NAME

dash — command interpreter (shell)

SYNOPSIS

dash [**-aCefnuvxIimqVEbp**] [**+aCefnuvxIimqVEbp**] [**-o** *option_name*]
[**+o** *option_name*] [*command_file* [*argument ...*]]
dash -c [**-aCefnuvxIimqVEbp**] [**+aCefnuvxIimqVEbp**] [**-o** *option_name*]
[**+o** *option_name*] *command_string* [*command_name* [*argument ...*]]
dash -s [**-aCefnuvxIimqVEbp**] [**+aCefnuvxIimqVEbp**] [**-o** *option_name*]
[**+o** *option_name*] [*argument ...*]

DESCRIPTION

dash is the standard command interpreter for the system. The current version of **dash** is in the process of being changed to conform with the POSIX 1003.2 and 1003.2a specifications for the shell. This version has many features which make it appear similar in some respects to the Korn shell, but it is not a Korn shell clone (see ksh(1)). Only features designated by POSIX, plus a few Berkeley extensions, are being incorporated into this shell. This man page is not intended to be a tutorial or 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 running 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 and the file can be executed directly by the shell.

Invocation

If no args are present and if the standard input of the shell is connected to a terminal (or if the **-i** flag is set), and the **-c** option is not present, 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 .profile if they exist. If the environment variable ENV is set on entry to an interactive shell, or is set in the .profile of a login shell, the shell next reads commands from the file named in ENV. 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 interactive shell inside the ENV file. To set the ENV variable to some file, place the following line in your .profile of your home directory

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

substituting for ".shinit" any filename you wish.

If command line arguments besides the options have been specified, then the shell treats the first argument 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.

Argument List Processing

All of the single letter options that have a corresponding name can be used as an argument to the **-o** option. The set **-o** name is provided next to the single letter option in the description below. Specifying a dash "-" turns the option on, while using a plus "+" disables the option. The following options can be set from the

command line or with the **set** builtin (described later).

-a *allexport*	Export all variables assigned to.
-c	Read commands from the *command_string* operand instead of from the standard input. Special parameter 0 will be set from the *command_name* operand and the positional parameters ($1, $2, etc.) set from the remaining argument operands.
-C *noclobber*	Don't overwrite existing files with ">".
-e *errexit*	If not interactive, exit immediately if any untested command fails. The exit status of a command is considered to be explicitly tested if the command is used to control an **if**, **elif**, **while**, or **until**; or if the command is the left hand operand of an "&&" or "‖" operator.
-f *noglob*	Disable pathname expansion.
-n *noexec*	If not interactive, read commands but do not execute them. This is useful for checking the syntax of shell scripts.
-u *nounset*	Write a message to standard error when attempting to expand a variable that is not set, and if the shell is not interactive, exit immediately.
-v *verbose*	The shell writes its input to standard error as it is read. Useful for debugging.
-x *xtrace*	Write each command to standard error (preceded by a '+ ') before it is executed. Useful for debugging.
-I *ignoreeof*	Ignore EOF's from input when interactive.
-i *interactive*	Force the shell to behave interactively.
-l	Make dash act as if it had been invoked as a login shell.
-m *monitor*	Turn on job control (set automatically when interactive).
-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. with **set**).
-V *vi*	Enable the built-in vi(1) command line editor (disables **-E** if it has been set).
-E *emacs*	Enable the built-in emacs(1) command line editor (disables **-V** if it has been set).
-b *notify*	Enable asynchronous notification of background job completion. (UNIMPLEMENTED for 4.4alpha)
-p *priv*	Do not attempt to reset effective uid if it does not match uid. This is not set by default to help avoid incorrect usage by setuid root programs via system(3) or popen(3).

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 that are special to the shell called "operators". There are two types of operators: control operators and redirection operators (their meaning is discussed later). Following is a list of operators:

Control operators:

```
& && ( ) ; ;; | || <newline>
```

Redirection operators:

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

Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell, such as operators, whitespace, or keywords. There are three types of quoting: matched single quotes, matched double quotes, and backslash.

Backslash

A backslash preserves the literal meaning of the following character, with the exception of ⟨newline⟩. A backslash preceding a ⟨newline⟩ is treated as a line continuation.

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).

Double Quotes

Enclosing characters within double quotes preserves the literal meaning of all characters except dollarsign (`$`), backquote (`‘`), and backslash (`\`). The backslash inside double quotes is historically weird, and serves to quote only the following characters:

```
$ ` " \ <newline>.
```

Otherwise it remains literal.

Reserved Words

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 reserved words:

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

Their meaning is discussed later.

Aliases

An alias is a name and corresponding value set using the `alias`(1) builtin command. Whenever a reserved word may occur (see above), and after checking for reserved words, 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` ⟨return⟩

would become

`ls -F foobar` ⟨return⟩

Aliases provide a convenient way for naive users to create shorthands for commands without having to learn how to create functions with arguments. They can also be used to create lexically obscure code. This use is discouraged.

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 POSIX 1003.2 document). Essentially though, a line is read and if

the first word of the line (or after a control operator) is not a reserved word, 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. 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 "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 item 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`

where *redir-op* is one of the redirection operators mentioned previously. Following is a list of the possible redirections. The `[n]` is an optional number between 0 and 9, as in '3' (not '`[3]`'), that refers to a file descriptor.

[n]> file	Redirect standard output (or n) to file.
[n]>| file	Same, but override the **-C** option.
[n]>> file	Append standard output (or n) to file.
[n]< file	Redirect standard input (or n) from file.
[n1]<&n2	Copy file descriptor n2 as stdout (or fd n1). fd n2.
[n]<&-	Close standard input (or n).
[n1]>&n2	Copy file descriptor n2 as stdin (or fd n1). fd n2.
[n]>&-	Close standard output (or n).
[n]<> file	Open file for reading and writing on standard input (or 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 "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, builtin commands, and normal programs -- and the command is searched for (by name) in that order. They each are 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 builtins are executed internally to the shell, without spawning a new process.

Otherwise, if the command name doesn't match a function or builtin, 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 "#!", so `execve`(2) returns `ENOEXEC` then) the shell will interpret the program in a subshell. The child shell will reinitialize itself in this case, so that the effect will be as if a new shell had been invoked to handle the ad-hoc shell script, except that the location of hashed commands located in the parent shell will be remembered by the child.

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 builtin command by that name. If a builtin 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 `PATH` 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 behaviour 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 builtin commands return exit codes, as does an executed shell function.

If a command consists entirely of variable assignments then the exit status of the command is that of the last command substitution if any, otherwise 0.

Complex Commands

Complex commands are combinations of simple commands with control operators or reserved words, 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.

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.

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

If the reserved word ! does not precede the pipeline, the exit status is the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status is 1; if the last command returns 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 next) to be executed sequentially; a & causes asynchronous execution of the preceding AND-OR-list.

Note that unlike some other shells, each process in the pipeline is a child of the invoking shell (unless it is a shell builtin, in which case it executes in the current shell -- but any effect it has on the environment is wiped).

Background Commands -- &

If a command is terminated by the control operator ampersand (&), the shell executes the command asynchronously -- that is, 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`.

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 and only if the exit status of the first command is zero. "||" is similar, but executes the second command if and only 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 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 syntax of the for command is

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

The words following `in` are expanded, and then the list is executed repeatedly with the variable set to each word in turn. Omitting in word ... is equivalent to in "$@".

The syntax of the break and continue command is

```
break [ num ]
continue [ num ]
```

Break terminates the num innermost for or while loops. Continue continues with the next iteration of the innermost loop. These are implemented as builtin 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. The "(" character before the pattern is optional.

Grouping Commands Together

Commands may be grouped by writing either

```
(list)
```

or

```
{ list; }
```

The first of these executes the commands in a subshell. Builtin commands grouped into a (list) will not affect the current shell. The second form does not fork another shell so is slightly more efficient. Grouping commands together this way allows you to redirect their output as though they were one program:

```
{ printf " hello " ; printf " world\n" ; } > greeting
```

Note that "}" must follow a control operator (here, ";") so that it is recognized as a reserved word and not as another command argument.

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 a local command. This should appear as the first statement of a function, and the syntax is

```
local [variable | -] ...
```

Local is implemented as a builtin command.

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 you make the variable x 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 any shell options 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 currently executing function. Return is implemented as a builtin command.

Variables and Parameters

The shell maintains a set of parameters. A parameter denoted by a name is called a variable. When starting up, the shell turns all the environment variables into shell variables. New variables can be set using the form

```
name=value
```

Variables set by the user must have a name consisting solely of alphabetics, numerics, and underscores - the first of which must not be numeric. A parameter can also be denoted by a number or a special character as explained below.

Positional Parameters

A positional parameter is a parameter denoted by a number (n > 0). The shell sets these initially to the values of its command line arguments that follow the name of the shell script. The **set** builtin can also be used to set or reset them.

Special Parameters

A special parameter is a parameter denoted by one of the following special characters. The value of the parameter is listed next to its character.

*	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 builtin 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.
0 (Zero.)	Expands to the name of the shell or shell script.

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 set **`-f`** is in effect).
4. Quote Removal.

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

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 quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, are 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. Pathname expansion is not performed on the results of the expansion.
2. Field splitting is not performed on the results of the expansion, with the exception of @.

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. 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 Alternative 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.

${#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 * 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 (“backquoted” version):

```
`command`
```

The shell expands the command substitution by executing command in a subshell environment and replacing the command substitution with the standard output of the command, removing sequences of one or more ⟨newline⟩s at the end of the substitution. (Embedded ⟨newline⟩s before the end of the output are not removed; however, during field splitting, they may be translated into ⟨space⟩s, depending on the value of `IFS` and quoting that is in effect.)

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, and quote removal.

Next, the shell treats this as an arithmetic expression and substitutes the value of the expression.

White Space Splitting (Field Splitting)

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.

The shell treats each character of the `IFS` as a delimiter and uses the delimiters to split the results of parameter expansion and command substitution into fields.

Pathname Expansion (File Name Generation)

Unless the **`-f`** flag 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 both Pathname Expansion 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 range of characters may be specified using a minus sign. The character class may be complemented by making an exclamation point the first character of the character class.

To include a “]” in a character class, make it the first character listed (after the “!”, if any). To include a minus sign, make it the first or last character listed.

Builtins

This section lists the builtin commands which are builtin because they need to perform some operation that can’t be performed by a separate process. In addition to these, there are several other commands that may be builtin for efficiency (e.g. `printf`(1), `echo`(1), `test`(1), etc).

:

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

. file The commands in the specified file are read and executed by the shell.

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** builtin prints the names and values of all defined aliases (see **unalias**).

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

command [**-p**] [**-v**] [**-V**] *command* [*arg ...*]
Execute the specified command but ignore shell functions when searching for it. (This is useful when you have a shell function with the same name as a builtin command.)

-p search for command using a `PATH` that guarantees to find all the standard utilities.

-V Do not execute the command but search for the command and print the resolution of the command search. This is the same as the type builtin.

-v Do not execute the command but search for the command and print the absolute pathname of utilities, the name for builtins or the expansion of aliases.

cd −

cd [**-LP**] [*directory*]
Switch to the specified directory (default `HOME`). If an entry for `CDPATH` appears in the environment of the **cd** command or the shell variable `CDPATH` is set and the directory name does not begin with a slash, then the directories listed in `CDPATH` will be searched for the specified directory. The format of `CDPATH` is the same as that of `PATH`. If a single dash is specified as the argument, it will be replaced by the value of `OLDPWD`. The **cd** command will print out the name of the directory that it actually switched to if this is different from the name that the user gave. These may be different either because the `CDPATH` mechanism was used or because the argument is a single dash. The **-P** option causes the physical directory structure to be used, that is, all symbolic links are resolved to their respective values. The **-L** option turns off the effect of any preceding **-P** options.

echo [**-n**] *args...*
Print the arguments on the standard output, separated by spaces. Unless the **-n** option is present, a newline is output following the arguments.

If any of the following sequences of characters is encountered during output, the sequence is not output. Instead, the specified action is performed:

`\b` A backspace character is output.

`\c` Subsequent output is suppressed. This is normally used at the end of the last argument to suppress the trailing newline that **echo** would otherwise output.

`\e` Outputs an escape character (ESC).

`\f` Output a form feed.

`\n` Output a newline character.

`\r` Output a carriage return.

`\t` Output a (horizontal) tab character.

`\v` Output a vertical tab.

`\0`*digits*
Output the character whose value is given by zero to three octal digits. If there are zero digits, a nul character is output.

`\\` Output a backslash.

All other backslash sequences elicit undefined behaviour.

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 builtin 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 the exit status of the preceding command is used.

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 it is exported by writing

```
export name=value
```

With no arguments the export command lists the names of all exported variables. With the **-p** option specified the output will be formatted suitably for non-interactive use.

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

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

fc **-s** [*old=new*] [*first*]
The **fc** builtin 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:

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

-number
A negative decimal number representing the command that was executed number 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 environment variables affect the execution of fc:

FCEDIT Name of the editor to use.

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, not to be confused with the *Bell Labs* -derived getopt(1).

The first argument should be a series of letters, each of which may be optionally followed by a colon to indicate that the option requires an argument. The variable specified is set to the parsed option.

The **getopts** command deprecates the older getopt(1) utility due to its handling of arguments containing whitespace.

The **getopts** builtin may be used to obtain options and their arguments from a list of parameters. When invoked, **getopts** places the value of the next option from the option string in the list in the shell variable specified by *var* and its index in the shell variable OPTIND. When the shell is invoked, OPTIND is initialized to 1. For each option that requires an argument, the **getopts** builtin will place it in the shell variable OPTARG. If an option is not allowed for in the *optstring*, then OPTARG will be unset.

optstring is a string of recognized option letters (see `getopt`(3)). If a letter is followed by a colon, the option is expected to have an argument which may or may not be separated from it by white space. If an option character is not found where expected, **getopts** will set the variable *var* to a "?"; **getopts** will then unset `OPTARG` and write output to standard error. By specifying a colon as the first character of *optstring* all errors will be ignored.

After the last option **getopts** will return a non-zero value and set *var* to "?".

The following code fragment shows how one might process the arguments for a command that can take the options `[a]` and `[b]`, and the option `[c]`, which requires an argument.

```
while getopts abc: f
do
        case $f in
        a | b)  flag=$f;;
        c)      carg=$OPTARG;;
        \?)     echo $USAGE; exit 1;;
        esac
done
shift `expr $OPTIND - 1`
```

This code will accept any of the following as equivalent:

```
cmd -acarg file file
cmd -a -c arg file file
cmd -carg -a file file
cmd -a -carg -- file file
```

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. Entries which have not been looked at since the last **cd** command are marked with an asterisk; it is possible for these entries to be invalid.

With arguments, the **hash** command removes the specified commands from the hash table (unless they are functions) and then locates them. 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.

pwd [**-LP**]

builtin 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 builtin version of **pwd** will continue to print the old name for the directory. The **-P** option causes the physical value of the current working directory to be shown, that is, all symbolic links are resolved to their respective values. The **-L** option turns off the effect of any preceding **-P** options.

read [**-p** *prompt*] [**-r**] *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 word splitting above, and the pieces are assigned to the variables in order. At least one variable must be specified. 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. The **read** builtin will indicate success unless EOF is encountered on input, in which case failure is returned.

By default, unless the **-r** option is specified, the backslash "\" acts as an escape character, causing the following character to be treated literally. If a backslash is followed by a newline, the backslash and the newline will be deleted.

readonly *name ...*

readonly **-p**

The specified names are marked as read only, so that they cannot be subsequently modified or unset. The shell allows the value of a variable to be set at the same time it is marked read only by writing

```
readonly name=value
```

With no arguments the readonly command lists the names of all read only variables. With the **-p** option specified the output will be formatted suitably for non-interactive use.

printf *format* [*arguments ...*]

printf formats and prints its arguments, after the first, under control of the *format*. The *format* is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive *argument*.

The *arguments* after the first are treated as strings if the corresponding format is either **b**, **c** or **s**; otherwise it is evaluated as a C constant, with the following extensions:

- A leading plus or minus sign is allowed.
- If the leading character is a single or double quote, the value is the ASCII code of the next character.

The format string is reused as often as necessary to satisfy the *arguments*. Any extra format specifications are evaluated with zero or the null string.

Character escape sequences are in backslash notation as defined in ANSI X3.159-1989 ("ANSI C89"). The characters and their meanings are as follows:

\a Write a <bell> character.

\b Write a <backspace> character.

\e Write an <escape> (ESC) character.

\f Write a <form-feed> character.

\n Write a <new-line> character.

\r Write a <carriage return> character.

\t Write a <tab> character.

\v Write a <vertical tab> character.

**** Write a backslash character.

*****num* Write an 8−bit character whose ASCII value is the 1−, 2−, or 3−digit octal number *num*.

Each format specification is introduced by the percent character (''%''). The remainder of the format specification includes, in the following order:

Zero or more of the following flags:

A '#' character specifying that the value should be printed in an ''alternative form''. For **b**, **c**, **d**, and **s** formats, this option has no effect. For the **o** format the precision of the number is increased to force the first character of the output string to a zero.

For the **x** (**X**) format, a non-zero result has the string 0x (0X) prepended to it. For **e**, **E**, **f**, **g**, and **G** formats, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point only appears in the results of those formats if a digit follows the decimal point). For **g** and **G** formats, trailing zeros are not removed from the result as they would otherwise be.

- A minus sign '–' which specifies *left adjustment* of the output in the indicated field;

+ A '+' character specifying that there should always be a sign placed before the number when using signed formats.

' ' A space specifying that a blank should be left before a positive number for a signed format. A '+' overrides a space if both are used;

0 A zero '0' character indicating that zero-padding should be used rather than blank-padding. A '–' overrides a '0' if both are used;

Field Width:

An optional digit string specifying a *field width*; if the output string has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width (note that a leading zero is a flag, but an embedded zero is part of a field width);

Precision:

An optional period, '**.**', followed by an optional digit string giving a *precision* which specifies the number of digits to appear after the decimal point, for **e** and **f** formats, or the maximum number of bytes to be printed from a string (**b** and **s** formats); if the digit string is missing, the precision is treated as zero;

Format:

A character which indicates the type of format to use (one of **diouxXfwEgGbcs**).

A field width or precision may be '*' instead of a digit string. In this case an *argument* supplies the field width or precision.

The format characters and their meanings are:

diouXx The *argument* is printed as a signed decimal (d or i), unsigned octal, unsigned decimal, or unsigned hexadecimal (X or x), respectively.

f The *argument* is printed in the style [–]ddd**.**ddd where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

eE The *argument* is printed in the style [–]d**.**ddd**e**±dd where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. An upper-case E is used for an 'E' format.

gG The *argument* is printed in style **f** or in style **e** (**E**) whichever gives full precision in minimum space.

b Characters from the string *argument* are printed with backslash-escape sequences expanded.
The following additional backslash-escape sequences are supported:

\c Causes **dash** to ignore any remaining characters in the string operand containing it, any remaining string operands, and any additional characters in the format operand.

\0*num* Write an 8–bit character whose ASCII value is the 1–, 2–, or 3–digit octal number *num*.

c The first character of *argument* is printed.

s Characters from the string *argument* are printed until the end is reached or until the number of bytes indicated by the precision specification is reached; if the precision is omitted, all characters in the string are printed.

% Print a '%'; no argument is used.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

set [{ **–options** | **+options** | **--** }] *arg* **...**

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

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

If options are given, it sets the specified option flags, or clears them as described in the section called **Argument List Processing**. As a special case, if the option is -o or +o and no argument is supplied, the shell prints the settings of all its options. If the option is -o, the settings are printed in a human-readable format; if the option is +o, the settings are printed in a format suitable for reinput to the shell to affect the same option settings.

The third use of the set command is to set the values of the shell's positional parameters to the specified args. To change the positional parameters without changing any options, use "--" as the first argument to set. If no args are present, the set command will clear all the positional parameters (equivalent to executing "shift $#".)

shift [*n*]

Shift the positional parameters n times. 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. If n is greater than the number of positional parameters, **shift** will issue an error message, and exit with return status 2.

test *expression*

[*expression*]

The **test** utility evaluates the expression and, if it evaluates to true, returns a zero (true) exit status; otherwise it returns 1 (false). If there is no expression, test also returns 1 (false).

All operators and flags are separate arguments to the **test** utility.

The following primaries are used to construct expression:

-b *file* True if *file* exists and is a block special file.

-c *file* True if *file* exists and is a character special file.

-d *file* True if *file* exists and is a directory.

-e *file* True if *file* exists (regardless of type).

-f *file* True if *file* exists and is a regular file.

-g *file* True if *file* exists and its set group ID flag is set.

-h *file* True if *file* exists and is a symbolic link.

-k *file* True if *file* exists and its sticky bit is set.

-n *string* True if the length of *string* is nonzero.

-p *file* True if *file* is a named pipe (FIFO).

-r *file* True if *file* exists and is readable.

-s *file* True if *file* exists and has a size greater than zero.

-t *file_descriptor*
True if the file whose file descriptor number is *file_descriptor* is open and is associated with a terminal.

-u *file* True if *file* exists and its set user ID flag is set.

-w *file* True if *file* exists and is writable. True indicates only that the write flag is on. The file is not writable on a read-only file system even if this test indicates true.

-x *file* True if *file* exists and is executable. True indicates only that the execute flag is on. If *file* is a directory, true indicates that *file* can be searched.

-z *string* True if the length of *string* is zero.

-L *file* True if *file* exists and is a symbolic link. This operator is retained for compatibility with previous versions of this program. Do not rely on its existence; use **-h** instead.

-O *file* True if *file* exists and its owner matches the effective user id of this process.

-G *file* True if *file* exists and its group matches the effective group id of this process.

-S *file* True if *file* exists and is a socket.

file1 **-nt** *file2*
True if *file1* and *file2* exist and *file1* is newer than *file2*.

file1 **-ot** *file2*
True if *file1* and *file2* exist and *file1* is older than *file2*.

file1 **-ef** *file2*
True if *file1* and *file2* exist and refer to the same file.

string True if *string* is not the null string.

s1 **=** *s2* True if the strings *s1* and *s2* are identical.

s1 **!=** *s2* True if the strings *s1* and *s2* are not identical.

s1 **<** *s2* True if string *s1* comes before *s2* based on the ASCII value of their characters.

s1 **>** *s2* True if string *s1* comes after *s2* based on the ASCII value of their characters.

n1 **-eq** *n2* True if the integers *n1* and *n2* are algebraically equal.

n1 **-ne** *n2* True if the integers *n1* and *n2* are not algebraically equal.

n1 **-gt** *n2* True if the integer *n1* is algebraically greater than the integer *n2*.

n1 **-ge** *n2* True if the integer *n1* is algebraically greater than or equal to the integer *n2*.

n1 **-lt** *n2* True if the integer *n1* is algebraically less than the integer *n2*.

n1 **-le** *n2* True if the integer *n1* is algebraically less than or equal to the integer *n2*.

These primaries can be combined with the following operators:

! *expression*
True if *expression* is false.

expression1 **-a** *expression2*
True if both *expression1* and *expression2* are true.

expression1 **-o** *expression2*
True if either *expression1* or *expression2* are true.

(*expression***)**
True if expression is true.

The **-a** operator has higher precedence than the **-o** operator.

times Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.

trap [*action signal ...*]
Cause the shell to parse and execute action when any of the specified signals are received. The signals are specified by signal number or as the name of the signal. If *signal* is 0 or EXIT, the action is executed when the shell exits. *action* may be empty (**''**), which causes the specified signals to be ignored. With *action* omitted or set to '-' the specified signals are set to their default action. When the shell forks off a subshell, it 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. **trap** without any arguments cause it to write a list of signals and their associated action to the standard output in a format that is suitable as an input to the shell that achieves the same trapping results.

Examples:

```
trap
```

List trapped signals and their corresponding action

```
trap '' INT QUIT tstp 30
```

Ignore signals INT QUIT TSTP USR1

```
trap date INT
```

Print date upon receiving signal INT

type [*name ...*]
Interpret each name as a command and print the resolution of the command search. Possible resolutions are: shell keyword, alias, 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 [**-H** | **-S**] [**-a** | **-tfdscmlpnv** [*value*]]
Inquire about or set the hard or soft limits on processes or set new limits. The choice between hard limit (which no process is allowed to violate, and which may not be raised once it has been lowered) and soft limit (which causes processes to be signaled but not necessarily killed, and which may be raised) is made with these flags:

-H set or inquire about hard limits

-S set or inquire about soft limits. If neither **-H** nor **-S** is specified, the soft limit is displayed or both limits are set. If both are specified, the last one wins.

The limit to be interrogated or set, then, is chosen by specifying any one of these flags:

-a show all the current limits

-t show or set the limit on CPU time (in seconds)

-f show or set the limit on the largest file that can be created (in 512-byte blocks)

-d show or set the limit on the data segment size of a process (in kilobytes)

-s show or set the limit on the stack size of a process (in kilobytes)

-c show or set the limit on the largest core dump size that can be produced (in 512-byte blocks)

-m show or set the limit on the total physical memory that can be in use by a process (in kilobytes)

-l show or set the limit on how much memory a process can lock with `mlock`(2) (in kilobytes)

-p show or set the limit on the number of processes this user can have at one time

-n show or set the limit on the number files a process can have open at once

-v show or set the limit on the total virtual memory that can be in use by a process (in kilobytes)

-r show or set the limit on the real-time scheduling priority of a process

If none of these is specified, it is the limit on file size that is shown or set. If value is specified, the limit is set to that number; otherwise the current limit is displayed.

Limits of an arbitrary process can be displayed or set using the `sysctl`(8) utility.

umask [*mask*]
Set the value of umask (see `umask`(2)) to the specified octal value. If the argument is omitted, the umask value is printed.

unalias [**-a**] [*name*]
If *name* is specified, the shell removes that alias. If **-a** is specified, all aliases are removed.

unset [**-fv**] *name* **...**
The specified variables and functions are unset and unexported. If **-f** or **-v** is specified, the corresponding function or variable is unset, respectively. If a given name corresponds to both a variable and a function, and no options are given, only the variable is unset.

wait [*job*]
Wait for the specified job to complete and return the exit status of the last process in the job. If the argument is omitted, wait for all jobs to complete and return an exit status of zero.

Command Line Editing

When **dash** is being used interactively from a terminal, the current command and the command history (see **fc** in **Builtins**) can be edited using vi-mode command-line editing. This mode uses commands, described below, similar to a subset of those described in the vi man page. The command `set -o vi` 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. It is similar to vi: typing ⟨ESC⟩ enters vi command mode. Hitting ⟨return⟩ while

in command mode will pass the line to the shell.

EXIT STATUS

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.

ENVIRONMENT

`HOME` Set automatically by `login`(1) from the user's login directory in the password file (`passwd`(4)). This environment variable also functions as the default argument for the cd builtin.

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

`CDPATH` The search path used with the cd builtin.

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

`MAILCHECK` The frequency in seconds that the shell checks for the arrival of mail in the files specified by the `MAILPATH` or the `MAIL` file. If set to 0, the check will occur at each prompt.

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

`PS1` The primary prompt string, which defaults to "$ ", unless you are the superuser, in which case it defaults to "# ".

`PS2` The secondary prompt string, which defaults to "> ".

`PS4` Output before each line when execution trace (set -x) is enabled, defaults to "+ ".

`IFS` Input Field Separators. This is normally set to ⟨space⟩, ⟨tab⟩, and ⟨newline⟩. See the **White Space Splitting** section for more details.

`TERM` The default terminal setting for the shell. This is inherited by children of the shell, and is used in the history editing modes.

`HISTSIZE` The number of lines in the history buffer for the shell.

`PWD` The logical value of the current working directory. This is set by the **`cd`** command.

`OLDPWD` The previous logical value of the current working directory. This is set by the **`cd`** command.

`PPID` The process ID of the parent process of the shell.

FILES

`$HOME/.profile`

`/etc/profile`

SEE ALSO

`csh`(1), `echo`(1), `getopt`(1), `ksh`(1), `login`(1), `printf`(1), `test`(1), `getopt`(3), `passwd`(5), `environ`(7), `sysctl`(8)

HISTORY

dash is a POSIX-compliant implementation of /bin/sh that aims to be as small as possible. **dash** is a direct descendant of the NetBSD version of ash (the Almquist SHell), ported to Linux in early 1997. It was renamed to **dash** in 2002.

BUGS

Setuid shell scripts should be avoided at all costs, as they are a significant security risk.

PS1, PS2, and PS4 should be subject to parameter expansion before being displayed.