NAME

sh - command interpreter (shell)

SYNOPSIS

sh [**-/+abCEefhIimnPpTuVvx**] [**-/+o** *longname*] [*script* [*arg* ...]]
sh [**-/+abCEefhIimnPpTuVvx**] [**-/+o** *longname*] **-c** *string* [*name* [*arg* ...]]
sh [**-/+abCEefhIimnPpTuVvx**] [**-/+o** *longname*] **-s** [*arg* ...]

DESCRIPTION

The **sh** utility is the standard command interpreter for the system. The current version of **sh** is close to the IEEE Std 1003.1 ("POSIX.1") specification for the shell. It only supports features designated by POSIX, plus a few Berkeley extensions. This man page is not intended to be a tutorial nor a complete specification of the shell.

Overview

The shell is a command that reads lines from either a file or the terminal, interprets them, and generally executes other commands. It is the program that is started when a user logs into the system, although a user can select a different shell with the chsh(1) command. The shell implements a language that has flow control constructs, a macro facility that provides a variety of features in addition to data storage, along with built-in history and line editing capabilities. It incorporates many features to aid interactive use and has the advantage that the interpretative language is common to both interactive and non-interactive use (shell scripts). That is, commands can be typed directly to the running shell or can be put into a file, which can be executed directly by the shell.

Invocation

If no arguments are present and if the standard input of the shell is connected to a terminal (or if the **-i** option is set), the shell is considered an interactive shell. An interactive shell generally prompts before each command and handles programming and command errors differently (as described below). When first starting, the shell inspects argument 0, and if it begins with a dash (‘-’), the shell is also considered a login shell. This is normally done automatically by the system when the user first logs in. A login shell first reads commands from the files */etc/profile* and then *.profile* in a user’s home directory, if they exist. If the environment variable ENV is set on entry to a shell, or is set in the *.profile* of a login shell, the shell then subjects its value to parameter expansion and arithmetic expansion and reads commands from the named file. Therefore, a user should place commands that are to be executed only at login time in the *.profile* file, and commands that are executed for every shell inside the ENV file. The user can set the ENV variable to some file by placing the following line in the file *.profile* in the home directory, substituting for *.shrc* the filename desired:

```
ENV=$HOME/.shrc; export ENV
```

The first non-option argument specified on the command line will be treated as the name of a file from which to read commands (a shell script), and the remaining arguments are set as the positional parameters of the shell ($1, $2, etc.). Otherwise, the shell reads commands from its standard input.

Unlike older versions of **sh** the ENV script is only sourced on invocation of interactive shells. This closes a well-known, and sometimes easily exploitable security hole related to poorly thought out ENV scripts.

Argument List Processing

All of the single letter options to **sh** have a corresponding long name, with the exception of **-c** and **-/+o**. These long names are provided next to the single letter options in the descriptions below. The long name for an option may be specified as an argument to the **-/+o** option of **sh**. Once the shell is running, the long name for an option may be specified as an argument to the **-/+o** option of the **set** built-in command (described later in the section called *Built-in Commands*). Introducing an option with a dash (‘-’) enables the option, while using a plus (‘+’) disables the option. A "--" or plain ‘-’ will stop option processing and will force the remaining words on the command line to be treated as arguments. The **-/+o** and **-c** options do not have long names. They take arguments and are described after the single letter options.

-a allexport

Flag variables for export when assignments are made to them.

-b notify

Enable asynchronous notification of background job completion. (UNIMPLEMENTED)

-C noclobber

Do not overwrite existing files with ‘>’.

-E emacs

Enable the built-in emacs(1) (*ports/editors/emacs*) command line editor (disables the **-V** option if it has been set; set automatically when interactive on terminals).

-e errexit

Exit immediately if any untested command fails in non-interactive mode. The exit status of a command is considered to be explicitly tested if the command is part of the list used to control an **if**, **elif**, **while**, or **until**; if the command is the left hand operand of an "&&" or "||" operator; or if the command is a pipeline preceded by the **!** keyword. If a shell function is executed and its exit status is explicitly tested, all commands of the function are considered to be tested as well.

It is recommended to check for failures explicitly instead of relying on **-e** because it tends to

behave in unexpected ways, particularly in larger scripts.

-f noglob
Disable pathname expansion.

-h trackall
A do-nothing option for POSIX compliance.

-I ignoreeof
Ignore EOF's from input when in interactive mode.

-i interactive
Force the shell to behave interactively.

-m monitor
Turn on job control (set automatically when interactive). A new process group is created for each pipeline (called a job). It is possible to suspend jobs or to have them run in the foreground or in the background. In a non-interactive shell, this option can be set even if no terminal is available and is useful to place processes in separate process groups.

-n noexec
If not interactive, read commands but do not execute them. This is useful for checking the syntax of shell scripts.

-P physical
Change the default for the **cd** and **pwd** commands from **-L** (logical directory layout) to **-P** (physical directory layout).

-p privileged
Turn on privileged mode. This mode is enabled on startup if either the effective user or group ID is not equal to the real user or group ID. Turning this mode off sets the effective user and group IDs to the real user and group IDs. When this mode is enabled for interactive shells, the file */etc/suid_profile* is sourced instead of *~/.profile* after */etc/profile* is sourced, and the contents of the ENV variable are ignored.

-s stdin
Read commands from standard input (set automatically if no file arguments are present). This option has no effect when set after the shell has already started running (i.e., when set with the **set** command).

-T trapsasync
When waiting for a child, execute traps immediately. If this option is not set, traps are executed after the child exits, as specified in IEEE Std 1003.2 ("POSIX.2"). This nonstandard option is useful for putting guarding shells around children that block signals. The surrounding shell may kill the child or it may just return control to the tty and leave the child alone, like this:

sh -T -c "trap 'exit 1' 2 ; some-blocking-program"

-u nounset
Write a message to standard error when attempting to expand a variable, a positional parameter or the special parameter *!* that is not set, and if the shell is not interactive, exit immediately.

-V vi Enable the built-in vi(1) command line editor (disables **-E** if it has been set).

-v verbose
The shell writes its input to standard error as it is read. Useful for debugging.

-x xtrace
Write each command (preceded by the value of the *PS4* variable subjected to parameter expansion and arithmetic expansion) to standard error before it is executed. Useful for debugging.

nolog Another do-nothing option for POSIX compliance. It only has a long name.

pipefail
Change the exit status of a pipeline to the last non-zero exit status of any command in the pipeline, if any. Since an exit due to SIGPIPE counts as a non-zero exit status, this option may cause non-zero exit status for successful pipelines if a command such as head(1) in the pipeline terminates with status 0 without reading its input completely. This option only has a long name.

verify Set O_VERIFY when sourcing files or loading profiles.

The **-c** option causes the commands to be read from the *string* operand instead of from the standard input. Keep in mind that this option only accepts a single string as its argument, hence multi-word strings must be quoted.

The **-/+o** option takes as its only argument the long name of an option to be enabled or disabled. For example, the following two invocations of **sh** both enable the built-in emacs(1) (*ports/editors/emacs*) command line editor:

```
set -E
set -o emacs
```

If used without an argument, the **-o** option displays the current option settings in a human-readable format. If **+o** is used without an argument, the current option settings are output in a format suitable for re-input into the shell.

Lexical Structure

The shell reads input in terms of lines from a file and breaks it up into words at whitespace (blanks and tabs), and at certain sequences of characters called "operators", which are special to the shell. There are two types of operators: control operators and redirection operators (their meaning is discussed later). The following is a list of valid operators:

Control operators:

```
&       &&      (       )       \n
;;      ;&      ;       |       ||
```

Redirection operators:

```
<       >       <<      >>      <>
<&      >&      <<-     >|
```

The character ‘#’ introduces a comment if used at the beginning of a word. The word starting with ‘#’ and the rest of the line are ignored.

ASCII NUL characters (character code 0) are not allowed in shell input.

Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell, such as operators, whitespace, keywords, or alias names.

There are four types of quoting: matched single quotes, dollar-single quotes, matched double quotes, and backslash.

Single Quotes

Enclosing characters in single quotes preserves the literal meaning of all the characters (except single quotes, making it impossible to put single-quotes in a single-quoted string).

Dollar-Single Quotes

Enclosing characters between $’ and ’ preserves the literal meaning of all characters except backslashes and single quotes. A backslash introduces a C-style escape sequence:

\a	Alert (ring the terminal bell)
\b	Backspace
\c*c*	The control character denoted by ^*c* in stty(1). If *c* is a backslash, it must be doubled.
\e	The ESC character (ASCII 0x1b)
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Literal backslash
\’	Literal single-quote
\"	Literal double-quote
nnn	The byte whose octal value is *nnn* (one to three digits)
\x*nn*	The byte whose hexadecimal value is *nn* (one or more digits only the last two of which are used)
\u*nnnn*	The Unicode code point *nnnn* (four hexadecimal digits)
\U*nnnnnnnn*	The Unicode code point *nnnnnnnn* (eight hexadecimal digits)

The sequences for Unicode code points are currently only useful with UTF-8 locales. They reject code point 0 and UTF-16 surrogates.

If an escape sequence would produce a byte with value 0, that byte and the rest of the string until the matching single-quote are ignored.

Any other string starting with a backslash is an error.

Double Quotes

Enclosing characters within double quotes preserves the literal meaning of all characters except dollar sign (‘$’), backquote (‘‘’), and backslash (‘\’). The backslash inside double quotes is historically weird. It remains literal unless it precedes the following characters, which it serves to quote:

$ ‘ " \ \n

Backslash

A backslash preserves the literal meaning of the following character, with the exception of the newline character (‘\n’). A backslash preceding a newline is treated as a line continuation.

Keywords

Keywords or reserved words are words that have special meaning to the shell and are recognized at the beginning of a line and after a control operator. The following are keywords:

!	{	}	**case**	**do**
done	**elif**	**else**	**esac**	**fi**
for	**if**	**then**	**until**	**while**

Aliases

An alias is a name and corresponding value set using the **alias** built-in command. Wherever the command word of a simple command may occur, and after checking for keywords if a keyword may occur, the shell checks the word to see if it matches an alias. If it does, it replaces it in the input stream with its value. For example, if there is an alias called "lf" with the value "ls -F", then the input

lf foobar

would become

ls -F foobar

Aliases are also recognized after an alias whose value ends with a space or tab. For example, if there is also an alias called "nohup" with the value "nohup ", then the input

nohup lf foobar

would become

```
nohup ls -F foobar
```

Aliases provide a convenient way for naive users to create shorthands for commands without having to learn how to create functions with arguments. Using aliases in scripts is discouraged because the command that defines them must be executed before the code that uses them is parsed. This is fragile and not portable.

An alias name may be escaped in a command line, so that it is not replaced by its alias value, by using quoting characters within or adjacent to the alias name. This is most often done by prefixing an alias name with a backslash to execute a function, built-in, or normal program with the same name. See the *Quoting* subsection.

Commands

The shell interprets the words it reads according to a language, the specification of which is outside the scope of this man page (refer to the BNF in the IEEE Std 1003.2 ("POSIX.2") document). Essentially though, a line is read and if the first word of the line (or after a control operator) is not a keyword, then the shell has recognized a simple command. Otherwise, a complex command or some other special construct may have been recognized.

Simple Commands

If a simple command has been recognized, the shell performs the following actions:

1. Leading words of the form "name=value" are stripped off and assigned to the environment of the simple command (they do not affect expansions). Redirection operators and their arguments (as described below) are stripped off and saved for processing.

2. The remaining words are expanded as described in the section called *Word Expansions*, and the first remaining word is considered the command name and the command is located. The remaining words are considered the arguments of the command. If no command name resulted, then the "name=value" variable assignments recognized in 1) affect the current shell.

3. Redirections are performed as described in the next section.

Redirections

Redirections are used to change where a command reads its input or sends its output. In general, redirections open, close, or duplicate an existing reference to a file. The overall format used for redirection is:

[*n*] *redir-op file*

The *redir-op* is one of the redirection operators mentioned previously. The following gives some examples of how these operators can be used. Note that stdin and stdout are commonly used abbreviations for standard input and standard output respectively.

[*n*]> *file*	redirect stdout (or file descriptor *n*) to *file*
[*n*]>| *file*	same as above, but override the **-C** option
[*n*]>> *file*	append stdout (or file descriptor *n*) to *file*
[*n*]< *file*	redirect stdin (or file descriptor *n*) from *file*
[*n*]<> *file*	redirect stdin (or file descriptor *n*) to and from *file*
[*n1*]<&*n2*	duplicate stdin (or file descriptor *n1*) from file descriptor *n2*
[*n*]<&-	close stdin (or file descriptor *n*)
[*n1*]>&*n2*	duplicate stdout (or file descriptor *n1*) to file descriptor *n2*
[*n*]>&-	close stdout (or file descriptor *n*)

The following redirection is often called a "here-document".

[*n*]<< *delimiter*
here-doc-text
...
delimiter

All the text on successive lines up to the delimiter is saved away and made available to the command on standard input, or file descriptor *n* if it is specified. If the *delimiter* as specified on the initial line is quoted, then the *here-doc-text* is treated literally, otherwise the text is subjected to parameter expansion, command substitution, and arithmetic expansion (as described in the section on *Word Expansions*). If the operator is "<<-" instead of "<<", then leading tabs in the *here-doc-text* are stripped.

Search and Execution

There are three types of commands: shell functions, built-in commands, and normal programs. The command is searched for (by name) in that order. The three types of commands are all executed in a different way.

When a shell function is executed, all of the shell positional parameters (except $0, which remains unchanged) are set to the arguments of the shell function. The variables which are explicitly placed in the environment of the command (by placing assignments to them before the function name) are made local to the function and are set to the values given. Then the command given in the function definition is executed. The positional parameters are restored to their original values when the command completes. This all occurs within the current shell.

Shell built-in commands are executed internally to the shell, without spawning a new process. There are two kinds of built-in commands: regular and special. Assignments before special builtins persist after they finish executing and assignment errors, redirection errors and certain operand errors cause a script to be aborted. Special builtins cannot be overridden with a function. Both regular and special builtins can affect the shell in ways normal programs cannot.

Otherwise, if the command name does not match a function or built-in command, the command is searched for as a normal program in the file system (as described in the next section). When a normal program is executed, the shell runs the program, passing the arguments and the environment to the program. If the program is not a normal executable file (i.e., if it does not begin with the "magic number" whose ASCII representation is "#!", resulting in an ENOEXEC return value from execve(2)) but appears to be a text file, the shell will run a new instance of **sh** to interpret it.

Note that previous versions of this document and the source code itself misleadingly and sporadically refer to a shell script without a magic number as a "shell procedure".

Path Search

When locating a command, the shell first looks to see if it has a shell function by that name. Then it looks for a built-in command by that name. If a built-in command is not found, one of two things happen:

1. Command names containing a slash are simply executed without performing any searches.

2. The shell searches each entry in the *PATH* variable in turn for the command. The value of the *PATH* variable should be a series of entries separated by colons. Each entry consists of a directory name. The current directory may be indicated implicitly by an empty directory name, or explicitly by a single period.

Command Exit Status

Each command has an exit status that can influence the behavior of other shell commands. The paradigm is that a command exits with zero for normal or success, and non-zero for failure, error, or a false indication. The man page for each command should indicate the various exit codes and what they mean. Additionally, the built-in commands return exit codes, as does an executed shell function.

If a command is terminated by a signal, its exit status is greater than 128. The signal name can be found by passing the exit status to kill -l.

If there is no command word, the exit status is the exit status of the last command substitution executed, or zero if the command does not contain any command substitutions.

Complex Commands

Complex commands are combinations of simple commands with control operators or keywords, together creating a larger complex command. More generally, a command is one of the following:

simple command

pipeline

list or compound-list

compound command

function definition

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command, or zero if no simple command was executed.

Pipelines

A pipeline is a sequence of one or more commands separated by the control operator '|'. The standard output of all but the last command is connected to the standard input of the next command. The standard output of the last command is inherited from the shell, as usual.

The format for a pipeline is:

[!] *command1* [| *command2* ...]

The standard output of *command1* is connected to the standard input of *command2*. The standard input, standard output, or both of a command is considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command.

Note that unlike some other shells, **sh** executes each process in a pipeline with more than one command in a subshell environment and as a child of the **sh** process.

If the pipeline is not in the background (discussed later), the shell waits for all commands to complete.

If the keyword **!** does not precede the pipeline, the exit status is the exit status of the last command specified in the pipeline if the **pipefail** option is not set or all commands returned zero, or the last non-zero exit status of any command in the pipeline otherwise. Otherwise, the exit status is the logical NOT of that exit status. That is, if that status is zero, the exit status is 1; if that status is greater than zero, the exit status is zero.

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
command1 2>&1 | command2
```

sends both the standard output and standard error of *command1* to the standard input of *command2*.

A ‘;’ or newline terminator causes the preceding AND-OR-list (described below in the section called *Short-Circuit List Operators*) to be executed sequentially; an ‘&’ causes asynchronous execution of the preceding AND-OR-list.

Background Commands (&)

If a command is terminated by the control operator ampersand (‘&’), the shell executes the command in a subshell environment (see *Grouping Commands Together* below) and asynchronously; the shell does not wait for the command to finish before executing the next command.

The format for running a command in background is:

command1 & [*command2* & ...]

If the shell is not interactive, the standard input of an asynchronous command is set to */dev/null*.

The exit status is zero.

Lists (Generally Speaking)

A list is a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters. The commands in a list are executed in the order they are written. If command is followed by an ampersand, the shell starts the command and immediately proceeds onto the next command; otherwise it waits for the command to terminate before proceeding to the next one.

Short-Circuit List Operators

"&&" and "||" are AND-OR list operators. "&&" executes the first command, and then executes the second command if the exit status of the first command is zero. "||" is similar, but executes the second

command if the exit status of the first command is nonzero. "&&" and "||" both have the same priority.

Flow-Control Constructs (if, while, for, case)

The syntax of the **if** command is:

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

The exit status is that of selected **then** or **else** list, or zero if no list was selected.

The syntax of the **while** command is:

while *list*
do *list*
done

The two lists are executed repeatedly while the exit status of the first list is zero. The **until** command is similar, but has the word **until** in place of **while**, which causes it to repeat until the exit status of the first list is zero.

The exit status is that of the last execution of the second list, or zero if it was never executed.

The syntax of the **for** command is:

for *variable* [**in** *word* ...]
do *list*
done

If **in** and the following words are omitted, **in** "$@" is used instead. The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn. The **do** and **done** commands may be replaced with ‘{’ and ‘}’.

The syntax of the **break** and **continue** commands is:

break [*num*]
continue [*num*]

The **break** command terminates the *num* innermost **for** or **while** loops. The **continue** command continues with the next iteration of the innermost loop. These are implemented as special built-in commands.

The syntax of the **case** command is:

case *word* **in**
pattern) *list* ;;
...
esac

The pattern can actually be one or more patterns (see *Shell Patterns* described later), separated by '|' characters. Tilde expansion, parameter expansion, command substitution, arithmetic expansion and quote removal are applied to the word. Then, each pattern is expanded in turn using tilde expansion, parameter expansion, command substitution and arithmetic expansion and the expanded form of the word is checked against it. If a match is found, the corresponding list is executed. If the selected list is terminated by the control operator ';&' instead of ';;', execution continues with the next list, continuing until a list terminated with ';;' or the end of the **case** command.

Grouping Commands Together

Commands may be grouped by writing either

(*list*)

or

{ *list*; }

The first form executes the commands in a subshell environment. A subshell environment has its own copy of:

1. The current working directory as set by **cd**.
2. The file creation mask as set by **umask**.
3. Resource limits as set by **ulimit**.
4. References to open files.
5. Traps as set by **trap**.
6. Known jobs.
7. Positional parameters and variables.

8. Shell options.

9. Shell functions.

10. Shell aliases.

These are copied from the parent shell environment, except that trapped (but not ignored) signals are reset to the default action and known jobs are cleared. Any changes do not affect the parent shell environment.

A subshell environment may be implemented as a child process or differently. If job control is enabled in an interactive shell, commands grouped in parentheses can be suspended and continued as a unit.

For compatibility with other shells, two open parentheses in sequence should be separated by whitespace.

The second form never forks another shell, so it is slightly more efficient. Grouping commands together this way allows the user to redirect their output as though they were one program:

```
{ echo -n "hello"; echo " world"; } > greeting
```

Functions

The syntax of a function definition is

name ( ) *command*

A function definition is an executable statement; when executed it installs a function named *name* and returns an exit status of zero. The *command* is normally a list enclosed between ‘{’ and ‘}’.

Variables may be declared to be local to a function by using the **local** command. This should appear as the first statement of a function, and the syntax is:

local [*variable* ...] [-]

The **local** command is implemented as a built-in command. The exit status is zero unless the command is not in a function or a variable name is invalid.

When a variable is made local, it inherits the initial value and exported and readonly flags from the variable with the same name in the surrounding scope, if there is one. Otherwise, the variable is initially unset. The shell uses dynamic scoping, so that if the variable *x* is made local to function *f*, which then

calls function *g*, references to the variable *x* made inside *g* will refer to the variable *x* declared inside *f*, not to the global variable named *x*.

The only special parameter that can be made local is '-'. Making '-' local causes any shell options (including those that only have long names) that are changed via the **set** command inside the function to be restored to their original values when the function returns.

The syntax of the **return** command is

return [*exitstatus*]

It terminates the current executional scope, returning from the closest nested function or sourced script; if no function or sourced script is being executed, it exits the shell instance. The **return** command is implemented as a special built-in command.

Variables and Parameters

The shell maintains a set of parameters. A parameter denoted by a name (consisting solely of alphabetics, numerics, and underscores, and starting with an alphabetic or an underscore) is called a variable. When starting up, the shell turns all environment variables with valid names into shell variables. New variables can be set using the form

name=value

A parameter can also be denoted by a number or a special character as explained below.

Assignments are expanded differently from other words: tilde expansion is also performed after the equals sign and after any colon and usernames are also terminated by colons, and field splitting and pathname expansion are not performed.

This special expansion applies not only to assignments that form a simple command by themselves or precede a command word, but also to words passed to the **export**, **local** or **readonly** built-in commands that have this form. For this, the builtin's name must be literal (not the result of an expansion) and may optionally be preceded by one or more literal instances of **command** without options.

Positional Parameters

A positional parameter is a parameter denoted by a number greater than zero. The shell sets these initially to the values of its command line arguments that follow the name of the shell script. The **set** built-in command can also be used to set or reset them.

Special Parameters

Special parameters are parameters denoted by a single special character or the digit zero. They are shown in the following list, exactly as they would appear in input typed by the user or in the source of a shell script.

$* Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string it expands to a single field with the value of each parameter separated by the first character of the *IFS* variable, or by a space if *IFS* is unset.

$@ Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, each positional parameter expands as a separate argument. If there are no positional parameters, the expansion of @ generates zero arguments, even when @ is double-quoted. What this basically means, for example, is if $1 is "abc" and $2 is "def ghi", then "$@" expands to the two arguments:

"abc" "def ghi"

$# Expands to the number of positional parameters.

$? Expands to the exit status of the most recent pipeline.

$- (hyphen) Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation, by the **set** built-in command, or implicitly by the shell.

$$ Expands to the process ID of the invoked shell. A subshell retains the same value of *$* as its parent.

$! Expands to the process ID of the most recent background command executed from the current shell. For a pipeline, the process ID is that of the last command in the pipeline. If this parameter is referenced, the shell will remember the process ID and its exit status until the **wait** built-in command reports completion of the process.

$0 (zero) Expands to the name of the shell script if passed on the command line, the *name* operand if given (with **-c**) or otherwise argument 0 passed to the shell.

Special Variables

The following variables are set by the shell or have special meaning to it:

CDPATH The search path used with the **cd** built-in.

EDITOR The fallback editor used with the **fc** built-in. If not set, the default editor is ed(1).

FCEDIT The default editor used with the **fc** built-in.

HISTFILE File used for persistent history storage. If unset *~/.sh_history* will be used. If set but empty or *HISTSIZE* is set to 0 the shell will not load and save the history.

HISTSIZE The number of previous commands that are accessible.

HOME The user's home directory, used in tilde expansion and as a default directory for the **cd** built-in.

IFS Input Field Separators. This is initialized at startup to <space>, <tab>, and <newline> in that order. This value also applies if *IFS* is unset, but not if it is set to the empty string. See the *White Space Splitting* section for more details.

LINENO The current line number in the script or function.

MAIL The name of a mail file, that will be checked for the arrival of new mail. Overridden by *MAILPATH*.

MAILPATH
A colon (':') separated list of file names, for the shell to check for incoming mail. This variable overrides the *MAIL* setting. There is a maximum of 10 mailboxes that can be monitored at once.

OPTIND The index of the next argument to be processed by **getopts**. This is initialized to 1 at startup.

PATH The default search path for executables. See the *Path Search* section for details.

PPID The parent process ID of the invoked shell. This is set at startup unless this variable is in the environment. A later change of parent process ID is not reflected. A subshell retains the same value of *PPID*.

PS1 The primary prompt string, which defaults to "$ ", unless you are the superuser, in which case it defaults to "# ". *PS1* may include any of the following formatting sequences, which are replaced by the given information:

\H This system's fully-qualified hostname (FQDN).

\h This system's hostname.

\u User name.

\W The final component of the current working directory.

\w The entire path of the current working directory.

\$ Superuser status. "$" for normal users and "#" for superusers.

\\ A literal backslash.

\[Start of a sequence of non-printing characters (used, for example, to embed ANSI CSI sequences into the prompt).

\] End of a sequence of non-printing characters.

The following special and non-printing characters are supported within the sequence of non-printing characters:

\a Emits ASCII BEL (0x07, 007) character.

\e Emits ASCII ESC (0x1b, 033) character.

\r Emits ASCII CR (0x0d, 015) character.

\n Emits CRLF sequence.

PS2 The secondary prompt string, which defaults to "> ". *PS2* may include any of the formatting sequences from *PS1*.

PS4 The prefix for the trace output (if **-x** is active). The default is "+ ".

Word Expansions

This clause describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained later.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter @ within double-quotes, as was described above.

The order of word expansion is:

1. Tilde Expansion, Parameter Expansion, Command Substitution, Arithmetic Expansion (these all occur at the same time).

2. Field Splitting is performed on fields generated by step (1) unless the *IFS* variable is null.

3. Pathname Expansion (unless the **-f** option is in effect).

4. Quote Removal.

The '$' character is used to introduce parameter expansion, command substitution, or arithmetic expansion.

Tilde Expansion (substituting a user's home directory)

A word beginning with an unquoted tilde character ('~') is subjected to tilde expansion. All the characters up to a slash ('/') or the end of the word are treated as a username and are replaced with the user's home directory. If the username is missing (as in *~/foobar*), the tilde is replaced with the value of the *HOME* variable (the current user's home directory).

Parameter Expansion

The format for parameter expansion is as follows:

${*expression*}

where *expression* consists of all characters until the matching '}'. Any '}' escaped by a backslash or within a single-quoted or double-quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, are not examined in determining the matching '}'. If the variants with '+', '-', '=' or '?' occur within a double-quoted string, as an extension there may be unquoted parts (via double-quotes inside the expansion); '}' within such parts are also not examined in determining the matching '}'.

The simplest form for parameter expansion is:

${*parameter*}

The value, if any, of *parameter* is substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when parameter is followed by a character that could be

interpreted as part of the name. If a parameter expansion occurs inside double-quotes:

1. Field splitting is not performed on the results of the expansion, with the exception of the special parameter @.

2. Pathname expansion is not performed on the results of the expansion.

In addition, a parameter expansion can be modified by using one of the following formats.

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

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

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

${*parameter*:+*word*}
Use Alternate Value. If *parameter* is unset or null, null is substituted; otherwise, the expansion of *word* is substituted.

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

The *word* inherits the type of quoting (unquoted, double-quoted or here-document) from the surroundings, with the exception that a backslash that quotes a closing brace is removed during quote removal.

${#*parameter*}
String Length. The length in characters of the value of *parameter*.

The following four varieties of parameter expansion provide for substring processing. In each case,

pattern matching notation (see *Shell Patterns*), rather than regular expression notation, is used to evaluate the patterns. If parameter is one of the special parameters * or @, the result of the expansion is unspecified. Enclosing the full parameter expansion string in double-quotes does not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces has this effect.

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

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

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

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

Command Substitution

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

$(*command*)

or the backquoted version:

‘*command*‘

The shell expands the command substitution by executing command and replacing the command substitution with the standard output of the command, removing sequences of one or more newlines at the end of the substitution. Embedded newlines before the end of the output are not removed; however, during field splitting, they may be translated into spaces depending on the value of *IFS* and the quoting

that is in effect. The command is executed in a subshell environment, except that the built-in commands **jobid**, **jobs**, and **trap** return information about the parent shell environment and **times** returns information about the same process if they are the only command in a command substitution.

If a command substitution of the $(form begins with a subshell, the $(and (must be separated by whitespace to avoid ambiguity with arithmetic expansion.

Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion is as follows:

$((*expression*))

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

The allowed expressions are a subset of C expressions, summarized below.

Values All values are of type *intmax_t*.

Constants
Decimal, octal (starting with 0) and hexadecimal (starting with 0x) integer constants.

Variables Shell variables can be read and written and contain integer constants.

Unary operators
! ~ + -

Binary operators
* / % + - << >> < <= > >= == != & ^ | && ||

Assignment operators
= += -= *= /= %= <<= >>= &= ^= |=

Conditional operator
? :

The result of the expression is substituted in decimal.

White Space Splitting (Field Splitting)

In certain contexts, after parameter expansion, command substitution, and arithmetic expansion the shell scans the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

Characters in *IFS* that are whitespace (<space>, <tab>, and <newline>) are treated differently from other characters in *IFS*.

Whitespace in *IFS* at the beginning or end of a word is discarded.

Subsequently, a field is delimited by either

1. a non-whitespace character in *IFS* with any whitespace in *IFS* surrounding it, or

2. one or more whitespace characters in *IFS*.

If a word ends with a non-whitespace character in *IFS*, there is no empty field after this character.

If no field is delimited, the word is discarded. In particular, if a word consists solely of an unquoted substitution and the result of the substitution is null, it is removed by field splitting even if *IFS* is null.

Pathname Expansion (File Name Generation)

Unless the **-f** option is set, file name generation is performed after word splitting is complete. Each word is viewed as a series of patterns, separated by slashes. The process of expansion replaces the word with the names of all existing files whose names can be formed by replacing each pattern with a string that matches the specified pattern. There are two restrictions on this: first, a pattern cannot match a string containing a slash, and second, a pattern cannot match a string starting with a period unless the first character of the pattern is a period. The next section describes the patterns used for Pathname Expansion, the four varieties of parameter expansion for substring processing and the **case** command.

Shell Patterns

A pattern consists of normal characters, which match themselves, and meta-characters. The meta-characters are ‘*’, ‘?’, and ‘[’. These characters lose their special meanings if they are quoted. When command or variable substitution is performed and the dollar sign or back quotes are not double-quoted, the value of the variable or the output of the command is scanned for these characters and they are turned into meta-characters.

An asterisk (‘*’) matches any string of characters. A question mark (‘?’) matches any single character. A left bracket (‘[’) introduces a character class. The end of the character class is indicated by a ‘]’; if the ‘]’ is missing then the ‘[’ matches a ‘[’ rather than introducing a character class. A character class

matches any of the characters between the square brackets. A locale-dependent range of characters may be specified using a minus sign. A named class of characters (see wctype(3)) may be specified by surrounding the name with '[:' and ':]'. For example, '[[:alpha:]]' is a shell pattern that matches a single letter. The character class may be complemented by making an exclamation point ('!') the first character of the character class. A caret ('^') has the same effect but is non-standard.

To include a ']' in a character class, make it the first character listed (after the '!' or '^', if any). To include a '-', make it the first or last character listed.

Built-in Commands

This section lists the built-in commands.

: A null command that returns a 0 (true) exit value.

. *file* The commands in the specified file are read and executed by the shell. The **return** command may be used to return to the **.** command's caller. If *file* contains any '/' characters, it is used as is. Otherwise, the shell searches the *PATH* for the file. If it is not found in the *PATH*, it is sought in the current working directory.

[A built-in equivalent of test(1).

alias [*name*[=*string*] ...]
If *name=string* is specified, the shell defines the alias *name* with value *string*. If just *name* is specified, the value of the alias *name* is printed. With no arguments, the **alias** built-in command prints the names and values of all defined aliases (see **unalias**). Alias values are written with appropriate quoting so that they are suitable for re-input to the shell. Also see the *Aliases* subsection.

bg [*job* ...]
Continue the specified jobs (or the current job if no jobs are given) in the background.

bind [**-aeklrsv**] [*key* [*command*]]
List or alter key bindings for the line editor. This command is documented in editrc(5).

break [*num*]
See the *Flow-Control Constructs* subsection.

builtin *cmd* [*arg* ...]
Execute the specified built-in command, *cmd*. This is useful when the user wishes to override a shell function with the same name as a built-in command.

cd [**-L** | **-P**] [**-e**] [*directory*]

cd - Switch to the specified *directory*, to the directory specified in the *HOME* environment variable if no *directory* is specified or to the directory specified in the *OLDPWD* environment variable if *directory* is **-**. If *directory* does not begin with */*, *.*, or *..*, then the directories listed in the *CDPATH* variable will be searched for the specified *directory*. If *CDPATH* is unset, the current directory is searched. The format of *CDPATH* is the same as that of *PATH*. In an interactive shell, the **cd** command will print out the name of the directory that it actually switched to if the *CDPATH* mechanism was used or if *directory* was **-**.

If the **-P** option is specified, *..* is handled physically and symbolic links are resolved before *..* components are processed. If the **-L** option is specified, *..* is handled logically. This is the default.

The **-e** option causes **cd** to return exit status 1 if the full pathname of the new directory cannot be determined reliably or at all. Normally this is not considered an error, although a warning is printed.

If changing the directory fails, the exit status is greater than 1. If the directory is changed, the exit status is 0, or also 1 if **-e** was given.

chdir A synonym for the **cd** built-in command.

command [**-p**] [*utility* [*argument* ...]]

command [**-p**] **-v** *utility*

command [**-p**] **-V** *utility*

The first form of invocation executes the specified *utility*, ignoring shell functions in the search. If *utility* is a special builtin, it is executed as if it were a regular builtin.

If the **-p** option is specified, the command search is performed using a default value of *PATH* that is guaranteed to find all of the standard utilities.

If the **-v** option is specified, *utility* is not executed but a description of its interpretation by the shell is printed. For ordinary commands the output is the path name; for shell built-in commands, shell functions and keywords only the name is written. Aliases are printed as "**alias** *name=value*".

The **-V** option is identical to **-v** except for the output. It prints "*utility* **is** *description*" where

description is either the path name to *utility*, a special shell builtin, a shell builtin, a shell function, a shell keyword or an alias for *value*.

continue [*num*]

See the *Flow-Control Constructs* subsection.

echo [**-e** | **-n**] [*string* ...]

Print a space-separated list of the arguments to the standard output and append a newline character.

-n Suppress the output of the trailing newline.

-e Process C-style backslash escape sequences. The **echo** command understands the following character escapes:

\a Alert (ring the terminal bell)

\b Backspace

\c Suppress the trailing newline (this has the side-effect of truncating the line if it is not the last character)

\e The ESC character (ASCII 0x1b)

\f Formfeed

\n Newline

\r Carriage return

\t Horizontal tab

\v Vertical tab

\\ Literal backslash

\0nnn (Zero) The character whose octal value is *nnn*

If *string* is not enclosed in quotes then the backslash itself must be escaped with a backslash to protect it from the shell. For example

```
$ echo -e "a\vb"
a
 b
$ echo -e a\\vb
a
 b
$ echo -e "a\\b"
a\b
$ echo -e a\\\\b
a\b
```

Only one of the **-e** and **-n** options may be specified.

eval *string ...*

Concatenate all the arguments with spaces. Then re-parse and execute the command.

exec [*command* [arg ...]]

Unless *command* is omitted, the shell process is replaced with the specified program (which must be a real program, not a shell built-in command or function). Any redirections on the **exec** command are marked as permanent, so that they are not undone when the **exec** command finishes.

exit [*exitstatus*]

Terminate the shell process. If *exitstatus* is given it is used as the exit status of the shell. Otherwise, if the shell is executing an **EXIT** trap, the exit status of the last command before the trap is used; if the shell is executing a trap for a signal, the shell exits by resending the signal to itself. Otherwise, the exit status of the preceding command is used. The exit status should be an integer between 0 and 255.

export *name ...*

export [**-p**]

The specified names are exported so that they will appear in the environment of subsequent commands. The only way to un-export a variable is to **unset** it. The shell allows the value of a variable to be set at the same time as it is exported by writing

export *name=value*

With no arguments the **export** command lists the names of all exported variables. If the **-p** option is specified, the exported variables are printed as "**export** *name=value*" lines, suitable for re-input

to the shell.

false A null command that returns a non-zero (false) exit value.

fc [**-e** *editor*] [*first* [*last*]]

fc -l [**-nr**] [*first* [*last*]]

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

The **fc** built-in command lists, or edits and re-executes, commands previously entered to an interactive shell.

-e *editor*

Use the editor named by *editor* to edit the commands. The *editor* string is a command name, subject to search via the *PATH* variable. The value in the *FCEDIT* variable is used as a default when **-e** is not specified. If *FCEDIT* is null or unset, the value of the *EDITOR* variable is used. If *EDITOR* is null or unset, ed(1) is used as the editor.

-l (ell)

List the commands rather than invoking an editor on them. The commands are written in the sequence indicated by the *first* and *last* operands, as affected by **-r**, with each command preceded by the command number.

-n Suppress command numbers when listing with **-l**.

-r Reverse the order of the commands listed (with **-l**) or edited (with neither **-l** nor **-s**).

-s Re-execute the command without invoking an editor.

first

last Select the commands to list or edit. The number of previous commands that can be accessed are determined by the value of the *HISTSIZE* variable. The value of *first* or *last* or both are one of the following:

[+]*num*

A positive number representing a command number; command numbers can be displayed with the **-l** option.

-*num* A negative decimal number representing the command that was executed *num* of

commands previously. For example, -1 is the immediately previous command.

string A string indicating the most recently entered command that begins with that string. If the *old=new* operand is not also specified with **-s**, the string form of the first operand cannot contain an embedded equal sign.

The following variables affect the execution of **fc**:

FCEDIT Name of the editor to use for history editing.

HISTSIZE The number of previous commands that are accessible.

fg [*job*]

Move the specified *job* or the current job to the foreground.

getopts *optstring var*

The POSIX **getopts** command. The **getopts** command deprecates the older getopt(1) command. The first argument should be a series of letters, each possibly followed by a colon which indicates that the option takes an argument. The specified variable is set to the parsed option. The index of the next argument is placed into the shell variable *OPTIND*. If an option takes an argument, it is placed into the shell variable *OPTARG*. If an invalid option is encountered, *var* is set to '?'. It returns a false value (1) when it encounters the end of the options. A new set of arguments may be parsed by assigning OPTIND=1.

hash [**-rv**] [*command* ...]

The shell maintains a hash table which remembers the locations of commands. With no arguments whatsoever, the **hash** command prints out the contents of this table.

With arguments, the **hash** command removes each specified *command* from the hash table (unless they are functions) and then locates it. With the **-v** option, **hash** prints the locations of the commands as it finds them. The **-r** option causes the **hash** command to delete all the entries in the hash table except for functions.

jobid [*job*]

Print the process IDs of the processes in the specified *job*. If the *job* argument is omitted, use the current job.

jobs [**-lps**] [*job* ...]

Print information about the specified jobs, or all jobs if no *job* argument is given. The information printed includes job ID, status and command name.

If the **-l** option is specified, the PID of each job is also printed. If the **-p** option is specified, only the process IDs for the process group leaders are printed, one per line. If the **-s** option is specified, only the PIDs of the job commands are printed, one per line.

kill A built-in equivalent of kill(1) that additionally supports sending signals to jobs.

local [*variable ...*] [-]
See the *Functions* subsection.

printf A built-in equivalent of printf(1).

pwd [**-L** | **-P**]
Print the path of the current directory. The built-in command may differ from the program of the same name because the built-in command remembers what the current directory is rather than recomputing it each time. This makes it faster. However, if the current directory is renamed, the built-in version of pwd(1) will continue to print the old name for the directory.

If the **-P** option is specified, symbolic links are resolved. If the **-L** option is specified, the shell's notion of the current directory is printed (symbolic links are not resolved). This is the default.

read [**-p** *prompt*] [**-t** *timeout*] [**-er**] *variable ...*
The *prompt* is printed if the **-p** option is specified and the standard input is a terminal. Then a line is read from the standard input. The trailing newline is deleted from the line and the line is split as described in the section on *White Space Splitting (Field Splitting)* above, and the pieces are assigned to the variables in order. If there are more pieces than variables, the remaining pieces (along with the characters in *IFS* that separated them) are assigned to the last variable. If there are more variables than pieces, the remaining variables are assigned the null string.

Backslashes are treated specially, unless the **-r** option is specified. If a backslash is followed by a newline, the backslash and the newline will be deleted. If a backslash is followed by any other character, the backslash will be deleted and the following character will be treated as though it were not in *IFS*, even if it is.

If the **-t** option is specified and the *timeout* elapses before a complete line of input is supplied, the **read** command will return an exit status as if terminated by SIGALRM without assigning any values. The *timeout* value may optionally be followed by one of 's', 'm' or 'h' to explicitly specify seconds, minutes or hours. If none is supplied, 's' is assumed.

The **-e** option exists only for backward compatibility with older scripts.

The exit status is 0 on success, 1 on end of file, between 2 and 128 if an error occurs and greater than 128 if a trapped signal interrupts **read**.

readonly [**-p**] [*name* ...]

Each specified *name* is marked as read only, so that it cannot be subsequently modified or unset. The shell allows the value of a variable to be set at the same time as it is marked read only by using the following form:

readonly *name=value*

With no arguments the **readonly** command lists the names of all read only variables. If the **-p** option is specified, the read-only variables are printed as "**readonly** *name=value*" lines, suitable for re-input to the shell.

return [*exitstatus*]

See the *Functions* subsection.

set [**-/+abCEefIimnpTuVvx**] [**-/+o** *longname*] [**--** *arg* ...]

The **set** command performs three different functions:

With no arguments, it lists the values of all shell variables.

If options are given, either in short form or using the long "**-/+o** *longname*" form, it sets or clears the specified options as described in the section called *Argument List Processing*.

If the "**--**" option is specified, **set** will replace the shell's positional parameters with the subsequent arguments. If no arguments follow the "**--**" option, all the positional parameters will be cleared, which is equivalent to executing the command "shift $#". The "**--**" flag may be omitted when specifying arguments to be used as positional replacement parameters. This is not recommended, because the first argument may begin with a dash ('-') or a plus ('+'), which the **set** command will interpret as a request to enable or disable options.

setvar *variable value*

Assigns the specified *value* to the specified *variable*. The **setvar** command is intended to be used in functions that assign values to variables whose names are passed as parameters. In general it is better to write "*variable=value*" rather than using **setvar**.

shift [*n*]

Shift the positional parameters *n* times, or once if *n* is not specified. A shift sets the value of $1 to the value of $2, the value of $2 to the value of $3, and so on, decreasing the value of $# by

one. For portability, shifting if there are zero positional parameters should be avoided, since the shell may abort.

test A built-in equivalent of test(1).

times Print the amount of time spent executing the shell process and its children. The first output line shows the user and system times for the shell process itself, the second one contains the user and system times for the children.

trap [*action*] *signal* ...

trap -l Cause the shell to parse and execute *action* when any specified *signal* is received. The signals are specified by name or number. In addition, the pseudo-signal **EXIT** may be used to specify an *action* that is performed when the shell terminates. The *action* may be an empty string or a dash ('-'); the former causes the specified signal to be ignored and the latter causes the default action to be taken. Omitting the *action* and using only signal numbers is another way to request the default action. In a subshell or utility environment, the shell resets trapped (but not ignored) signals to the default action. The **trap** command has no effect on signals that were ignored on entry to the shell.

Option **-l** causes the **trap** command to display a list of valid signal names.

true A null command that returns a 0 (true) exit value.

type [*name* ...]

Interpret each *name* as a command and print the resolution of the command search. Possible resolutions are: shell keyword, alias, special shell builtin, shell builtin, command, tracked alias and not found. For aliases the alias expansion is printed; for commands and tracked aliases the complete pathname of the command is printed.

ulimit [**-HSabcdfklmnopstuvw**] [*limit*]

Set or display resource limits (see getrlimit(2)). If *limit* is specified, the named resource will be set; otherwise the current resource value will be displayed.

If **-H** is specified, the hard limits will be set or displayed. While everybody is allowed to reduce a hard limit, only the superuser can increase it. The **-S** option specifies the soft limits instead. When displaying limits, only one of **-S** or **-H** can be given. The default is to display the soft limits, and to set both the hard and the soft limits.

Option **-a** causes the **ulimit** command to display all resources. The parameter *limit* is not

acceptable in this mode.

The remaining options specify which resource value is to be displayed or modified. They are mutually exclusive.

-b *sbsize*
The maximum size of socket buffer usage, in bytes.

-c *coredumpsize*
The maximal size of core dump files, in 512-byte blocks. Setting *coredumpsize* to 0 prevents core dump files from being created.

-d *datasize*
The maximal size of the data segment of a process, in kilobytes.

-f *filesize*
The maximal size of a file, in 512-byte blocks.

-k *kqueues*
The maximal number of kqueues (see kqueue(2)) for this user ID.

-l *lockedmem*
The maximal size of memory that can be locked by a process, in kilobytes.

-m *memoryuse*
The maximal resident set size of a process, in kilobytes.

-n *nofiles*
The maximal number of descriptors that could be opened by a process.

-o *umtxp*
The maximal number of process-shared locks (see pthread(3)) for this user ID.

-p *pseudoterminals*
The maximal number of pseudo-terminals for this user ID.

-s *stacksize*
The maximal size of the stack segment, in kilobytes.

-t *time*

The maximal amount of CPU time to be used by each process, in seconds.

-u *userproc*
The maximal number of simultaneous processes for this user ID.

-v *virtualmem*
The maximal virtual size of a process, in kilobytes.

-w *swapuse*
The maximum amount of swap space reserved or used for this user ID, in kilobytes.

umask [**-S**] [*mask*]
Set the file creation mask (see umask(2)) to the octal or symbolic (see chmod(1)) value specified by *mask*. If the argument is omitted, the current mask value is printed. If the **-S** option is specified, the output is symbolic, otherwise the output is octal.

unalias [**-a**] [*name* ...]
The specified alias names are removed. If **-a** is specified, all aliases are removed.

unset [**-fv**] *name* ...
The specified variables or functions are unset and unexported. If the **-v** option is specified or no options are given, the *name* arguments are treated as variable names. If the **-f** option is specified, the *name* arguments are treated as function names.

wait [*job* ...]
Wait for each specified *job* to complete and return the exit status of the last process in the last specified *job*. If any *job* specified is unknown to the shell, it is treated as if it were a known job that exited with exit status 127. If no operands are given, wait for all jobs to complete and return an exit status of zero.

Command Line Editing

When **sh** is being used interactively from a terminal, the current command and the command history (see **fc** in *Built-in Commands*) can be edited using **vi**-mode command line editing. This mode uses commands similar to a subset of those described in the vi(1) man page. The command "set -o vi" (or "set -V") enables **vi**-mode editing and places **sh** into **vi** insert mode. With **vi**-mode enabled, **sh** can be switched between insert mode and command mode by typing <ESC>. Hitting <return> while in command mode will pass the line to the shell.

Similarly, the "set -o emacs" (or "set -E") command can be used to enable a subset of **emacs**-style command line editing features.

ENVIRONMENT

The following environment variables affect the execution of **sh**:

ENV	Initialization file for interactive shells.
LANG, LC_*	Locale settings. These are inherited by children of the shell, and is used in a limited manner by the shell itself.
OLDPWD	The previous current directory. This is used and updated by **cd**.
PWD	An absolute pathname for the current directory, possibly containing symbolic links. This is used and updated by the shell.
TERM	The default terminal setting for the shell. This is inherited by children of the shell, and is used in the history editing modes.

Additionally, environment variables are turned into shell variables at startup, which may affect the shell as described under *Special Variables*.

FILES

~/.profile	User’s login profile.
/etc/profile	System login profile.
/etc/shells	Shell database.
/etc/suid_profile	Privileged shell profile.

EXIT STATUS

If the *script* cannot be found, the exit status will be 127; if it cannot be opened for another reason, the exit status will be 126. Other errors that are detected by the shell, such as a syntax error, will cause the shell to exit with a non-zero exit status. If the shell is not an interactive shell, the execution of the shell file will be aborted. Otherwise the shell will return the exit status of the last command executed, or if the **exit** builtin is used with a numeric argument, it will return the argument.

SEE ALSO

builtin(1), chsh(1), echo(1), ed(1), emacs(1) (*ports/editors/emacs*), kill(1), printf(1), pwd(1), test(1), vi(1), execve(2), getrlimit(2), umask(2), wctype(3), editrc(5), shells(5)

HISTORY

A **sh** command, the Thompson shell, appeared in Version 1 AT&T UNIX. It was superseded in Version 7 AT&T UNIX by the Bourne shell, which inherited the name **sh**.

This version of **sh** was rewritten in 1989 under the BSD license after the Bourne shell from AT&T System V Release 4 UNIX.

AUTHORS

This version of **sh** was originally written by Kenneth Almquist.

BUGS

The **sh** utility does not recognize multibyte characters other than UTF-8. Splitting using *IFS* does not recognize multibyte characters.