

BASH(1)

BASH(1)

**NAME**

bash - GNU Bourne-Again Shell

**SYNOPSIS****bash** [options] [file]

[- Cut -]

**RESERVED WORDS**

Reserved words are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a simple command (see **SHELL GRAMMAR** below) or the third word of a **case** or **for** command:

```
! case do done elif else esac fi for function if in select then until while { } time [[ ]]
```

**SHELL GRAMMAR****Simple Commands**

A simple command is a sequence of optional variable assignments followed by **blank**-separated words and redirections, and terminated by a control operator. The first word specifies the command to be executed, and is passed as argument zero. The remaining words are passed as arguments to the invoked command.

The return value of a simple command is its exit status, or 128+n if the command is terminated by signal n.

**Pipelines**

A pipeline is a sequence of one or more commands separated by the character |. The format for a pipeline is:

```
[time [-p]] [ ! ] command [ | command2 ... ]
```

The standard output of command is connected via a pipe to the standard input of command2. This connection is performed before any redirections specified by the command (see **REDIRECTION** below).

The return status of a pipeline is the exit status of the last command, unless the **pipefail** option is enabled. If **pipefail** is enabled, the pipeline's return status is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands exit successfully. If the reserved word ! precedes a pipeline, the exit status of that pipeline is the logical negation of the exit status as described above. The shell waits for all commands in the pipeline to terminate before returning a value.

If the **time** reserved word precedes a pipeline, the elapsed as well as user and system time consumed by its execution are reported when the pipeline terminates. The **-p** option changes the output format to that specified by POSIX. The **TIMEFORMAT** variable may be set to a format string that specifies how the timing information should be displayed; see the description of **TIMEFORMAT** under **Shell Variables** below.

Each command in a pipeline is executed as a separate process (i.e., in a subshell).

**Lists**

A list is a sequence of one or more pipelines separated by one of the operators **;**, **&**, **&&**, or **||**, and optionally terminated by one of **;**, **&**, or **<newline>**.

Of these list operators, **&&** and **||** have equal precedence, followed by **;** and **&**, which have equal precedence.

A sequence of one or more newlines may appear in a list instead of a semicolon to delimit commands.

If a command is terminated by the control operator **&**, the shell executes the command in the background in a subshell. The shell does not wait for the command to finish, and the return status is 0. Commands separated by a **;** are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

The control operators **&&** and **||** denote AND lists and OR lists, respectively. An AND list has the form

```
command1 && command2
```

command2 is executed if, and only if, command1 returns an exit status of zero.

An OR list has the form

command1 || command2

command2 is executed if and only if command1 returns a non-zero exit status. The return status of AND and OR lists is the exit status of the last command executed in the list.

### Compound Commands

A compound command is one of the following:

(list) list is executed in a subshell environment (see **COMMAND EXECUTION ENVIRONMENT** below). Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes. The return status is the exit status of list.

```
{ list; }
```

list is simply executed in the current shell environment. list must be terminated with a new-line or semicolon. This is known as a group command. The return status is the exit status of list. Note that unlike the metacharacters ( and ), { and } are reserved words and must occur where a reserved word is permitted to be recognized. Since they do not cause a word break, they must be separated from list by whitespace.

((expression))  
The expression is evaluated according to the rules described below under **ARITHMETIC EVALUATION**. If the value of the expression is non-zero, the return status is 0; otherwise the return status is 1. This is exactly equivalent to **let "expression"**.

```
[[ expression ]]
```

Return a status of 0 or 1 depending on the evaluation of the conditional expression expression. Expressions are composed of the primaries described below under **CONDITIONAL EXPRESSIONS**. Word splitting and pathname expansion are not performed on the words between the [[ and ]]; tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal are performed. Conditional operators such as **-f** must be unquoted to be recognized as primaries.

When the == and != operators are used, the string to the right of the operator is considered a pattern and matched according to the rules described below under **Pattern Matching**. If the shell option **nocasematch** is enabled, the match is performed without regard to the case of alphabetic characters. The return value is 0 if the string matches (==) or does not match (!=) the pattern, and 1 otherwise. Any part of the pattern may be quoted to force it to be matched as a string.

An additional binary operator, =~, is available, with the same precedence as == and !=. When it is used, the string to the right of the operator is considered an extended regular expression and matched accordingly (as in [regex\(3\)](#)). The return value is 0 if the string matches the pattern, and 1 otherwise. If the regular expression is syntactically incorrect, the conditional expression's return value is 2. If the shell option **nocasematch** is enabled, the match is performed without regard to the case of alphabetic characters. Substrings matched by parenthesized subexpressions within the regular expression are saved in the array variable **BASH\_REMATCH**. The element of **BASH\_REMATCH** with index 0 is the portion of the string matching the entire regular expression. The element of **BASH\_REMATCH** with index n is the portion of the string matching the nth parenthesized subexpression.

Expressions may be combined using the following operators, listed in decreasing order of precedence:

```
( expression )
    Returns the value of expression. This may be used to override the normal precedence of operators.
! expression
    True if expression is false.
expression1 && expression2
    True if both expression1 and expression2 are true.
expression1 || expression2
    True if either expression1 or expression2 is true.
```

The && and || operators do not evaluate expression2 if the value of expression1 is sufficient to determine the return value of the entire conditional expression.

```
for name [ in word ] ; do list ; done
```

The list of words following **in** is expanded, generating a list of items. The variable name is set to each element of this list in turn, and list is executed each time. If the **in word** is omitted, the **for** command executes list once for each positional parameter that is set (see **PARAMETERS** below). The return status is the exit status of the last command that executes. If the expansion of the items following **in** results in an empty list, no commands are executed, and the return status is 0.

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
```

First, the arithmetic expression expr1 is evaluated according to the rules described below

under **ARITHMETIC EVALUATION**. The arithmetic expression expr2 is then evaluated repeatedly until it evaluates to zero. Each time expr2 evaluates to a non-zero value, list is executed and the arithmetic expression expr3 is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in list that is executed, or false if any of the expressions is invalid.

**select** name [ **in** word ] ; **do** list ; **done**

The list of words following **in** is expanded, generating a list of items. The set of expanded words is printed on the standard error, each preceded by a number. If the **in word** is omitted, the positional parameters are printed (see **PARAMETERS** below). The **PS3** prompt is then displayed and a line read from the standard input. If the line consists of a number corresponding to one of the displayed words, then the value of name is set to that word. If the line is empty, the words and prompt are displayed again. If EOF is read, the command completes. Any other value read causes name to be set to null. The line read is saved in the variable **REPLY**. The list is executed after each selection until a **break** command is executed. The exit status of **select** is the exit status of the last command executed in list, or zero if no commands were executed.

**case** word **in** [ ([ pattern [ | pattern ] ... ) list ;; ] ... **esac**

A **case** command first expands word, and tries to match it against each pattern in turn, using the same matching rules as for pathname expansion (see **Pathname Expansion** below). The word is expanded using tilde expansion, parameter and variable expansion, arithmetic substitution, command substitution, process substitution and quote removal. Each pattern examined is expanded using tilde expansion, parameter and variable expansion, arithmetic substitution, command substitution, and process substitution. If the shell option **nocasematch** is enabled, the match is performed without regard to the case of alphabetic characters. When a match is found, the corresponding list is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no pattern matches. Otherwise, it is the exit status of the last command executed in list.

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

The **if** list is executed. If its exit status is zero, the **then** list is executed. Otherwise, each **elif** list is executed in turn, and if its exit status is zero, the corresponding **then** list is executed and the command completes. Otherwise, the **else** list is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.

**while** list; **do** list; **done**

**until** list; **do** list; **done**

The **while** command continuously executes the **do** list as long as the last command in list returns an exit status of zero. The **until** command is identical to the **while** command, except that the test is negated; the **do** list is executed as long as the last command in list returns a non-zero exit status. The exit status of the **while** and **until** commands is the exit status of the last **do** list command executed, or zero if none was executed.

### Shell Function Definitions

A shell function is an object that is called like a simple command and executes a compound command with a new set of positional parameters. Shell functions are declared as follows:

[ **function** ] name ( ) compound-command [redirection]

This defines a function named name. The reserved word **function** is optional. If the **function** reserved word is supplied, the parentheses are optional. The body of the function is the compound command compound-command (see **Compound Commands** above). That command is usually a list of commands between { and }, but may be any command listed under **Compound Commands** above. compound-command is executed whenever name is specified as the name of a simple command. Any redirections (see **REDIRECTION** below) specified when a function is defined are performed when the function is executed. The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body. (See **FUNCTIONS** below.)

### PARAMETERS

A parameter is an entity that stores values. It can be a name, a number, or one of the special characters listed below under **Special Parameters**. A variable is a parameter denoted by a name. A variable has a value and zero or more attributes. Attributes are assigned using the **declare** builtin command (see **declare** below in **SHELL BUILTIN COMMANDS**).

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the **unset** builtin command (see **SHELL BUILTIN COMMANDS** below).

A variable may be assigned to by a statement of the form

```
name=[value]
```

If value is not given, the variable is assigned the null string. All values undergo tilde expansion,

parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (see **EXPANSION** below). If the variable has its **integer** attribute set, then value is evaluated as an arithmetic expression even if the `$((...))` expansion is not used (see **Arithmetic Expansion** below). Word splitting is not performed, with the exception of "\$@" as explained below under **Special Parameters**. Pathname expansion is not performed. Assignment statements may also appear as arguments to the **alias**, **declare**, **typeset**, **export**, **readonly**, and **local** builtin commands.

In the context where an assignment statement is assigning a value to a shell variable or array index, the += operator can be used to append to or add to the variable's previous value. When += is applied to a variable for which the integer attribute has been set, value is evaluated as an arithmetic expression and added to the variable's current value, which is also evaluated. When += is applied to an array variable using compound assignment (see **Arrays** below), the variable's value is not unset (as it is when using =), and new values are appended to the array beginning at one greater than the array's maximum index. When applied to a string-valued variable, value is expanded and appended to the variable's value.

### Positional Parameters

A positional parameter is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the **set** builtin command. Positional parameters may not be assigned to with assignment statements. The positional parameters are temporarily replaced when a shell function is executed (see **FUNCTIONS** below).

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces (see **EXPANSION** below).

### Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

- \* Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the **IFS** special variable. That is, "\$\*" is equivalent to "\$1<sub>q</sub>\$2<sub>q</sub>...", where <sub>q</sub> is the first character of the value of the **IFS** variable. If **IFS** is unset, the parameters are separated by spaces. If **IFS** is null, the parameters are joined without intervening separators.
- @ Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word. That is, "\$@" is equivalent to "\$1" "\$2" ... If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the original word, and the expansion of the last parameter is joined with the last part of the original word. When there are no positional parameters, "\$@" and @ expand to nothing (i.e., they are removed).
- # Expands to the number of positional parameters in decimal.
- ? Expands to the status of the most recently executed foreground pipeline.
- Expands to the current option flags as specified upon invocation, by the **set** builtin command, or those set by the shell itself (such as the **-i** option).
- \$ Expands to the process ID of the shell. In a () subshell, it expands to the process ID of the current shell, not the subshell.
- ! Expands to the process ID of the most recently executed background (asynchronous) command.
- 0 Expands to the name of the shell or shell script. This is set at shell initialization. If **bash** is invoked with a file of commands, \$0 is set to the name of that file. If **bash** is started with the **-c** option, then \$0 is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the file name used to invoke **bash**, as given by argument zero.
- \_ At shell startup, set to the absolute pathname used to invoke the shell or shell script being executed as passed in the environment or argument list. Subsequently, expands to the last argument to the previous command, after expansion. Also set to the full pathname used to invoke each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file currently being checked.