Shell Commands

This section describes the following shell commands, listed in alphabetical order:

alias  for..do..done  readonly
bg    function      return
builtin history      select..do..done
case..in..esac if..then..elif..then..else..fi set
cd    jobs          source
declare kill          trap
dirs  local          unalias
env   popd           unset
export pushd         until..do..done
fg    read           while..do..done
Shell Variables

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$name</td>
<td>Replaced by the value of name.</td>
</tr>
<tr>
<td>$(name)</td>
<td>Replaced by the value of name. This form is useful if the expression is immediately followed by an alphanumeric that would otherwise be interpreted as part of the variable name.</td>
</tr>
</tbody>
</table>

```bash
$ verb=sing
$ echo I like $verb
...assign a variable.
I like
$ echo I like ${verb}ing
...there's no variable "verbing".
I like singing
$ _
...now it works.
```
Shell command: **declare** [-ax] [listname]

If the named variable does not already exist, it is created. If an array name is not specified when -a is used, declare will display all currently defined arrays and their values. If the -x option is used, the variable is exported to subshells. declare writes its output in a format that can be used again as input commands. This is useful when you want to create a script that sets variables as they are set in your current environment.

**Figure 6-5**  Example of the **declare** shell command.

```bash
$ declare -a teamnames
$ teamnames[0]="Dallas Cowboys"
$ teamnames[1]="Washington Redskins"
```
**Shell Command:** read \{ variable \}+

`read` reads one line from standard input and then assigns successive words from the line to the specified variables. Any words that are left over are assigned to the last-named variable.

*Figure 6-8* Description of the `read` shell command.

```
$ cat script.sh
...list the script.
echo "Please enter your name: \c"
read name  # read just one variable.
echo your name is $name  # display the variable.
$ bash script.sh
...run the script.
Please enter your name: Graham Walker Glass
your name is Graham Walker Glass  ...whole line was read.
$ _

$ cat script.sh
...list the script.
echo "Please enter your name: \c"
read firstName lastName  # read two variables.
echo your first name is $firstName
echo your last name is $lastName
$ bash script.sh
...run the script.
Please enter your name: Graham Walker Glass
your first name is Graham  ...first word.
your last name is Walker Glass  ...the rest.
$ bash script.sh
...run it again.
Please enter your name: Graham
your first name is Graham  ...first word.
your last name is          ...only one.
$ _
```
Shell Command: `export` { `variable` }+

`export` marks the specified variables for export to the environment. If no variables are specified, a list of all the variables marked for export during the shell session is displayed.

Figure 6–9  Description of the `export` shell command.

Utility: `env` { `variable=value` }* [ `command` ]

`env` assigns values to specified environment variables, and then executes an optional command using the new environment. If variables or command are not specified, a list of the current environment is displayed.

Figure 6–10  Description of the `env` command.
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-</td>
<td>The current shell options assigned from the command line or by the built-in set command—discussed later.</td>
</tr>
<tr>
<td>$$</td>
<td>The process ID of this shell.</td>
</tr>
<tr>
<td>$!</td>
<td>The process ID of the last background command.</td>
</tr>
<tr>
<td>$#</td>
<td>The number of positional parameters.</td>
</tr>
<tr>
<td>$?</td>
<td>The exit value of the last command.</td>
</tr>
<tr>
<td>@</td>
<td>An individually quoted list of all the positional parameters.</td>
</tr>
<tr>
<td>_</td>
<td>The last parameter of the previous command.</td>
</tr>
<tr>
<td>$BASH</td>
<td>The full pathname of the Bash executable.</td>
</tr>
<tr>
<td>$BASH_ENV</td>
<td>Location of Bash’s startup file (default is ~/.bashrc).</td>
</tr>
<tr>
<td>$BASHVERSINFO</td>
<td>A read-only array of version information.</td>
</tr>
<tr>
<td>$BASH_VERSION</td>
<td>Version string.</td>
</tr>
</tbody>
</table>

*Figure 6–13*  Bash predefined variables. (Part 1 of 3)
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DIRSTACK</td>
<td>Array defining the directory stack (discussed later).</td>
</tr>
<tr>
<td>$ENV</td>
<td>If this variable is not set, the shell searches the user’s home directory for the “profile” startup file when a new login shell is created. If this variable is set, then every new shell invocation runs the script specified by ENV.</td>
</tr>
<tr>
<td>$EUID</td>
<td>Read-only value of effective user ID of user running Bash.</td>
</tr>
<tr>
<td>$HISTFILE</td>
<td>Location of file containing shell history (default ~/.bash_history).</td>
</tr>
<tr>
<td>$HISTFILESIZE</td>
<td>Maximum number of lines allowed in history file (default is 500).</td>
</tr>
<tr>
<td>$HISTSIZE</td>
<td>Maximum number of commands in history (default is 500).</td>
</tr>
<tr>
<td>$HOSTNAME</td>
<td>Hostname of machine where Bash is running.</td>
</tr>
<tr>
<td>$HOSTTYPE</td>
<td>Type of host where Bash is running.</td>
</tr>
<tr>
<td>$IFS</td>
<td>When the shell tokenizes a command line prior to its execution, it uses the characters in this variable as delimiters. IFS usually contains a space, a tab, and a newline character.</td>
</tr>
<tr>
<td>$LINES</td>
<td>Used by select to determine how to display the selections.</td>
</tr>
<tr>
<td>$MAILCHECK</td>
<td>How often (seconds) to check for new mail.</td>
</tr>
<tr>
<td>$OLDPWD</td>
<td>The previous working directory of the shell.</td>
</tr>
<tr>
<td>$OSTYPE</td>
<td>Operating system of machine where Bash is running.</td>
</tr>
<tr>
<td>$PPID</td>
<td>The process ID number of the shell’s parent.</td>
</tr>
<tr>
<td>$PPPID</td>
<td>Read-only process ID of the parent process of Bash.</td>
</tr>
<tr>
<td>$PS1</td>
<td>This contains the value of the command-line prompt, and is $ by default. To change the command-line prompt, simply set PS1 to a new value.</td>
</tr>
<tr>
<td>$PS2</td>
<td>This contains the value of the secondary command-line prompt that is displayed when more input is required by the shell, and is &gt; by default. To change the prompt, set PS2 to a new value.</td>
</tr>
<tr>
<td>$PS3</td>
<td>The prompt used by the select command, #? by default.</td>
</tr>
<tr>
<td>$PWD</td>
<td>The current working directory of the shell.</td>
</tr>
<tr>
<td>$RANDOM</td>
<td>A random integer.</td>
</tr>
<tr>
<td>$REPLY</td>
<td>Set by a select command.</td>
</tr>
</tbody>
</table>

Figure 6–13  Bash predefined variables. (Part 2 of 3)
Shell Command: **alias [-p] [word[=string]]**

If you alias a new command word equal to string, then when you type the command word the string will be used in its place (and any succeeding arguments will be appended to string) and the command will be evaluated. In the usage “alias word” any alias defined for word will be printed. Its simplest usage “alias” will print all defined aliases. If the `-p` argument is used, the aliases are printed in a format suitable for input to the shell (so if you’ve manually set up aliases you like, you can write them to a file to include in your .bashrc file).

**Figure 6–14** Description of the **alias** shell command.

```
$ alias dir="ls -aF"
$ dir
  ./ main2.c  p.reverse.c  reverse.h
  ./ main2.o  palindrome.c  reverse.old
$ dir *.c
  main2.c  p.reverse.c  palindrome.c
$ 
```
Shell Command: `history [-c] [n]`

Print out the shell’s current command history. If a numeric value `n` is specified, show only the last `n` entries in the history list. If “-c” is used, clear the history list.

**Figure 6–16** Description of the `history` shell command.

<table>
<thead>
<tr>
<th>Form</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!!</code></td>
<td>Replaced with the text of the last command.</td>
</tr>
<tr>
<td><code>!number</code></td>
<td>Replaced with command number <code>number</code> in the history list.</td>
</tr>
<tr>
<td><code>!-number</code></td>
<td>Replaced with the text of the command <code>number</code> commands back from the end of the list (<code>!-1</code> is equivalent to <code>!!</code>).</td>
</tr>
<tr>
<td><code>!prefix</code></td>
<td>Replaced with the text of the last command that started with <code>prefix</code>.</td>
</tr>
<tr>
<td><code>!?substring?</code></td>
<td>Replaced with the text of the last command that contained <code>substring</code>.</td>
</tr>
</tbody>
</table>

**Figure 6–17** Command re-execution metacharacters in Bash.
<table>
<thead>
<tr>
<th>Tilde sequence</th>
<th>Replaced by</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>$HOME</td>
</tr>
<tr>
<td>~user</td>
<td>home directory of user</td>
</tr>
<tr>
<td>~/pathname</td>
<td>$HOME/pathname</td>
</tr>
<tr>
<td>~+</td>
<td>SPWD (current working directory)</td>
</tr>
<tr>
<td>~--</td>
<td>SOLDPWD (previous working directory)</td>
</tr>
</tbody>
</table>

Figure 6–21  Tilde substitutions in Bash.
Figure 6-24  Syntax of an arithmetic operation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>

Figure 6-25  Arithmetic operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-27  Arithmetic conditional operators.
$ cat divisors.sh
#!/bin/bash
#
declare -i testval=20
declare -i count=2    # start at 2, 1 always works

while ((( $count <= $testval ))); do
    (( result = $testval % $count ))
    if ((( $result == 0 ))); then   # evenly divisible
        echo "$testval is evenly divisible by $count"
    fi
    (( count++ ))
done
$ bash divisors.sh
20 is evenly divisible by 2
20 is evenly divisible by 4
20 is evenly divisible by 5
20 is evenly divisible by 10
20 is evenly divisible by 20
$ _
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-a file</code></td>
<td>True if the file exists.</td>
</tr>
<tr>
<td><code>-b file</code></td>
<td>True if the file exists and is a block-oriented special file.</td>
</tr>
<tr>
<td><code>-c file</code></td>
<td>True if the file exists and is a character-oriented special file.</td>
</tr>
<tr>
<td><code>-d file</code></td>
<td>True if the file exists and is a directory.</td>
</tr>
<tr>
<td><code>-e file</code></td>
<td>True if the file exists.</td>
</tr>
<tr>
<td><code>-f file</code></td>
<td>True if the file exists and is a regular file.</td>
</tr>
<tr>
<td><code>-g file</code></td>
<td>True if the file exists and its “set group ID” bit is set.</td>
</tr>
<tr>
<td><code>-p file</code></td>
<td>True if the file exists and is a named pipe.</td>
</tr>
<tr>
<td><code>-r file</code></td>
<td>True if the file exists and is readable.</td>
</tr>
<tr>
<td><code>-s file</code></td>
<td>True if the file exists and has a size greater than zero.</td>
</tr>
<tr>
<td><code>-t file</code></td>
<td>True if the file descriptor is open and refers to the terminal.</td>
</tr>
<tr>
<td><code>-u file</code></td>
<td>True if the file exists and its “set user ID” bit is set.</td>
</tr>
<tr>
<td><code>-w file</code></td>
<td>True if the file is writable.</td>
</tr>
<tr>
<td><code>-x file</code></td>
<td>True if the file exists and is executable.</td>
</tr>
<tr>
<td><code>-O file</code></td>
<td>True if the file exists and is owned by the effective user ID of the user.</td>
</tr>
<tr>
<td><code>-G file</code></td>
<td>True if the file exists and is owned by the effective group ID of the user.</td>
</tr>
<tr>
<td><code>-L file</code></td>
<td>True if the file exists and is a symbolic link.</td>
</tr>
<tr>
<td><code>-N file</code></td>
<td>True if the file exists and has been modified since it was last read.</td>
</tr>
<tr>
<td><code>-S file</code></td>
<td>True if the file exists and is a socket.</td>
</tr>
<tr>
<td><code>file1 -nt file2</code></td>
<td>True if <code>file1</code> is newer than <code>file2</code>.</td>
</tr>
<tr>
<td><code>file1 -ot file2</code></td>
<td>True if <code>file1</code> is older than <code>file2</code>.</td>
</tr>
<tr>
<td><code>file1 -ef file2</code></td>
<td>True if <code>file1</code> and <code>file2</code> have the same device and inode numbers.</td>
</tr>
</tbody>
</table>

Figure 6–29  File-oriented conditional operators. (Part 2 of 2)
$ cat owner.sh
#!/bin/bash

if [ -o /etc/passwd ]; then
    echo "you are the owner of /etc/passwd."
else
    echo "you are NOT the owner of /etc/passwd."
fi
Shell command: `case`  

```bash
case word in
    pattern { | pattern }* ) commands ;;
    ...
esac
```

Execute the commands specified by `commands` when the value of `word` matches the pattern specified by `pattern`. The `)` indicates the end of the list of patterns to match. The `;;` is required to indicate the end of the commands to be executed.

**Figure 6-30** Description of the `case` shell command.

```bash
case ${teamname[${index}]} in
    "Dallas Cowboys") echo "Dallas, TX" ;;
    "Denver Broncos") echo "Denver, CO" ;;
    "New York Giants"|"New York Jets") echo "New York, NY";;
    ...
    *) echo "Unknown location" ;;
esac
```
#!/bin/bash

echo menu test program

stop=0                      # reset loop termination flag.
while test $stop -eq 0      # loop until done.
do
  cat << ENDOFMENU          # display menu.
    1 : print the date.
    2, 3: print the current working directory.
    4 : exit
  ENDOFMENU
  echo echo -n 'your choice? ' # prompt.
  read reply                 # read response.
  echo echo case $reply in   # process response.
    "1")
      date                   # display date.
    ;;
    "2"|"3")
      pwd                    # display working directory.
    ;;
    "4")
      stop=1                # set loop termination flag.
    ;;
    *)
      echo illegal choice   # error.
    ;;
esac
  done

Here's the output from the "menu.sh" script:

$ bash menu.sh
menu test program
    1 : print the date.
    2, 3: print the current working directory.
    4 : exit

your choice? 1

Thu May  5 07:09:13 CST 2005
    1 : print the date.
    2, 3: print the current working directory.
    4 : exit
Shell command: if
if test1; then
  commands1;
[elif test2; then
  commands2;]
[else commands3;]
fi

test1 is a conditional expression (discussed above), which, if true, causes the commands specified by commands1 to be executed. If test1 tests false, then if an “elif” structure is present, the next test, test2, is evaluated (“else if”). If test2 evaluates to true, then the commands in commands2 are executed. The “else” construct is used when you always want to run commands after a test evaluated as false.

Figure 6–31 Description of the if shell command.
Shell command: **for**

```
for name in word { word }*
do
  commands
done
```

Perform *commands* for each *word* in list with "$name" containing the value of the current *word*.

**Figure 6–32** Description of the *for* shell command.

---

Shell commands: **while/until**

```
while test
do
  commands
done

until test
do
  commands
done
```

In a *while* statement, perform *commands* as long as the expression *test* evaluates to true. In an *until* statement, perform *commands* as long as the expression *test* evaluates to false (i.e., until *test* is true).

**Figure 6–33** Description of the *while* and *until* shell commands.
$ cat until.sh       ...list the script.
x=1
until [ $x -gt 3 ]
do
echo x = $x
    (( x = $x + 1 ))
done
$ bash until.sh     ...execute the script.
x = 1
x = 2
x = 3
$ 

$ cat multi.sh       ...list the script.
if [ $# -lt 1 ]; then
    echo "Usage: multi number"
    exit
fi
x=1                # set outer loop value
while [ $x -le $1 ]  # outer loop
do
    y=1             # set inner loop value
    while [ $y -le $1 ]  # inner loop
        # generate one table entry
        (( entry = $x * $y ))
        echo -e -n "$entry\t"
        (( y = $y + 1 ))  # update inner loop count
done
    echo            # blank line
    (( x = $x + 1 ))  # update outer loop count
done
$ bash multi.sh 7  ...execute the script.
1 2 3 4 5 6 7
2 4 6 8 10 12 14
3 6 9 12 15 18 21
4 8 12 16 20 24 28
5 10 15 20 25 30 35
6 12 18 24 30 36 42
7 14 21 28 35 42 49
```bash
$ cat newmenu.sh  ...list the script.
  echo menu test program
  select reply in "date" "pwd" "pwd" "exit"
do
  case $reply in
    "date")
      date
      ;;
    "pwd")
      pwd
      ;;
    "exit")
      break
      ;;
  *)
      echo illegal choice
      ;;
  esac
done
$ sh newmenu.sh  ...execute the script.
  menu test program
  1) date
  2) pwd
  3) pwd
  4) exit
  #? 1
Fri May   6 21:49:33 CST 2005
  #? 5
  illegal choice
  #? 4
$ _
```

Figure 6-41  Description of the select shell command.
select name [ in {word }+ ]
do
list
done

Figure 6-41  Description of the select shell command.

Shell Command: fg [ %job ]

fg resumes the specified job as the foreground process. If no job is specified, the last-referenced job is resumed.

Figure 6-49  Description of the fg shell command.