

kill-line: KILL (^U)
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**, **search-history** or **search-history-up**.

no-op: QUIT (^\)
This does nothing.

prefix-1: ^[
Introduces a 2-character command sequence.

prefix-2: ^X, ^[[, ^[O
Introduces a multi-character command sequence.

prev-hist-word: [*n*] ^[., ^[_
The last word or, if given, the *n*th word (zero-based) of the previous (on repeated execution, second-last, third-last, etc.) command is inserted at the cursor. Use of this editing command trashes the mark.

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

quote-region: ^[Q
Escapes the text between the mark and the cursor position (the entire line if no mark is set) into a shell command argument.

redraw: ^L
Reprints the last line of the prompt string and the current input line on a new 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 escape key will leave search mode. Other commands, including sequences of escape as **prefix-1** followed by a **prefix-1** or **prefix-2** key will be executed after leaving search mode. The **abort** (^G) command will restore the input line before search started. 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.

search-history-up: ANSI-PgUp, PC-PgUp
Search backwards through the history buffer for commands whose beginning match the portion of the input line before the cursor. When used on an empty line, this has the same effect as **up-history**.

search-history-down: ANSI-PgDn, PC-PgDn
Search forwards through the history buffer for commands whose beginning match the portion of the input line before the cursor. When used on an empty line, this has the same effect as **down-history**. This is only useful after an **up-history**, **search-history** or **search-history-up**.

set-mark-command: ^[<space>
Set the mark at the cursor position.

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, ANSI-CurUp, PC-CurUp
: Scrolls the history buffer backward *n* lines (earlier).

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

version: ^[^V
: Display the version of **mksh**. The current edit buffer is restored as soon as a key is pressed. The restoring keypress is processed, unless it is a space.

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.

The tab completion escapes characters the same way as the following code:

```
print -nr -- "${x@/[\"-\$\&-*:-?[\\\`\{-\}${IFS-$' \t\n'}]/\\$KSH_MATCH}"
```

Vi editing mode

Note: The vi command-line editing mode has not yet been brought up to the same quality and feature set as the emacs mode. It is 8-bit clean but specifically does not support UTF-8 or MBCS.

The vi command-line editor in **mksh** 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 **mksh**, 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 ex(1)-style colon (**:**) commands).

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 **mksh** 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 Internally run the command **fc -e "${VISUAL:-${EDITOR:-vi}}" --** *n* on a temporary script file to interactively edit line *n* (if *n* is not specified, the current line); then, unless the editor invoked exits nonzero but even if the script was not changed, execute the resulting script as if typed on the command line; both the edited (resulting) and original lines are added onto history.

* 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 recognised if the **vi-tabcomplete** option is set, while <Esc> is only recognised 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.

^V Display the version of **mksh**. The current edit buffer is restored as soon as a key is pressed. The restoring keypress is ignored.

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

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.

ANSI-CurUp, *PC-PgUp*
Take the characters from the beginning of the line to the current cursor position as search string and do a history search, backwards, for lines beginning with this string; keep the cursor position. This works only in insert mode and keeps it enabled.

ANSI-CurDown, *PC-PgDn*
Take the characters from the beginning of the line to the current cursor position as search string and do a history search, forwards, for lines beginning with this string; keep the cursor position. This works only in insert mode and keeps it enabled.

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.

PC Home, End, Del and cursor keys
They move as expected, both in insert and command mode.

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

FILES

`~/.mkshrc`	User mkshrc profile (non-privileged interactive shells); see Startup files. The location can be changed at compile time (e.g. for embedded systems); AOSP Android builds use `/system/etc/mkshrc`.
`~/.profile`	User profile (non-privileged login shells); see Startup files near the top of this manual.
`/etc/profile`	System profile (login shells); see Startup files.
`/etc/shells`	Shell database.
`/etc/suid_profile`	Privileged shells' profile (sugid); see Startup files.

Note: On Android, `/system/etc/` contains the system and suid profile.

SEE ALSO

awk(1), cat(1), ed(1), getopt(1), lksh(1), sed(1), sh(1), stty(1), dup(2), execve(2), getgid(2), getuid(2), mknod(2), mkfifo(2), open(2), pipe(2), rename(2), wait(2), getopt(3), nl_langinfo(3), setlocale(3), signal(3), system(3), tty(4), shells(5), environ(7), script(7), utf-8(7), mknod(8)

The FAQ at `http://www.mirbsd.org/mksh-faq.htm` or in the `mksh.faq` file.

`http://www.mirbsd.org/ksh-chan.htm`

Morris Bolsky, *The KornShell Command and Programming Language*, *Prentice Hall PTR*, xvi + 356 pages, 1989, ISBN 978-0-13-516972-8 (0-13-516972-0).

Morris I. Bolsky and David G. Korn, *The New KornShell Command and Programming Language (2nd Edition)*, *Prentice Hall PTR*, xvi + 400 pages, 1995, ISBN 978-0-13-182700-4 (0-13-182700-6).

Stephen G. Kochan and Patrick H. Wood, *UNIX Shell Programming*, *Sams*, 3rd Edition, xiii + 437 pages, 2003, ISBN 978-0-672-32490-1 (0-672-32490-3).

IEEE Inc., *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)*, *IEEE Press*, Part 2: Shell and Utilities, xvii + 1195 pages, 1993, ISBN 978-1-55937-255-8 (1-55937-255-9).

Bill Rosenblatt, *Learning the Korn Shell*, *O'Reilly*, 360 pages, 1993, ISBN 978-1-56592-054-5 (1-56592-054-6).

Bill Rosenblatt and Arnold Robbins, *Learning the Korn Shell, Second Edition*, *O'Reilly*, 432 pages, 2002, ISBN 978-0-596-00195-7 (0-596-00195-9).

Barry Rosenberg, *KornShell Programming Tutorial*, *Addison-Wesley Professional*, xxi + 324 pages, 1991, ISBN 978-0-201-56324-5 (0-201-56324-X).

AUTHORS

The MirBSD Korn Shell is developed by mirabilos <m@mirbsd.org> as part of The MirOS Project. This shell is based on the public domain 7th edition Bourne shell clone by Charles Forsyth, who kindly agreed to, in countries where the Public Domain status of the work may not be valid, grant a copyright licence to the general public to deal in the work without restriction and permission to sublicence derivatives under the terms of any (OSI approved) Open Source licence, 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, Simon J. Gerraty and Michael Rendell. The effort of several projects, such as Debian and OpenBSD, and other contributors including our users, to improve the shell is appreciated. See the documentation, website and source code (CVS) for details.

mksh-os2 is developed by KO Myung-Hun <komh@chollian.net>.

mksh-w32 is developed by Michael Langguth <lan@scalaris.com>.

mksh/z/OS is contributed by Daniel Richard G. <skunk@iSKUNK.ORG>.

The BSD daemon is Copyright © Marshall Kirk McKusick. The complete legalese is at: http://www.mirbsd.org/TaC-mksh.txt

CAVEATS

mksh provides a consistent, clear interface normally. This may deviate from POSIX in historic or opinionated places. **set -o posix** (see POSIX mode for details) will make the shell more conformant, but mind the FAQ (see SEE ALSO), especially regarding locales. **mksh** (but not **lksh**) provides a consistent 32-bit integer arithmetic implementation, both signed and unsigned, with sign of the result of a remainder operation and wraparound defined, even (defying POSIX) on 36-bit and 64-bit systems.

mksh currently uses OPTU-16 internally, which is the same as UTF-8 and CESU-8 with 0000..FFFD being valid codepoints; raw octets are mapped into the PUA range EF80..EFFF, which is assigned by CSUR for this purpose.

BUGS

Suspending (using ^Z) pipelines like the one below will only suspend the currently running part of the pipeline; in this example, "fubar" is immediately printed on suspension (but not later after an **fg**).

```
$ /bin/sleep 666 && echo fubar
```

The truncation process involved when changing HISTFILE does not free old history entries (leaks memory) and leaks old entries into the new history if their line numbers are not overwritten by same-number entries from the persistent history file; truncating the on-disc file to HISTSIZE lines has always been broken and prone to history file corruption when multiple shells are accessing the file; the rollover process for the in-memory portion of the history is slow, should use memmove(3).

This document attempts to describe **mksh R59c** and up, compiled without any options impacting functionality, such as MKSH_SMALL, when not called as /bin/sh which, on some systems only, enables **set -o posix** or **set -o sh** automatically (whose behaviour differs across targets), for an operating environment supporting all of its advanced needs.

Please report bugs in **mksh** to the public development mailing list at <miros-mksh@mirbsd.org> (please note the EU-DSGVO/GDPR notice on http://www.mirbsd.org/rss.htm#lists and in the SMTP banner!) or in the #!/bin/mksh (or #ksh) IRC channel at irc.freenode.net (Port 6697 SSL, 6667 unencrypted), or at: https://launchpad.net/mksh