NAME

sh, jsh – shell: the standard shell, and job control shell -- command interpreters

SYNOPSIS

/usr/bin/sh [**–acefhiknprstuvx**] [*argument*. . .]
/usr/xpg4/bin/sh [±**abCefhikmnoprstuvx**] [±**o** *option*] . . . [–**c** *string*] [*arg* . . .]

/usr/bin/jsh [–**acefhiknprstuvx**] [*argument*. . .]

AVAILABILITY

/usr/bin/sh
/usr/bin/jsh SUNWcsu

/usr/xpg4/bin/sh SUNWxcu4

DESCRIPTION

/usr/xpg4/bin/sh is identical to **/usr/bin/ksh**. See **ksh**(1).

/usr/bin/sh is a command programming language that executes commands read from a terminal or a file. The command **jsh** is an interface to the shell which provides all of the functionality of **sh** and enables Job Control (see ‘‘Job Control,’’ below). See ‘‘Invocation,’’ below for the meaning of arguments to the shell.

Definitions

A *blank* is a tab or a space. A *name* is a sequence of ASCII letters, digits, or underscores, beginning with a letter or an underscore. A *parameter* is a name, a digit, or any of the characters **∗**, **@**, **#**, **?**, **–**, **$**, and **!**\^.

USAGE

Commands

A *simple-command* is a sequence of non-blank *word*s separated by *blank*s. The first *word* specifies the name of the command to be executed. Except as specified below, the remaining *word*s are passed as arguments to the invoked command. The command name is passed as argument 0 (see **exec**(2)). The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) **200**+*status* if it terminates abnormally; see **signal**(5) for a list of status values.

A *pipeline* is a sequence of one or more *command*s separated by **|** . The standard output of each *command* but the last is connected by a **pipe**(2) to the standard input of the next *command*. Each *command* is run as a separate process; the shell waits for the last *command* to terminate. The exit status of a *pipeline* is the exit status of the last command in the *pipeline*.

A *list* is a sequence of one or more *pipeline*s separated by **;**, **&**, **&&**, or **| |** , and optionally terminated by **;** or **&**. Of these four symbols, **;** and **&** have equal precedence, which is lower than that of **&&** and **| |** . The symbols **&&** and **| |** also have equal precedence. A semicolon (**;**) causes sequential execution of the preceding *pipeline* (that is, the shell waits for the *pipeline* to finish before executing any commands following the semicolon); an ampersand (**&**) causes asynchronous execution of the preceding pipeline (that is, the shell does *not* wait for that pipeline to finish). The symbol **&&** (**| |**) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of newlines may appear in a *list*, instead of semicolons, to delimit commands.

A *command* is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the last *simple-command* executed in the command.

for *name* [**in** *word* ...] **do** *list* **done**
Each time a **for** command is executed, *name* is set to the next *word* taken from the **in** *word* list. If **in** *word* ... is omitted, then the **for** command executes the **do** *list* once for each positional parameter that is set (see ''Parameter Substitution,'' below). Execution ends when there are no more words in the list.

case *word* **in** [*pattern* [| *pattern*] **)** *list* **;;**] ... **esac**
A **case** command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see ''File Name Generation'') except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

if *list* **; then** *list* **;** [**elif** *list* **; then** *list* **;**] ... [**else** *list* **;**] **fi**
The *list* following **if** is executed and, if it returns a zero exit status, the *list* following the first **then** is executed. Otherwise, the *list* following **elif** is executed and, if its value is zero, the *list* following the next **then** is executed. Failing that, the **else** *list* is executed. If no **else** *list* or **then** *list* is executed, then the **if** command returns a zero exit status.

while *list* **do** *list* **done**
A **while** command repeatedly executes the **while** *list* and, if the exit status of the last command in the list is zero, executes the **do** *list*; otherwise the loop terminates. If no commands in the **do** *list* are executed, then the **while** command returns a zero exit status; **until** may be used in place of **while** to negate the loop termination test.

(*list***)**
Execute *list* in a sub-shell.

{ *list***;}**
list is executed in the current (that is, parent) shell. The **{** must be followed by a space.

name **()** **{** *list***;}**
Define a function which is referenced by *name*. The body of the function is the *list* of commands between **{** and **}**. The **{** must be followed by a space. Execution of functions is described below (see ''Execution''). The **{** and **}** are unnecessary if the body of the function is a *command* as defined above, under ''Commands.''

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

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

Comments Lines

A word beginning with **#** causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The shell reads commands from the string between two grave accents (‘ ‘) and the standard output from these commands may be used as all or part of a word. Trailing newlines from the standard output are removed.

No interpretation is done on the string before the string is read, except to remove backslashes (\) used to escape other characters. Backslashes may be used to escape a grave accent (‘) or another backslash (\) and are removed before the command string is read. Escaping grave accents allows nested command substitution. If the command substitution lies within a pair of double quotes (**"**...**‘**...**‘**... **"**), a backslash used to escape a double quote (**"**) will be removed; otherwise, it will be left intact.

If a backslash is used to escape a newline character (**newline**), both the backslash and the newline are removed (see the later section on “Quoting”). In addition, backslashes used to escape dollar signs (**$**) are removed. Since no parameter substitution is done on the command string before it is read, inserting a backslash to escape a dollar sign has no effect. Backslashes that precede characters other than ****, **‘**, **"**, **newline**, and **$** are left intact when the command string is read.

Parameter Substitution

The character **$** is used to introduce substitutable *parameters*. There are two types of parameters, positional and keyword. If *parameter* is a digit, it is a positional parameter. Positional parameters may be assigned values by **set**. Keyword parameters (also known as variables) may be assigned values by writing:

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

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same *name*.

${*parameter***}**
The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is * or @, all the positional parameters, starting with **$1**, are substituted (separated by spaces). Parameter **$0** is set from argument zero when the shell is invoked.

${*parameter***:–***word***}**
If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

${*parameter***:=***word***}**
If *parameter* is not set or is null set it to *word*; the value of the parameter is substituted. Positional parameters may not be assigned in this way.

${*parameter***:?***word***}**
If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message “parameter null or not set” is printed.

${*parameter***:+***word***}**
If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

echo ${d:–‘pwd‘}

If the colon (**:**) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell.

The number of positional parameters in decimal.

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

? The decimal value returned by the last synchronously executed command.

$ The process number of this shell.

! The process number of the last background command invoked.

The following parameters are used by the shell. The parameters in this section are also referred to as environment variables.

HOME The default argument (home directory) for the **cd** command, set to the user's login directory by **login**(1) from the password file (see **passwd**(4)).

PATH The search path for commands (see ‘‘Execution,’’ below).

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

MAIL If this parameter is set to the name of a mail file *and* the **MAILPATH** parameter is not set, the shell informs the user of the arrival of mail in the specified file.

MAILCHECK
This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** parameters. The default value is **600** seconds (10 minutes). If set to 0, the shell will check before each prompt.

MAILPATH
A colon (**:**) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is you have mail.

PS1 Primary prompt string, by default ‘‘.SB $ ’’.

PS2 Secondary prompt string, by default ‘‘ > ’’.

IFS Internal field separators, normally **space**, **tab**, and **newline** (see ‘‘Blank Interpretation’’).

SHACCT
If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed.

SHELL When the shell is invoked, it scans the environment (see **ENVIRONMENT**, below) for this name.

LC_CTYPE
Determines how the shell handles characters. When **LC_CTYPE** is set to a valid value, the shell can display and handle text and filenames containing valid characters for that locale. The shell can display and handle Extended Unix Code (EUC) characters where any individual character can be 1, 2, or 3 bytes wide. The shell can also handle EUC characters of 1, 2, or more column widths. In the "C" locale, only characters from ISO 8859-1 are valid.

LC_MESSAGES
Determines how diagnostic and informative messages are presented. This includes the language and style of the messages, and the correct form of affirmative and negative responses. In the "C" locale, the messages are presented in the default form found in the program itself (in most cases, U.S. English).

If **LC_CTYPE** and **LC_MESSAGES** (see **environ**(5)) are not set in the environment, the operational behavior of the shell for each corresponding locale category is determined by the value of the **LANG** environment variable. If **LC_ALL** is set, its contents are used to override both the **LANG** and the other **LC_*** variables. If none of the above variables is set in the environment, the "C" (U.S. style) locale determines how the shell behaves.

The shell gives default values to **PATH**, **PS1**, **PS2**, **MAILCHECK**, and **IFS**. **HOME** and **MAIL** are set by **login**(1).

Blank Interpretation

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

Input/Output Redirection

A command’s input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may precede or follow a *command* and are *not* passed on as arguments to the invoked command. Note: Parameter and command substitution occurs before *word* or *digit* is used.

<*word* Use file *word* as standard input (file descriptor 0).

>*word* Use file *word* as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length.

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

<<[–]*word* After parameter and command substitution is done on *word*, the shell input is read up to the first line that literally matches the resulting *word*, or to an EOF. If, however, – is appended to **<<**:

1) leading tabs are stripped from *word* before the shell input is read (but after parameter and command substitution is done on *word*),
2) leading tabs are stripped from the shell input as it is read and before each line is compared with *word*, and
3) shell input is read up to the first line that literally matches the resulting *word*, or to an EOF.

If any character of *word* is quoted (see ''Quoting,'' later), no additional processing is done to the shell input. If no characters of *word* are quoted:

1) parameter and command substitution occurs,
2) (escaped) **\newline**s are removed, and
3) \ must be used to quote the characters \, **$**, and **`**.

The resulting document becomes the standard input.

<&*digit* Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using **>&***digit*.

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

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default **0** or **1**). For example:

. . . 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

. . . 1>*xxx* **2>&1**

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

Using the terminology introduced on the first page, under ''Commands,'' if a *command* is composed of several *simple commands*, redirection will be evaluated for the entire *command* before it is evaluated for each *simple command*. That is, the shell evaluates redirection for the entire *list*, then each *pipeline* within the *list*, then each *command* within each *pipeline*, then each *list* within each *command*.

If a command is followed by **&** the default standard input for the command is the empty file **/dev/null**. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

File Name Generation

Before a command is executed, each command *word* is scanned for the characters **∗**, **?**, and **[** . If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character **.** at the start of a file name or immediately following a **/**, as well as the character **/** itself, must be matched explicitly.

∗ Matches any string, including the null string.

? Matches any single character.

[. . . **]** Matches any one of the enclosed characters. A pair of characters separated by – matches any character lexically between the pair, inclusive. If the first character following the opening **[** is a **!**, any character not enclosed is matched.

Note: All quoted characters (see below) must be matched explicitly in a filename.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

; & ( ) | ^ < > newline space tab

A character may be *quoted* (that is, made to stand for itself) by preceding it with a backslash (\) or inserting it between a pair of quote marks (**´´** or **""**). During processing, the shell may quote certain characters to prevent them from taking on a special meaning. Backslashes used to quote a single character are removed from the word before the command is executed. The pair **\newline** is removed from a word before command and parameter substitution.

All characters enclosed between a pair of single quote marks (**´´**), except a single quote, are quoted by the shell. Backslash has no special meaning inside a pair of single quotes. A single quote may be quoted inside a pair of double quote marks (for example, **"´"**), but a single quote can not be quoted inside a pair of single quotes.

Inside a pair of double quote marks (**""**), parameter and command substitution occurs and the shell quotes the results to avoid blank interpretation and file name generation. If **$∗** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by quoted spaces (**"$1 $2** . . .**"**); however, if **$@** is within a pair of double quotes, the positional parameters are substituted and quoted, separated by unquoted spaces (**"$1" "$2"** . . .). \ quotes the characters \, **‘**, **,** and **$**. The pair **\newline** is removed before parameter and command substitution. If a backslash precedes characters other than \, **‘**, **,** **$**, and newline, then the backslash itself is quoted by the shell.

Prompting

When used interactively, the shell prompts with the value of **PS1** before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (that is, the value of **PS2**) is issued.

Environment

The *environment* (see **environ**(5)) is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment (see also **set** –**a**). A parameter may be removed from the environment with the **unset** command. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by **unset**, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

TERM=450 *command*
and
(export TERM; TERM=450; *command***)**

are equivalent as far as the execution of *command* is concerned if *command* is not a Special Command. If *command* is a Special Command, then

TERM=450 *command*

will modify the **TERM** variable in the current shell.

If the –**k** flag is set, *all* keyword arguments are placed in the environment, even if they occur after the command name. The following example first prints **a=b c** and **c**:

```
echo a=b c
a=b c
set –k
echo a=b c
c
```

Signals

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent, with the exception of signal 11 (but see also the **trap** command below).

Execution

Each time a command is executed, the command substitution, parameter substitution, blank interpretation, input/output redirection, and filename generation listed above are carried out. If the command name matches the name of a defined function, the function is executed in the shell process (note how this differs from the execution of shell script files, which require a sub-shell for invocation). If the command name does not match the name of a defined function, but matches one of the *Special Commands* listed below, it is executed in the shell process.

The positional parameters **$1**, **$2**, ... are set to the arguments of the function. If the command name matches neither a *Special Command* nor the name of a defined function, a new process is created and an attempt is made to execute the command via **exec**(2).

The shell parameter **PATH** defines the search path for the directory containing the command. Alternative directory names are separated by a colon (**:**). The default path is **/usr/bin**. The current directory is specified by a null path name, which can appear immediately after the equal sign, between two colon delimiters anywhere in the path list, or at the end of the path list. If the command name contains a **/** the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an **a.out** file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary *execs* later). If the command was found in a relative directory, its location must be re-determined whenever the current directory changes. The shell forgets all remembered locations whenever the **PATH** variable is changed or the **hash −r** command is executed (see below).

Special Commands

Input/output redirection is now permitted for these commands. File descriptor 1 is the default output location. When Job Control is enabled, additional *Special Commands* are added to the shell's environment (see ''Job Control'').

: No effect; the command does nothing. A zero exit code is returned.

. *filename*
Read and execute commands from *filename* and return. The search path specified by **PATH** is used to find the directory containing *filename*.

bg [%*jobid . . .*]
When Job Control is enabled, the **bg** command is added to the user's environment to manipulate jobs. Resumes the execution of a stopped job in the background. If %*jobid* is omitted the current job is assumed. (See **"Job Control"** section below for more detail).

break [*n*]
Exit from the enclosing **for** or **while** loop, if any. If *n* is specified, break *n* levels.

cd [*argument*]
Change the current directory to *argument*. The shell parameter **HOME** is the default *argument*. The shell parameter **CDPATH** defines the search path for the directory containing *argument*. Alternative directory names are separated by a colon (**:**). The default path is **<null>** (specifying the current directory). Note: The current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *argument* begins with a **/** the search path is not used. Otherwise, each directory in the path is searched for *argument*.

chdir [*dir*]
chdir changes the shell's working directory to directory *dir*. If no argument is given, change to the home directory of the user. If *dir* is a relative pathname not found in the current directory, check for it in those directories listed in the **CDPATH** variable. If *dir* is the name of a shell variable whose value starts with a **/**, change to the directory named by that value.

continue [*n*]
Resume the next iteration of the enclosing **for** or **while** loop. If *n* is specified,

resume at the *n*-th enclosing loop.

echo [*arguments* ...]

The words in *arguments* are written to the shell's standard output, separated by space characters. See **echo**(1) for fuller usage and description.

eval [*argument* ...]

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

exec [*argument* ...]

The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

exit [*n*]

Causes the calling shell or shell script to exit with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an EOF will also cause the shell to exit.)

export [*name* ...]

The given *name*s are marked for automatic export to the *environment* of subsequently executed commands. If no arguments are given, variable names that have been marked for export during the current shell's execution are listed. (Variable names exported from a parent shell are listed only if they have been exported again during the current shell's execution.) Function names are *not* exported.

fg [%*jobid* ...]

When Job Control is enabled, the **fg** command is added to the user's environment to manipulate jobs. Resumes the execution of a stopped job in the foreground, also moves an executing background job into the foreground. If %*jobid* is omitted the current job is assumed. (See **"Job Control"** section below for more detail).

getopts

Use in shell scripts to support command syntax standards (see **intro**(1)); it parses positional parameters and checks for legal options. See **getoptcvt**(1) for usage and description.

hash [**−r**] [*name* ...]

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

jobs [**−p**|**−l**] [%*jobid* ...]

jobs –x *command* [*arguments*]
Reports all jobs that are stopped or executing in the background. If %*jobid* is omitted, all jobs that are stopped or running in the background will be reported. (See **"Job Control"** section below for more detail).

kill [*–sig*] %*job* **. . .**
kill –l Sends either the **TERM** (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in **signal**(5) stripped of the prefix ''SIG'' with the exception that **SIGCHD** is named **CHLD**). If the signal being sent is **TERM** (terminate) or **HUP** (hangup), then the job or process will be sent a **CONT** (continue) signal if it is stopped. The argument *job* can be the process id of a process that is not a member of one of the active jobs. See **"Job Control" section below for a description of the format of** *job*. In the second form, **kill –l**, the signal numbers and names are listed. (See **kill**(1)).

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

newgrp [*argument*]
Equivalent to **exec newgrp** *argument.* See **newgrp**(1) for usage and description.

pwd Print the current working directory. See **pwd**(1) for usage and description.

read *name* . . .
One line is read from the standard input and, using the internal field separator, **IFS** (normally space or tab), to delimit word boundaries, the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. Lines can be continued using **\newline**. Characters other than **newline** can be quoted by preceding them with a backslash. These backslashes are removed before words are assigned to *names*, and no interpretation is done on the character that follows the backslash. The return code is **0**, unless an EOF is encountered.

readonly [*name* . . .]
The given *name*s are marked *readonly* and the values of the these *name*s may not be changed by subsequent assignment. If no arguments are given, a list of all *readonly* names is printed.

return [*n*]
Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

set [**—aefhkntuvx** [*argument* . . .]]

–a Mark variables which are modified or created for export.

–e Exit immediately if a command exits with a non-zero exit status.

–f Disable file name generation.

–h Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).

–k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

–n Read commands but do not execute them.

–t Exit after reading and executing one command.

–u Treat unset variables as an error when substituting.

–v Print shell input lines as they are read.

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

– – Do not change any of the flags; useful in setting **$1** to –.

Using **+** rather than – causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in **$–**. The remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, ... If no arguments are given the values of all names are printed.

shift [*n*]
The positional parameters from **$***n***+1** . . . are renamed **$1** . . . . If *n* is not given, it is assumed to be 1.

stop *pid . . .*
Halt execution of the process number *pid*. (see **ps**(1)).

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

test
Evaluate conditional expressions. See **test**(1) for usage and description.

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

trap [*argument n* [*n2* . . .]]
The command *argument* is to be read and executed when the shell receives numeric or symbolic signal(s) (*n*). (Note: *argument* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number or corresponding symbolic names. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. An attempt to trap on signal 11 (memory fault) produces an error. If *argument* is absent all trap(s) *n* are reset to their original values. If *argument* is the null string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *argument* is executed on exit from the shell. The **trap** command with no arguments prints a list of commands associated with each signal number.

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

ulimit [–[**HS**][**a** | **cdfnstv**]]

ulimit [–[**HS**][**c** | **d** | **f** | **n** | **s** | **t** | **v**]] *limit*
ulimit prints or sets hard or soft resource limits. These limits are described in

getrlimit(2).

If *limit* is not present, **ulimit** prints the specified limits. Any number of limits may be printed at one time. The −**a** option prints all limits.

If *limit* is present, **ulimit** sets the specified limit to *limit*. The string **unlimited** requests the largest valid limit. Limits may be set for only one resource at a time. Any user may set a soft limit to any value below the hard limit. Any user may lower a hard limit. Only a super-user may raise a hard limit; see **su**(1M).

The −**H** option specifies a hard limit. The −**S** option specifies a soft limit. If neither option is specified, **ulimit** will set both limits and print the soft limit.

The following options specify the resource whose limits are to be printed or set. If no option is specified, the file size limit is printed or set.

- −**c** maximum core file size (in 512-byte blocks)
- −**d** maximum size of data segment or heap (in kbytes)
- −**f** maximum file size (in 512-byte blocks)
- −**n** maximum file descriptor plus 1
- −**s** maximum size of stack segment (in kbytes)
- −**t** maximum CPU time (in seconds)
- −**v** maximum size of virtual memory (in kbytes)

(Run the **sysdef**(1M) command to obtain the maximum possible limits for your system. The values reported are in hexidecimal, but can be translated into decimal numbers using the **bc**(1) command. Also, see **swap**(1M).)

Example of ulimit: to limit the size of a core file dump to 0 Megabytes, type the following:

ulimit -c 0

umask [*nnn*]

The user file-creation mask is set to *nnn* (see **umask**(1)). If *nnn* is omitted, the current value of the mask is printed.

unset [*name* . . .]

For each *name*, remove the corresponding variable or function value. The variables **PATH**, **PS1**, **PS2**, **MAILCHECK**, and **IFS** cannot be unset.

wait [*n*]

Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code will be zero.

Invocation

If the shell is invoked through **exec**(2) and the first character of argument zero is −, commands are initially read from **/etc/profile** and from **$HOME/.profile**, if such files exist. Thereafter, commands are read as described below, which is also the case when the shell is invoked as **/usr/bin/sh**. The flags below are interpreted by the shell on invocation only. Note: Unless the −**c** or −**s** flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

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

–i If the **–i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.

–p If the **–p** flag is present, the shell will not set the effective user and group IDs to the real user and group IDs.

–r If the **–r** flag is present the shell is a restricted shell (see **rsh**(1M)).

–s If the **–s** flag is present or if no arguments remain, commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for *Special Commands*) is written to file descriptor 2.

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

Job Control (jsh)

When the shell is invoked as **jsh**, Job Control is enabled in addition to all of the functionality described previously for **sh**. Typically Job Control is enabled for the interactive shell only. Non-interactive shells typically do not benefit from the added functionality of Job Control.

With Job Control enabled every command or pipeline the user enters at the terminal is called a *job*. All jobs exist in one of the following states: foreground, background or stopped. These terms are defined as follows: 1) a job in the foreground has read and write access to the controlling terminal; 2) a job in the background is denied read access and has conditional write access to the controlling terminal (see **stty**(1)); 3) a stopped job is a job that has been placed in a suspended state, usually as a result of a **SIGTSTP** signal (see **signal**(5)).

Every job that the shell starts is assigned a positive integer, called a *job number* which is tracked by the shell and will be used as an identifier to indicate a specific job. Additionally the shell keeps track of the *current* and *previous* jobs. The *current job* is the most recent job to be started or restarted. The *previous job* is the first non-current job.

The acceptable syntax for a Job Identifier is of the form:

%*jobid*

where, *jobid* may be specified in any of the following formats:

% or **+** for the current job

– for the previous job

?<*string*> specify the job for which the command line uniquely contains *string*.

n for job number *n*, where *n* is a job number

pref where *pref* is a unique prefix of the command name (for example, if the command **ls –l name** were running in the background, it could be referred to as **%ls**); *pref* cannot contain blanks unless it is quoted.

When Job Control is enabled, the following commands are added to the user's environment to manipulate jobs:

bg [%*jobid* . . .]
: Resumes the execution of a stopped job in the background. If %*jobid* is omitted the current job is assumed.

fg [%*jobid* . . .]
: Resumes the execution of a stopped job in the foreground, also moves an executing background job into the foreground. If %*jobid* is omitted the current job is assumed.

jobs [**–p**|**–l**] [%*jobid* ...]

jobs **–x** *command* [*arguments*]
: Reports all jobs that are stopped or executing in the background. If %*jobid* is omitted, all jobs that are stopped or running in the background will be reported. The following options will modify/enhance the output of **jobs**:

  –l Report the process group ID and working directory of the jobs.

  –p Report only the process group ID of the jobs.

  –x Replace any *jobid* found in *command* or *arguments* with the corresponding process group ID, and then execute *command* passing it *arguments*.

kill [**–signal**] %*jobid*
: Builtin version of **kill** to provide the functionality of the **kill** command for processes identified with a *jobid*.

stop %*jobid* . . .
: Stops the execution of a background job(s).

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

wait [%*jobid* . . .]
: **wait** builtin accepts a job identifier. If %*jobid* is omitted **wait** behaves as described above under **Special Commands**.

EXIT CODES

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

jsh Only

If the shell is invoked as **jsh** and an attempt is made to exit the shell while there are stopped jobs, the shell issues one warning:

There are stopped jobs.

This is the only message. If another exit attempt is made, and there are still stopped jobs they will be sent a **SIGHUP** signal from the kernel and the shell is exited.

FILES

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

SEE ALSO

bc(1), **intro**(1), **echo**(1), **getoptcvt**(1) **login**(1), **newgrp**(1), **pwd**(1), **ps**(1), **shell_builtins**(1), **stty**(1), **rsh**(1M), **swap**(1M), **sysdef**(1M), **dup**(2), **exec**(2), **fork**(2), **getrlimit**(2), **pipe**(2), **ulimit**(2), **setlocale**(3C), **profile**(4), **passwd**(4), **environ**(5), **signal**(5)

NOTES

Words used for filenames in input/output redirection are not interpreted for filename generation (see **File Name Generation**, above). For example, **cat file1 >a*** will create a file named **a***.

Because commands in pipelines are run as separate processes, variables set in a pipeline have no effect on the parent shell.

If you get the error message **cannot fork**,**too many processes**, try using the **wait**(1) command to clean up your background processes. If this doesn't help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

Only the last process in a pipeline can be waited for.

If a command is executed, and a command with the same name is installed in a directory in the search path before the directory where the original command was found, the shell will continue to **exec** the original command. Use the **hash** command to correct this situation.