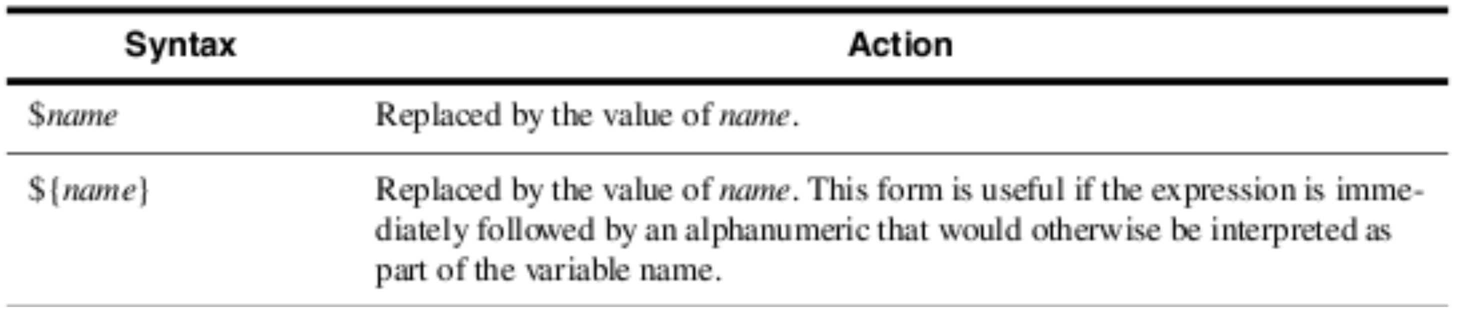
Shell Commands

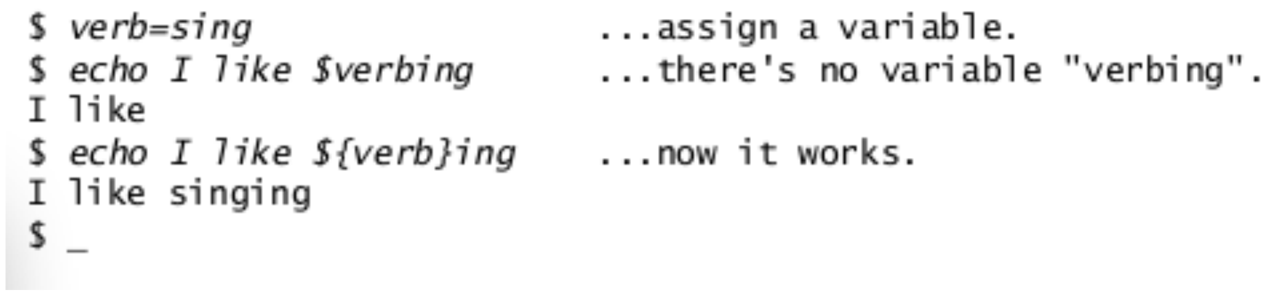
This section describes the following shell commands, listed in alphabetical order:

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



Syntax	Action
$*name*	Replaced by the value of *name*.
${*name*}	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.



```
$ verb=sing                    ...assign a variable.
$ echo I like $verbing         ...there's no variable "verbing".
I like
$ echo I like ${verb}ing       ...now it works.
I like singing
$ _
```

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.

```
$ declare -a teamnames
$ teamnames[0]="Dallas Cowboys"
$ teamnames[1]="Washington Redskins"
$ teamnames[2]="New York Giants"
```

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.

Name	Value
$-	The current shell options assigned from the command line or by the built-in set command—discussed later.
$$	The process ID of this shell.
$!	The process ID of the last background command.
$#	The number of positional parameters.
$?	The exit value of the last command.
$@	An individually quoted list of all the positional parameters.
$_	The last parameter of the previous command.
$BASH	The full pathname of the Bash executable.
$BASH_ENV	Location of Bash's startup file (default is ~/.bashrc).
$BASH_VERSINFO	A read-only array of version information.
$BASH_VERSION	Version string.

Figure 6–13 Bash predefined variables. (Part 1 of 3)

Name	Value
$DIRSTACK	Array defining the directory stack (discussed later).
$ENV	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.
$EUID	Read-only value of effective user ID of user running Bash.
$HISTFILE	Location of file containing shell history (default ~/.bash_history).
$HISTFILESIZE	Maximum number of lines allowed in history file (default is 500).
$HISTSIZE	Maximum number of commands in history (default is 500).
$HOSTNAME	Hostname of machine where Bash is running.
$HOSTTYPE	Type of host where Bash is running.
$IFS	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.
$LINES	Used by *select* to determine how to display the selections.
$MAILCHECK	How often (seconds) to check for new mail.
$OLDPWD	The previous working directory of the shell.
$OSTYPE	Operating system of machine where Bash is running.
$PPID	The process ID number of the shell's parent.
$PPID	Read-only process ID of the parent process of Bash.
$PS1	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.
$PS2	This contains the value of the secondary command-line prompt that is displayed when more input is required by the shell, and is > by default. To change the prompt, set PS2 to a new value.
$PS3	The prompt used by the *select* command, #? by default.
$PWD	The current working directory of the shell.
$RANDOM	A random integer.
$REPLY	Set by a *select* command.

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.

Form	Action
!!	Replaced with the text of the last command.
!*number*	Replaced with command number *number* in the history list.
!-*number*	Replaced with the text of the command *number* commands back from the end of the list (!-1 is equivalent to !!).
!*prefix*	Replaced with the text of the last command that started with *prefix*.
!?*substring*?	Replaced with the text of the last command that contained *substring*.

Figure 6–17 Command re-execution metacharacters in Bash.

Tilde sequence	Replaced by
~	$HOME
~*user*	home directory of *user*
~/*pathname*	$HOME/*pathname*
~+	$PWD (current working directory)
~-	$OLDPWD (previous working directory)

Figure 6–21 Tilde substitutions in Bash.

```
(( operation ))
```

Figure 6–24 Syntax of an arithmetic operation.

+ -	Addition, subtraction.
++ --	Increment, decrement.
* / %	Multiplication, division, remainder.
**	Exponentiation.

Figure 6–25 Arithmetic operators.

<= >= < >	Less than or equal to, greater than or equal to, less than, greater than comparisons.
== !=	Equal, not equal.
!	Logical NOT.
&&	Logical AND.
||	Logical OR.

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
$ _
```

-a *file*	True if the file exists.
-b *file*	True if the file exists and is a block-oriented special file.
-c *file*	True if the file exists and is a character-oriented special file.

-d *file*	True if the file exists and is a directory.
-e *file*	True if the file exists.
-f *file*	True if the file exists and is a regular file.
-g *file*	True if the file exists and its “set group ID” bit is set.
-p *file*	True if the file exists and is a named pipe.
-r *file*	True if the file exists and is readable.
-s *file*	True if the file exists and has a size greater than zero.
-t *fd*	True if the file descriptor is open and refers to the terminal.
-u *file*	True if the file exists and its “set user ID” bit is set.
-w *file*	True if the file is writable.
-x *file*	True if the file exists and is executable.
-O *file*	True if the file exists and is owned by the effective user ID of the user.
-G *file*	True if the file exists and is owned by the effective group ID of the user.
-L *file*	True if the file exists and is a symbolic link.
-N *file*	True if the file exists and has been modified since it was last read.
-S *file*	True if the file exists and is a socket.
file1 –nt *file2*	True if *file1* is newer than *file2*.
file1 –ot *file2*	True if *file1* is older than *file2*.
file1 –ef *file2*	True if *file1* and *file2* have the same device and inode numbers.

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**

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.

```
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
 case $reply in              # process response.
   "1")
     date                    # display date.
     ;;
   "2"|"3")
     pwd                     # display working directory.
     ;;
   "4")
     stop=1                  # set loop termination flag.
     ;;
   *)                        # default.
     echo illegal choice     # error.
     ;;
 esac
 echo
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 ]
  do                          # 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
```

select *name* [**in** {*word* }+]
do
  list
done

Figure 6–41 Description of the *select* shell command.

```
$ 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
$ _
```

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.