

**NAME**

osh – old shell (command interpreter)

**SYNOPSIS**

**osh** [-v] [-l | -c *string* | -i | -l | -t | *file* [*arg1* ...]]

**DESCRIPTION**

*Osh* is an enhanced, backward-compatible port of the standard command interpreter from Sixth Edition UNIX. It may be used either as an interactive shell or as a non-interactive shell. Throughout this manual, '(+)' indicates those cases where *osh* is known to differ from the original *sh*(1), as it appeared in Sixth Edition UNIX.

The options are as follows:

- The shell reads and executes command lines from the standard input until end-of-file or **exit**.
- c *string*  
The shell executes *string* as a command line and exits.
- i (+) The shell behaves as an interactive shell by reading and executing commands from the appropriate rc files if possible (see *Startup and shutdown* below) before prompting the user, reading, and executing command lines from the standard input. The shell prints a diagnostic and exits with a non-zero status if it is not connected to a terminal.
- l (+) The shell behaves as a login shell by reading and executing commands from the appropriate rc files if possible (see *Startup and shutdown* below) before prompting the user, reading, and executing command lines from the standard input. The shell prints a diagnostic and exits with a non-zero status if it is not connected to a terminal.
- t The shell reads a single command line from the standard input, executes it, and exits.
- v (+) The shell verbosely prints the words of each command line to the standard error after performing parameter substitution and word splitting, but before executing the resulting command line.

The shell may also be invoked non-interactively to read, interpret, and execute a command file. The specified *file* and any arguments are treated as positional parameters (see *Parameter substitution* below) during execution of the command file.

Otherwise, if no arguments except for -v are specified and if both the standard input and standard error are connected to a terminal, the shell is interactive. An interactive shell prompts the user with a '%' (or '#' for the superuser) before reading each command line from the terminal.

(+) When an interactive shell starts, it reads and executes commands from the appropriate rc files if possible (see *Startup and shutdown* below) before reading and executing command lines from the terminal.

**Commands**

Each command is a sequence of non-blank command arguments separated by blanks (spaces or tabs). The first argument specifies the name of a command to be executed. Except for certain types of special arguments described below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument names a special command, the shell executes it (see *Special commands* below). Otherwise, the shell treats it as an external command, which is located as follows.

(+) If the command name contains no '/' characters, the sequence of directories in the environment variable PATH is searched for the first occurrence of an executable file by that name, which the shell attempts to execute. However, if the command name contains one or more '/' characters, the shell attempts to execute it without performing any PATH search.

(+) If an executable file does not begin with the proper magic number or a '#!shell' sequence, it is assumed to be a shell command file, and a new shell is automatically invoked to execute it. The environment variable EXECHELL specifies the shell which is invoked to execute such a file.

If a command cannot be found or executed, a diagnostic is printed.

**Command lines**

Commands separated by ‘|’ or ‘^’ constitute a chain of *filters*, or a *pipeline*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see *pipe(2)*) to its neighbors.

A *command line*, or *list*, consists of one or more pipelines separated, and perhaps terminated by ‘;’ or ‘&’. The semicolon designates sequential execution. The ampersand designates asynchronous execution, which causes the preceding pipeline to be executed without waiting for it to finish. The process ID of each command in such a pipeline is reported, so that it may be used if necessary for a subsequent *kill(1)*.

A list contained within parentheses such as ‘( list )’ is executed in a subshell and may appear in place of a simple command as a filter.

If a command line is syntactically incorrect, a diagnostic is printed.

**Termination reporting**

All terminations other than exit and interrupt are considered to be abnormal. If a sequential process terminates abnormally, a message is printed. The termination report for an asynchronous process is given upon execution of the first sequential command subsequent to its termination, or when the **wait** special command is executed. The following is a list of the possible termination messages:

- Hangup
- Quit
- Illegal instruction
- Trace/BPT trap
- IOT trap
- EMT trap
- Floating exception
- Killed
- Bus error
- Memory fault
- Bad system call
- Broken pipe (+)

For an asynchronous process, its process ID is prepended to the appropriate message. If a core image is produced, ‘— Core dumped’ is appended to the appropriate message.

**I/O redirection**

Each of the following argument forms is interpreted as a *redirection* by the shell itself. Such a redirection may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

A redirection of the form ‘<arg’ causes the file ‘arg’ to be used as the standard input (file descriptor 0) for the associated command.

A redirection of the form ‘>arg’ causes the file ‘arg’ to be used as the standard output (file descriptor 1) for the associated command. If ‘arg’ does not already exist, it is created; otherwise, it is truncated at the outset.

A redirection of the form ‘>>arg’ is the same as ‘>arg’, except if ‘arg’ already exists the command output is always appended to the end of the file.

For example, either of the following command lines:

```
% date >.dirlist ; pwd >>.dirlist ; ls -l >>.dirlist
% ( date ; pwd ; ls -l ) >.dirlist
```

creates on the file ‘.dirlist’, the current date and time, followed by the name and a long listing of the current working directory.

(+) A ‘<-’ redirection causes input for the associated command to be redirected from the standard input which existed when the shell was invoked. This allows a command file to be used as a filter.

A ‘>arg’ or ‘>>arg’ redirection associated with any but the last command of a pipeline is ineffectual, as is a

'<arg' redirection with any but the first.

The standard error (file descriptor 2) is never subject to redirection by the shell itself. Thus, commands may write diagnostics to a location where they have a chance to be seen. However, **fd2** provides a way to redirect the diagnostic output to another location.

If the file for a redirection cannot be opened or created, a diagnostic is printed.

### Quoting

The shell treats all *quoted* characters literally. This includes characters which may have special meaning to the shell such as '|', '^', ';', '&', '<', '>', and others described in this manual. If such characters are quoted, they represent themselves and may be passed as part of arguments.

An individual *backslash* (\) quotes, or *escapes*, the next individual character. A backslash followed by a newline is a special case which allows continuation of command-line input onto the next line. Each backslash-newline sequence in the input is translated into a blank.

Individual characters, and sequences of characters, are also quoted when enclosed by a matched pair of *double* (") or *single* (') quotes. For example:

```
% awk '{ print NR "\t" $0 }' README ^ more
```

causes *awk*(1) to write each line in 'README', preceded by its line number and a tab, to the standard output which is piped to *more*(1) for viewing. The quotes prevent the shell from trying to interpret any part of the string, which is then passed as a single argument to *awk*.

If a double or single quote appears but is not part of a matched pair, a diagnostic is printed.

### Parameter substitution

When the shell is invoked with arguments besides **-v**, it has additional string processing capabilities which are not otherwise available. Such a shell may be invoked as follows:

```
osh [-v] name [arg1 ...]
```

If the first character of *name* is not '-', it is taken as the name of a *command file*, or *shell script*, which is opened as the standard input for a new shell instance. Thus, the new shell reads and interprets command lines from the named file.

Otherwise, *name* is taken as one of the shell options, and a new shell instance is invoked to read and interpret command lines from its standard input. However, notice that the **-c** option followed by a *string* is the one case where the shell does not read and interpret command lines from its standard input. Instead, the string itself is taken as a command line and executed.

In each command line, an unquoted character sequence of the form '\$N', where *N* is a digit, is treated as a *positional parameter* by the shell. Each occurrence of a positional parameter in the command line is substituted with the value of the *N*th argument to the invocation of the shell (*argN*). '\$0' is substituted with *name*.

In all shell instances, '\$\$' is substituted with the process ID of the current shell. The value is represented as a 5-digit ASCII string, padded on the left with zeros when the process ID is less than 10000.

(+) All shell instances attempt to set the special parameters in the following list. '(\*)' indicates those which are always set. Otherwise, the parameter is unset when the shell cannot determine its value.

\$d	The value of the environment variable OSHDIR.
\$e	The value of the environment variable EXECShell.
\$h	The value of the environment variable HOME.
\$m	The value of the environment variable MANPATH.
\$n (*)	The number of positional parameters currently available to the shell.
\$p	The value of the environment variable PATH.
\$s (*)	The exit status of the last sequential command from the <i>previous</i> command line.

- `$t` The terminal name with which the standard input was associated when the shell was invoked, as determined by `ttyname(3)`. The value (if any) is equivalent to that given by `'tty <-'`.
- `$u` The effective user name of the current user, as determined by `getpwuid(3)`. The value (if any) is equivalent to that given by `'id -un'`.
- `$v (*)` The version of the current shell represented as a one-word, read-only string.

All substitution on a command line is performed *before* the line is interpreted. Thus, no action which alters the value of any parameter can have any effect on a reference to that parameter occurring on the *same* line.

A positional-parameter value may contain any number of characters with special meaning to the shell. Each one which is *unquoted*, or *unescaped*, within a positional-parameter value retains its special meaning when the value is substituted in a command line by the invoked shell.

Take the following two shell invocations for example:

```
% osh -c '$1' 'echo Hello! >/dev/null'
% osh -c '$1' 'echo Hello! \>/dev/null'
Hello! >/dev/null
```

In the first invocation, the `>` in the value substituted by `'$1'` retains its special meaning. This causes all output from **echo** to be redirected to `/dev/null`. However, in the second invocation, the meaning of `>` is escaped by `\` in the value substituted by `'$1'`. This causes the shell to pass `>/dev/null` as an argument to **echo** instead of interpreting it as a redirection.

### File name generation

Prior to executing a command, the shell scans each argument for unquoted `*`, `?`, or `[` characters. If one or more of these characters appears, the argument is treated as a *pattern* and causes the shell to search for file names which *match* it. Otherwise, the argument is used as is.

The meaning of each pattern character is as follows:

- o The `*` character in a pattern matches any string of characters in a file name (including the null string).
- o The `?` character in a pattern matches any single character in a file name.
- o The `[...]` brackets in a pattern specifies a class of characters which matches any single file-name character in the class. Within the brackets, each character is taken to be a member of the class. A pair of characters separated by an unquoted `-` specifies the class as a range which matches each character lexically between the first and second member of the pair, inclusive. A `-` matches itself when quoted or when first or last in the class.

Any other character in a pattern matches itself in a file name.

Notice that the `.` character at the beginning of a file name, or immediately following a `/`, is always special in that it must be matched explicitly. The same is true of the `/` character itself.

If the pattern contains no `/` characters, the current directory is always used. Otherwise, the specified directory is the one obtained by taking the pattern up to the last `/` before the first unquoted `*`, `?`, or `[`. The matching process matches the remainder of the pattern after this `/` against the files in the specified directory.

In any event, a list of file names is obtained from the current (or specified) directory which match the given pattern. This list is sorted in ascending ASCII order, and the new sequence of arguments replaces the given pattern. The same process is carried out for each of the given pattern arguments; the resulting lists are *not* merged. Finally, the shell attempts to execute the command with the resulting argument list.

If a pattern argument refers to a directory which cannot be opened, a `'No directory'` diagnostic is printed.

If a command has only *one* pattern argument, a `'No match'` diagnostic is printed if it fails to match any files. However, if a command has more than one pattern argument, a diagnostic is printed only when they *all* fail to match any files. Otherwise, each pattern argument failing to match any files is simply removed from the argument list.

**Startup and shutdown (+)**

If the first character of the file name used to invoke an interactive shell is ‘-’ (e.g., `-osh`), it is a login shell and tries to read and execute commands from the following four rc init files in sequence: `/usr/local/etc/osh.login`, `/usr/local/etc/osh.oshrc`, `$h/.osh.login`, and `$h/.oshrc`. The same is true when the shell is invoked with the `-l` option, regardless of the shell’s file name.

In the case where an interactive shell is not a login shell according to its file name, it tries to read and execute commands from the following two rc init files in sequence: `/usr/local/etc/osh.oshrc` and `$h/.oshrc`. The same is true when the shell is invoked with the `-i` option, regardless of the shell’s file name.

In any case, after the shell finishes its startup actions, it then prompts the user, reads, and executes command lines from the standard input as usual.

If the shell is invoked as a login shell, it tries to read and execute commands from `/usr/local/etc/osh.logout` and `$h/.osh.logout` in sequence upon logout. These two rc logout files may be used if necessary for cleanup upon termination of a login session by an EOT (see *End of file* below) or a SIGHUP signal (see *Signals* below).

Notice that the shell only performs the startup and shutdown actions described above for readable, regular rc files. If any rc file is *not* readable, the shell ignores it and continues as normal. If any rc file is *not* a regular file (or a link to a regular file), the shell ignores it, prints a diagnostic, and continues as normal.

In the normal case, a SIGINT or SIGQUIT signal received by the shell during execution of any rc file causes it to cease execution of that file without terminating. Thus, it may be desirable to use the **sign** special command to ignore these and other signals in some cases. For example, this is particularly true for `/usr/local/etc/osh.login`, `/usr/local/etc/osh.oshrc`, and `/usr/local/etc/osh.logout`.

The **exit** special command always causes the shell to terminate if it occurs in any rc file.

**History file (+)**

If the shell is invoked as an interactive shell, it tries to open the `$h/.osh.history` file to save the user’s command-line history. Notice that the history file must already exist as a writable, regular file (or a link to a regular file) when the shell is invoked to save the user’s command-line history. Otherwise, it will not do so.

An interactive shell reads each command line from its terminal and writes the words of each one to the history file as a history entry after performing parameter substitution and word splitting.

The shell does not read the history file or have any features that allow the user to make direct use of the saved history. Such features are available via standard external commands and also via the `history` command file that is available in the `v6scripts` collection. See <http://v6shell.org/v6scripts/history.osh> for full details.

Notice that the shell never creates or removes the `$h/.osh.history` file. It always leaves these actions to the user. For example:

```
% history -r ; history -c ; exec osh -l
```

causes `history` to remove the existing history file (if any), to create a new (empty) one, and causes the current shell to replace itself with a new login shell, while opening the new history file. This, and future, interactive shells then save the user’s command-line history as long as the history file exists.

If desired, the user can use the history file to repeat any command line as a command substitution with `sed(1)` and `osh`. Taking the following command line and history entry for example:

```
% history -n 44
Number Command Line
-----
44      uname -s | \
        if { fd2 -ef/dev/null egrep '(Darwin|Linux|NetBSD)' } \
        echo 'Running on GNU/Linux, Mac, or NetBSD.'
```

and then doing a:

```
% sed -n 44p <$h/.osh.history | osh
```

Running on GNU/Linux, Mac, or NetBSD.

causes sed to output the 44th command line from the history file via pipe for repetition as a command substitution by osh.

### End of file

An end-of-file in the shell's input causes it to exit. If the shell is interactive, this means it exits when the user types an EOT (^D) at the prompt.

### Special commands

The shell treats the following built-in commands specially.

**:** [*arg ...*]

Does nothing and sets the exit status to zero.

**cd** [*dir ...*] (+)

Is a synonym for the **chdir** special command.

**chdir** [*dir ...*]

Changes the shell's current working directory to *dir*. (+) If *dir* is an unquoted '-', the shell's previous working directory is used instead. Otherwise, if *dir* is not specified, the user's home directory is used by default.

**echo** [-n] [*string ...*] (+)

Writes its string arguments (if any) separated by blanks and terminated by a newline to the standard output. If '-n' is specified, the terminating newline is not written.

**exec** *command* [*arg ...*] (+)

Replaces the current shell with an instance of *command*, which must be external to the shell.

**exit** Causes the shell to cease execution of a file. This means exit has no effect at the prompt of an interactive shell.

**fd2** [-e] [-f *file*] *command* [*arg ...*] (+)

Redirects from/to file descriptor 2 for *command*. See the *fd2(1)* manual page for full details.

**goto** *label* [...] (+)

Transfers shell control to the ': *label*' line of the current command file. See the *goto(1)* manual page for full details.

**if** *expression* [*command* [*arg ...*]] (+)

Evaluates *expression* and conditionally executes *command* if appropriate. See the *if(1)* manual page for full details.

**login** [*arg ...*]

Replaces the current interactive shell with *login(1)*.

**newgrp** [*arg ...*]

Replaces the current interactive shell with *newgrp(1)*.

**setenv** *name* [*value*] (+)

Sets the environment variable *name* to the string *value*. If *value* is not specified, the environment variable *name* is set to the empty string.

**shift** Shifts all positional-parameter values to the left by 1, so that the old value of '\$2' becomes the new value of '\$1' and so forth. The value of '\$0' does not shift.

**sigign** [+ | - *signal\_number ...*] (+)

+ causes the specified signals to be ignored if possible, and - causes the specified signals to be reset to the default action if possible. If a signal was already ignored when the shell was invoked, it can never be reset with -. If no arguments are specified, a list is printed of those signals which are ignored by sigign in the current shell context.

**source** *file* [*arg1 ...*] (+)

Causes the shell to read and execute commands from *file* and return. The specified file and any arguments are treated as positional parameters (see *Parameter substitution* above) during execution of the file. The source command may be nested. As with command files, most shell-detected errors cause the shell to cease execution of the file. If the source command is nested and such an error occurs, all nested source commands terminate.

**umask** [*mask*] (+)

Sets the file creation mask (see *umask(2)*) to the octal value specified by *mask*. If the mask is not specified, its current value is printed.

**unsetenv** *name* (+)

Removes the variable *name* from the environment.

**wait** Waits for all asynchronous processes to terminate, reporting on abnormal terminations.

**Signals** (+)

An interactive or login shell always ignores the SIGINT, SIGQUIT, and SIGTERM signals (see *signal(3)*). A login shell also handles the SIGHUP signal, the receipt of which causes the shell to terminate the login session and to read and execute its rc logout files if possible.

If SIGHUP, SIGINT, SIGQUIT, or SIGTERM is already ignored when the shell starts, it is also ignored by the current shell and all of its child processes. Otherwise, SIGINT and SIGQUIT are reset to the default action for sequential child processes, whereas SIGHUP and SIGTERM are reset to the default action for all child processes.

When a non-interactive shell executes a command file, it does not handle or ignore any signal by default. Any other non-interactive shell ignores SIGINT and SIGQUIT.

For any signal not mentioned above, the shell inherits the signal action (default or ignore) from its parent process and passes it to its child processes. Remember that the **sigign** special command may be used to ignore signals when the shell does not do so by default.

Asynchronous child processes always ignore both SIGINT and SIGQUIT. Also, if such a process has not redirected its input with a '<', '|', or '^', the shell automatically redirects it to come from */dev/null*.

**EXIT STATUS** (+)

The exit status of the shell is generally that of the last command executed prior to end-of-file or **exit**.

However, if the shell is interactive and detects an error, it exits with a non-zero status if the user types an EOT at the next prompt.

Otherwise, if the shell is non-interactive and is reading commands from a file, any shell-detected error causes the shell to cease execution of that file. This results in a non-zero exit status.

A non-zero exit status returned by the shell itself is always one of the values described in the following list, each of which may be accompanied by an appropriate diagnostic:

- 2 The shell detected a syntax, redirection, or other error not described in this list.
- 125 An external command was found but did not begin with the proper magic number or a '#!shell' sequence, and a valid shell was not specified by EXECShell with which to execute it.
- 126 An external command was found but could not be executed.
- 127 An external command was not found.
- >128 An external command was terminated by a signal.

**ENVIRONMENT** (+)

Notice that the concept of 'user environment' was not defined in Sixth Edition UNIX. Thus, use of the following environment variables by this port of the shell is an enhancement:

**EXECShell**

If set to a non-empty string, the value of this variable is taken as the path name of the shell which is invoked to execute an external command when it does not begin with the proper magic number

or a '#!shell' sequence. Its value is available to the shell via the '\$e' special parameter.

### HOME

If set to a non-empty string, the value of this variable is taken as the user's home directory. Its value is available to the shell via the '\$h' special parameter and is the default directory for the **chdir** special command.

### MANPATH

If set, the value of this variable is taken as the sequence of directories used by *man(1)* to search for manual page files. Its value is available to the shell via the '\$m' special parameter.

### OSHDIR

If set to a non-empty string, the value of this variable is taken as the path name of a directory which may be used for temporary files. Its value is available to the shell via the '\$d' special parameter.

**PATH** If set to a non-empty string, the value of this variable is taken as the sequence of directories used by the shell to search for external commands. Its value is available to the shell via the '\$p' special parameter. Notice that the Sixth Edition UNIX shell always used the equivalent of './bin:/usr/bin', not PATH.

## FILES

*/dev/null*  
default source of input for asynchronous processes

*/usr/local/etc/osh.login* (+)  
system-wide rc init file for login shells

*/usr/local/etc/osh.oshrc* (+)  
system-wide rc init file for *all* interactive shells

*\$h.osh.history* (+)  
user history file for *all* interactive shells

*\$h.osh.login* (+)  
user rc init file for login shells

*\$h.oshrc* (+)  
user rc init file for *all* interactive shells

*/usr/local/etc/osh.logout* (+)  
system-wide rc logout file for login shells

*\$h.osh.logout* (+)  
user rc logout file for login shells

## SEE ALSO

awk(1), env(1), expr(1), fd2(1), goto(1), grep(1), if(1), kill(1), login(1), man(1), newgrp(1), sed(1), sh6(1)

Osh home page: <http://v6shell.org/>

## AUTHORS

This enhanced port of the Thompson shell is derived from Sixth Edition UNIX */usr/source/s2/sh.c*, which was principally written by Ken Thompson of Bell Labs. Jeffrey Allen Neitzel initially ported it in January 2004 and currently maintains it as *sh6(1)*. In addition, he is the principal developer and maintainer of this enhanced version of the shell, which is hereby made available as *osh(1)*.

## HISTORY

A *sh* command appeared as */bin/sh* in First Edition UNIX.

The Thompson shell was used as the standard command interpreter through Sixth Edition UNIX. Then, in the Seventh Edition, it was replaced by the Bourne shell. However, the Thompson shell was still distributed with the system as *osh* because of known portability problems with the Bourne shell's memory management in Seventh Edition UNIX.

**LICENSE**

See either the LICENSE file which is distributed with *osh* or <http://v6shell.org/license/> for full details.

**COPYRIGHT**

Copyright (c) 2003-2010

Jeffrey Allen Neitzel. All rights reserved.

Copyright (c) 2001-2002

Caldera International Inc. All rights reserved.

Copyright (c) 1985, 1989, 1991, 1993

The Regents of the University of California. All rights reserved.

**NOTES**

Unlike the original, this port of the shell can handle 8-bit character sets, as well as the UTF-8 encoding. The original, on the other hand, can only handle 7-bit ASCII.

Notice that certain shell oddities were historically undocumented in this manual page. Particularly noteworthy is the fact that there is no such thing as a usage error. Thus, the following shell invocations are all perfectly valid:

```
osh -cats_are_nice!!! ': "Good kitty =)"
osh -tabbies_are_too!
osh -s
```

The first two cases correspond to the `-c` and `-t` options respectively; the third case corresponds to the `-` option.

**SECURITY**

This port of the Thompson shell does not support being used in a set-ID context. If the effective user (group) ID of the shell process is not equal to its real user (group) ID, the shell prints a diagnostic and exits with a non-zero status. The reasons for this are as follows.

First, the way in which the shell uses positional parameters (see *Parameter substitution* above) makes it a simple matter to invoke an interactive shell from a command file if the user knows the name of the current terminal (if any). This is distinctly *not* a bug and can be very useful in the normal case.

However, if the shell did support set-ID execution, this could possibly allow a user to violate the security policy on a host where the shell is used. For example, if the shell were running a `setuid-root` command file, a regular user could invoke an interactive root shell as a result.

**BUGS**

The shell makes no attempt to recover from *read(2)* errors and exits if this system call fails for any reason.