Table of Contents

COMMAND EXECUTION

The primary prompt ($**PS1** - default **$** or # for super-users) is displayed whenever the Bourne shell is ready to read a command.
The secondary prompt ($**PS2** - default **>**) is displayed when the command is incomplete.

Command Execution Format

command1 **;** *command2*	execute *command1* followed by *command2*
command **&**	execute *command* asynchronously in the background
command1 **|** *command2*	pass the standard output of *command1* to standard input of *command2*
command1 **&&** *command2*	execute *command2* if and only if *command1* returns zero (successful) exit status eg. /sbin/lsmod | grep -q ^ipw2200 && { rmmod ipw2200; modprobe ipw2200; }
command1 **||** *command2*	execute *command2* if and only if *command1* returns non-zero (unsuccessful) exit status
command ****	continue *command* onto the next line. '\' must be the last char.
if { *command* **; }**	execute *command* in the current shell. eg. `if { cat /etc/motd &>/dev/null ; } ; then`
if (*command***)**	execute *command* in a subshell. eg. `if (cat /etc/motd &>/dev/null) ; then`

REDIRECTING INPUT/OUTPUT

The Bourne shell provides a number of operators that can be used to manipulate command input/output, and files.

I/O Redirection Operators

<*file*	redirect standard input from *file*
>*file*	redirect standard output to *file*. Create *file* if non-existent, else overwrite.
>>*file*	append standard output to *file*; Create *file* if non-existent.
<&**-**	close standard input
>&**-**	close standard output
<&*n*	redirect standard input from file descriptor *n*
>&*n*	redirect standard output to file descriptor *n*
n<*file*	redirect file descriptor *n* from *file*
&>*file*	redirect file both *stdout(1)* and *stderr(2)* file descriptors to *file*
n>*file*	redirect file descriptor *n* to *file*. Create *file* if non-existent, else overwrite.
n>>*file*	redirect file descriptor *n* to *file*. Create *file* if non-existent.
n<&*m*	redirect file descriptor *n* from file descriptor *m*
n>&*m*	redirect file descriptor *n* to file descriptor *m*
n<<*x*	redirect to file descriptor *n* until *x* is read
n<<**-***x*	same as *n*<<*x*, except ignore leading tabs
n<&**-**	close file descriptor *n* for standard input
n>&**-**	close file descriptor *n* for standard output
&>*n*	redirect standard output and standard error to file descriptor *n*

FILENAME SUBSTITUTION

File name substitution is a feature which allows special characters and patterns to substituted with file names in the current directory, or arguments to the **`case`** and **`test`** commands.

Pattern-Matching Characters/Patterns

?	match any single character
*	match zero or more characters, including null
[**abc**]	match any characters between the brackets
[**x-z**]	match any characters in the range **x** to **z**
[**a-ce-g**]	match any characters in the range **a** to **c** or **e** to **g**
[**!abc**]	match any characters not between the brackets
[**!x-z**]	match any characters not in the range **x** to **z**
.	strings starting with **.** must be explicitly matched

VARIABLES

Variables are used by the Bourne shell to store values. Variable names can begin with an alphabetic or underscore character, followed by one or more alphanumeric or underscore characters. Other variable names that contain only digits or special characters are reserved for special variables (called *parameters*) set directly by the Bourne shell.

Variable Assignment Format

variable=, *variable*=""	declare *variable* and set it to null
variable=*value*	assign *value* to *variable*
variable=*value command*	Set the variable with value and run the command

VARIABLE SUBSTITUTION

Variable values can be accessed and manipulated using variable expansion. Basic expansion is done by preceding the variable name with the **$** character. Other types of expansion use default or alternate values, assign default or alternate values, and more.

Variable Expansion Format

$*variable*	value of *variable*
${*variable*}	value of *variable*
${#*variable*}	numeric length(number of chars.) of value of *variable*
${*variable*:-*word*}	value of *variable* if set and not null, else print *word* . If **:** is omitted, *variable* is only checked if it is set. eg. `echo ${USER:-halo} ; echo $USER`
${*variable*:+*word*}	value of *word* if *variable* is set and not null, else nothing is substituted. If **:** is omitted, *variable* is only checked if it is set. eg. `echo ${USER:+halo} ; echo $USER`
${*variable*:=*word*}	value of *variable* if set and not null, else variable is set to *word*, then expanded. If **:** is omitted, *variable* is only checked if it is set.
${*variable*:?}	value of *variable* if set and not null, else print "**variable: parameter null or not set**". If **:** is omitted, *variable* is only checked if it is set.
${*variable*:?*word*}	value of *variable* if set and not null, else print value of *word* and exit. If **:** is omitted, *variable* is only checked if it is set.

SPECIAL PARAMETERS

Some special parameters are automatically set by the Bourne shell, and usually cannot be directly set or modified. The $n can be modified by the command `set aaa bbb ccc ...`

$*n*	Positional parameter *n* max. n=9 ($0 is the name the shell script)
${nn}	Positional parameter *nn* (for *nn*>9)
$#	Number of positional parameters (not including the script program)
$@, $*	All positional parameters
"$@"	Same as "$**1**" "$**2**" . . . "$*n*"
"$*"	Same as "$**1***c*$**2***c* . . . $*n*" *c* = content of **$IFS** (default is space)
$**?**	Exit status of the last command
$**$**	Process ID of the current shell
$**-**	Current options in effect
$**!**	Process ID of the last background command
$**is**	Name of the curent shell (in this case 'bash')

The `shift` command:

The command shift moves the assignment of the positional parameters to the left.
eg. `script1 aaa bbb ccc ddd`
(inside the script *`script1`*)

```
                                    ($1  $2  $3)
    echo $1 $2 $3  -------> result   aaa bbb ccc

    shift                           ($1  $2  $3)
    echo $1 $2 $3  -------> result   bbb ccc ddd
```

SPECIAL VARIABLES

There are a number of variables provided by the Bourne shell that allow you to customize your working environment. Some are automatically set by the shell, some have a default value if not set, while others have no value unless specifically set.

Special Variables (keywords)

CDPATH	search path for `cd` when not given a full pathname; multiple pathnames are separated with a colon (no default)
HOME	default argument for the `cd` command; contains the path of home directory
IFS	internal field separator (default is space, tab, or newline)
LANG	contains the name of the current locale
MAIL	name of mail file to use if **MAILPATH** not set
MAILCHECK	specifies how often to check for mail in $**MAIL** or $**MAILPATH**. If set to `0`, mail is checked before each prompt. (default **600** seconds)
MAILPATH	contains a list of colon-separated file names that are checked for mail. File names can be followed by a "**%**" and a message to display each time new mail is received in the mail file. (no default)
PATH	search path for commands; multiple pathnames are separated with a colon (default `/bin:/usr/bin:`)
PS1	primary prompt string (default: `$, #`)
PS2	secondary prompt string (default: '>')

SHACCT	Contains the name of the accounting file that contains accounting records for user shell procedures.
SHELL	Pathname of the shell
TERM	Specifies your terminal type: `xterm`=in X-window environment `screen`='konsole' in 'screen' mode `linux`=from `tty` virtual terminal `dumb`=system(eg. shell scripts executed from cron)

JOB CONTROL

Job control is a process manipulation feature found in the Bourne shell when invoked as **jsh**. It allows programs to be stopped and restarted, moved between the foreground and background, their processing status to be displayed, and more. When a program is run in the background, a job number and process id are returned.

Job Control Commands

bg [*%n*]	Resume current or stopped job *n* in the background
fg [*%n*]	Move current or background job *n* into foreground
jobs [*option*]	Display status of all jobs `-n` Status since last job change `-r` List of running jobs only `-s` List stopped jobs only `-l` display status of all jobs and their process ID's `-p` display process ID's of all jobs
jobs `-x` *command*	Replace job *n* in *command* with corresponding process group id, then execute *command*
kill [**-***signal*] *%n*	Send specified signal to job *n* (default **15**)
stop *%n*	Stop job *n*
stty [**-**]**tostop**	Allow/prevent background jobs from generating output
suspend	Suspend execution of current shell
wait	Wait for all background jobs to complete
wait *%n*	Wait for background job *n* to complete
Ctl-z	Stop current job
disown [*option*] [*%n*]	Disown the last activated(+) background job or job *%n*. A disowned job will not die when shell dies. `init` will be its father. `-a` Disown all the background jobs `-r` Disown only the running jobs `-h` Disown active job (+)from shell only when shell is closed:

Job Name Format

`%%, %+`	current job
%n	job *n*
%-	previous job
%string	job whose name begins with *string*
%?string	job that matches part or all of *string*

QUOTING

Quotes are used when assigning values containing whitespace or special characters, to delimit variables, and to assign command output. They also improve readability by separating arguments from commands.

`'...'`	remove the special meaning of enclosed characters except `'`
`"..."`	remove the special meaning of enclosed characters except **$**, `'`, and `\`
`\`*c*	remove the special meaning of character *c*
`` `command` ``	replace with the standard output of *command* . Same as `$(`*command*`)`
Meta-characters in bash:	In the open: `$ & ; ( ) { } [ ] * ? ! < > \` In Double Quotes " ": `$ ! \`

OPTIONS

The Bourne shell has a number of options that specify your environment and control execution. They can be enabled/disabled with the **`set`** command or on the **sh** or **`jsh`** command line. Some options are only available on invocation.

Enabling/Disabling Options

`sh` `[-/+`*options*`]`	enable/disable the specified options
`jsh` `[-/+`*options*`]`	enable/disable the specified options; enable job control (see **JOB CONTROL** section)
`set` `[-/+`*options*`]`	enable/disable the specified options (see also **set**)

List of Options

`-a`	automatically export variables that are defined
`-c` *commands*	read and execute *commands* (w/**sh** only)
`-e`	exit if a command fails
`-f`	disable file name expansion
`-h`	remember locations of functions on definition instead of on execution (see also **hash**)
`-i`	execute in interactive mode (w/**sh** only)
`-k`	put variable assignment arguments in environment
`-n`	read commands without executing them
`-p`	do not set effective ids to real ids
`-r`	run a restricted shell (w/**sh** only)
`-s`	read commands from standard input (w/**sh** only)
`-t`	exit after reading and executing one command
`-u`	return error on substitution of unset variables
`-v`	display input lines as they are read
`-x`	display commands and arguments as executed

CONDITIONAL EXPRESSIONS

The **test** and `[...]` commands are used to evaluate conditional expressions with file attributes, strings, and integers. The basic format is:

```
test expression
   or
   [ expression ]
```

Where *expression* is the condition you are evaluating. There must be whitespace after the opening bracket, and before the closing bracket. Whitespace must also separate the expression arguments and operators. If the expression evaluates to true, then a zero exit status is returned, otherwise the expression evaluates to false and a non-zero exit status is returned.

Test File Operators

-a *file*	True if file exists.
-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.
-f *file*	True if file exists and is a regular file.
-g *file*	True if file exists and is set-group-id.
-h *file*	True if file exists and is a symbolic link.
-k *file*	True if file exists and its ``sticky'' bit is set.
-p *file*	True if file exists and 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 *fd*	True if file descriptor *fd* is open and refers to a terminal.
-u *file*	True if file exists and its SUID bit is set.
-w *file*	True if file exists and is writable.
-x *file*	True if file exists and is executable.
-O *file*	True if file exists and is owned by the effective UID.
-G *file*	True if file exists and is owned by the effective GID.
-L *file*	True if file exists and is a symbolic link.
-S *file*	True if file exists and is a socket.
-N *file*	True if file exists and has been modified since it was last read.
file1 **-nt** *file2*	True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.
file1 **-ot** *file2*	True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

file1 **-ef** *file2* True if *file1* and *file2* refer to the same device and inode numbers.

-o *optname* True if shell option *optname* is enabled.
See the list of options under the description of the -o option to the set builtin below.

Test String Operators

-n *string*	True if length of *string* is not zero
-z *string*	True if length of *string* is zero
string	True if *string* is not set to null
string1 = *string2*	True if *string1* is equal to *string2*
string1 == *string2*	" " " " " " " "
string1 != *string2*	True if *string1* is not equal to *string2*
string1 < *string2*	True if *string1* sorts before *string2* lexicographically in the current locale.
string1 > *string2*	True if *string1* sorts after *string2* lexicographically in the current locale.
string = *pattern*	True if *string* matches *pattern*
string != *pattern*	True if *string* does not match *pattern*

Test Integer Operators

exp1 **-eq** *exp2*	True if *exp1* is equal to *exp2*	eg. `[ "$#" -eq 4 ]`
exp1 **-ne** *exp2*	True if *exp1* is not equal to *exp2*	eg. `test "$#" -ne 3`
exp1 **-le** *exp2*	True if *exp1* is less than or equal to *exp2*	
exp1 **-lt** *exp2*	True if *exp1* is less than *exp2*	
exp1 **-ge** *exp2*	True if *exp1* is greater than or equal to *exp2*	
exp1 **-gt** *exp2*	True if *exp1* is greater than *exp2*	

Other test Operators

! *exp*	True if the given expression is false eg. `[ ! -r /etc/motd ]`
exp1 **-a** *exp2*	True if both *exp1* and *exp2* evaluate to true (see example below)
exp1 **-o** *exp2*	True if either *exp1* or *exp2* evaluate to true
\(*exp* \)	True if *exp* is true; used to group expressions (\ used to escape parentheses) Use space

eg : `[ "$A" = "$B" -a \( "$C" = "$D" -a "$E" = "$F" \) ]`
`^ ^ ^ ^ ^`

Note: always use a space between the [ ] \(\) and the expressions like seen in the above example pointed by '^'.

Example of logical AND of commands

```
if ( cat /etc/motd &>/dev/null && cat /etc/fstab
&>/dev/null ) ; then echo "all OK" ; fi
```

Example of logical OR of commands

```
if ( cat /etc/motd &>/dev/null || cat /etc/fstab
&>/dev/null ) ; then echo "all OK" ; fi
```

Arithmetic Operators (let)

`let` can also be replaced by `$[...]` eg. `B=$[$A/4]`

var++	Variable increment	eg. `let A++`	increment $A
var--	Variable decrement	eg. `let A--`	decrement $A
+ -	Unary minus and plus	eg. `let B=-$A`	B=B-A
**	Exponentiation	eg. `let B="$A**2"`	B=A^2
* /	Multiplication, division,	eg. `let B="$A*3"`	B=Ax3
%	Division remainder	eg. `let B=$A%3`	B=A/3
+ -	addition, subtraction	eg. `let B=$A+2`	B=A+2
<< >>	Bitwise shifting	eg. `let B="$A<<3"`	B=A left shift 3 bits
&	bitwise AND	eg. `let B="$A&14"`	B=A AND 14(bin)
^	bitwise exclusive OR	eg. `let B="$A^14"`	B=A XOR 14(bin)
|	bitwise OR	eg. `let B="$A|14"`	B=A OR 14(bin)
(...)	Expression grouping	eg. `let B="($A-5)*3"`	B=(A-5)x3

Assignment operations (the result goes into the original variable)

=*n*	Change of value of Variable to n	eg. `A=50`	
+=*n*	Add value of n to Variable	eg. `let A+=1`	A=A+1
-=*n*	Substract value of n from Variable	eg. `let A-=1`	A=A-1
*=*n*	Multiply Variable by n (inside " ")	eg. `let` **`"A*=3"`**	A=Ax3
/=*n*	Divide Variable by n	eg. `let A/=4`	A=A/4
%=*n*	Remainder of Variable divided by n	eg. `let A%=3`	A=Remainder A/3
<<=	Bitwise shift to the left (inside " ")	eg. `let` **`"A<<=3"`**	A=A left shift 3 bits
>>=	Bitwise shift to the right (inside " ")	eg. `let` **`"A>>=3"`**	A=A right shift 3 bits
&=	Bitwise AND (inside " ")	eg. `let` **`"A&=14"`**	A=A and 14 (Bitwise)
^=	Bitwise exclusive OR	eg. `let A^=14`	A=A XOR 14(bin)
|=	Bitwise OR (inside " ")	eg. `let` **`"A|=14"`**	A=A OR 14(bin)

Sample Integer Expression Assignments with `let`

Assignment	***Value***	
let x=	***$x***	
x++	***x=x+1***	
x--	***x=x-1***	
`1+4`	`5`	
`"1 + 4"`	`5`	
`"(2+3) * 5"`	`25`	`(5 *5)` expression in parentheses is processed first
`"2 + 3 * 5"`	`17`	`2 + (3 *5)` (* is processed first)
`"17 / 3"`	`5`	
`"17 % 3"`	`2`	`17 / 3 = 5` remainder = 2
`"1<<4"`	`16`	`00000001` shifted left 4 bits = `00010000 (16)`
`"48>>3"`	`6`	`00110000` shifted right 3 bits = `00000110 (6)`
`"17 & 3"`	`1`	
`"17 | 3"`	`19`	
`"17 ^ 3"`	`18`	

- **Other integer operators**

`expr var1 + var2` eg. `A=2; B=5; C=$(expr $A + $B);` or **`C=$[$A+$B]`**

CONDITIONAL CONTROL COMMANDS

------for-------

```
for variable in word1 word2 . . . wordn
do
     commands
done
```

Executes *commands* once for each *word*, setting *variable* to successive *words* each time.

```
for ((var=; $var<=limitvalue; var++))
do
     commands
done
```

Executes *commands* for each loop where `var` is an integer variable which is set initially with *initialvalue,* is incremented of '1' at each loop(`var++`) and will keep looping until *var* has exceeded the *limitvalue*.

eg1. `for ((i=100; $i>=10; i=i-5))` (from 100 to 10(included) step -5)

eg.2
```
for ((i=1; $i<=10; i++))
do
     echo "Value of \$i is $i"
done
```

Loops 10 times. For the initial loop the nalues of `$i` is '1'. At each subsequent loop the value of `$i` is incremented. The loop is not any more executed when the value of `$i` is higher than `10`.

```
for variable
do
     commands
done
```

Execute *commands* once for each positional parameter, setting *variable* to successive positional parameters each time.

------until------

```
until command1 or
until test
do
     commands
done
```

Execute *commands* until *command1* returns a zero exit status

---

------while------

```
while command1 or
while test
do
     commands
done
```

Execute *commands* while *command1* returns a zero exit status.

Example of reading a file which has a fixed number of columns(6):

```
while read dev mountpt fs options dump fsck; do
     echo dev mountpt fs options dump fsck
done </etc/fstab
```

or

Creating a list of empty directories

```
find $StartDir -type d >/tmp/dirlist
while { read dir ; }; do
    if ! (ls -1 "$dir"| egrep -v "^\.$|^\.\.$" &>/dev/null); then
        echo "$dir"
    fi
done < /tmp/dirlist
```

------if------

```
     if command1 or
     if (command1) or
     if { command1 ; } ;then
          commands
     fi
```

Execute *commands* if *command1* returns a zero exit status.
Command in `(...)` are executed in a forked shell, commands in `{ ... ; }` are executed in the same shell. The **`;`** at the end of commands, the spaces between `{ }` and the commands are important.

```
     if test_expression ; then
          commands
     fi
```

Execute *commands* if *test_expression* is true (returns a zero exit status).
test_expression is in format `test expression` or is enclosed in `[ expression ]`.
It uses the format listed in page 8 & 9.

```
     if command1 ; then
          commands2
     else
          commands3
     fi
```

Execute *commands2* if *commands1* returns a zero exit status, otherwise execute *commands3*.

```
     if command1
     then
          commands
     elif command2 ; then
          commands
     . . .
     elif commandn ; then
```

```
        commands
    else
        commands
    fi
```

If*command1* returns a zero exit status, or *command2* returns a zero exit status, or *commandn* returns a zero exit status, then execute the *commands* corresponding to the `if/elif` that returned a zero exit status. Otherwise, if all the `if/elif` commands return a non-zero exit status, execute the *commands* between `else` and `fi`.

Extra `if` examples:

```
eg1.  if [ "$#" -eq 2 ]
  or  if test "$#" -eq 2
```

```
eg2.  if [ ! -f $AA -a -f $BB ]; then mv $AA $BB; fi
```

eg3: Logical AND of commands

```
if (cat /etc/motd &>/dev/null && cat /etc/fstab &>/dev/null); then
    echo "all OK"
fi
```

eg4: Logical OR of commands

```
if (cat /etc/motd &>/dev/null || cat /etc/fstab &>/dev/null); then
    echo "all OK"
fi
```

------case------

```
    case value in
       pattern1 )     commands1 ;;
       pattern2 )     commands2 ;;
       . . .
       patternn )     commandsn ;;
    esac
```

Execute *commands*$_x$ associated with the *pattern* that matches *value*; patterns can contain the special filename substitution characters like *****, **?**, and **[ ]**. Multiple patterns can be given but must be separated with a **'|'** character.

--------Interrupting Loops------

`for`, `while`, or `until` loops can be interrupted by `break` or `continue` commands.

`break` command transfers the control to the command after the `done` command, terminating the execution of the loop.

`continue` command transfers control to the `done` command, which continues execution of the loop.

BUILTIN COMMANDS

:	null command; returns zero exit status
. *file*	read and execute commands from *file* in current shell
#	begin comments; terminate with a newline
alias [alias=...]	Displays or defines aliases
break	exit from current `for`, `until`, or `while` loop
break *n*	exit from *nth* enclosing `for`, `until`, or `while` loop
continue	jumps to the next `done` statement in a `for`, `until`, or `while` loop
cd *dir*	change current directory(pwd) to *dir* diretory If *dir* not specified, change directory to $**HOME**.
echo *args*	Display *args*
env	Displays all environment variables and functions tagged for export.
eval *command*	evaluate *command* and execute the result . eg. `L="l" ; eval $L"s"` Runs **ls** command
exec *command*	replace current process with *command*
exit	exit from current program with the exit status of the last command. If given at the command prompt, terminate the login shell.
exit *n*	exit from the current program with exit status *n*
export	display a list of exported variables
export *var*	export variable *var*
getopts	parse positional parameters and options
hash	display a list of hashed commands
hash *commands*	remember locations of *commands* by putting them in the hash table
hash -r	remove all commands from the hash table
hash -r *cmd*	remove command *(cmd)* from the hash table
newgrp	change the group-id to the default group-id
newgrp *gid*	change group id to *gid*
pwd	display the pathname of the current directory
read *varlist*	read a line from standard input; assign each word on the line to each variable. Words delimited with $**IFS**.
readonly	display a list of readonly variables
readonly *var*	set variable *var* to be readonly
return	exit from a function with return status of the last command
return *n*	exit from a function with return status *n*
set	display a list of current variables and their values, including functions
set *args*	set positional parameters to *args*
set -*args*	set positional parameters that begin with '**-**'
set [*options*]	enable/disable options (see **OPTIONS** section)
shift	shift positional parameters once to the left
shift *n*	shift positional parameters *n* times to the left
test *expr.*	evaluate *expr.* (see **CONDITIONAL EXPRESSIONS** section)
times	Show total user & system time for current shell and its child processes

trap	display list of current traps
trap *commands signals*	execute *commands* when *signals* are received
Trap "" *signals*	ignore *signals*
trap *signals*, **trap -***signals*	reset traps to their default values
trap *commands* **0**	execute *commands* on exit from the shell
type *command*	display information and location for *command*
ulimit [*type*] [*options*] *n*	set a resource limit to *n*. If *n* is not given, the specified resource limit is displayed. If no *option* is given, the file size limit (**-f**) is displayed. If no *type* is given, both limits are set, or soft limit is displayed;
type	**-H** hard limit
	-S soft limit
options	**-a** displays all current resource limits
	-c *n* set core dump size limit to *n* 512-byte blocks
	-d *n* set data area size limit to *n* kilobytes
	-f *n* set child process file write limit to *n* 512-byte blocks (default)
	-m *n* set physical memory size limit to *n* kilobytes
	-s *n* set stack area size limit to *n* kilobytes
	-t *n* set process time limit to *n* seconds
	-v*n* set virtual memory size to *n* kilobytes
umask	display current file creation mask value
umask *mask*	set default file creation mask to octal *mask*
unset *variable*	remove definition of *variable*
wait [*n*]	wait for execution (see **JOB CONTROL** section)

RESTRICTED SHELL

Running the restricted shell **rsh** is equivalent to **sh**, except the following are not allowed:

- changing directories
- setting the value of **PATH** or specifying the path of a command
- running command of which their names contain one or more '/'
- and redirecting output with '>' or '>>'.

DEBUGGING BOURNE SHELL SCRIPTS

The Bourne shell provides a number of options that are useful in debugging scripts:

-n causes commands to be read without being executed and is used to check for syntax errors.

-v option causes the input to displayed as it is read.

-x option causes commands in Bourne shell scripts to be displayed as they are executed. This is the most useful, general debugging option.
For example, **tscript** could be run in trace mode if invoked: **sh -x tscript**

FUNCTIONS

- They are normally used like fast local mini-scripts within a shell which need to be called more than once within the interactive shell or script.
- Variables can be passed-on to functions and will be recognized as `$1 $2 $3` etc. In fact the following variables are local within a function:

`$1 - $9`	Positional parameters
`$#`	Number of positional parameters
`$*`	`"$1 $2 $3 ..."`
`$@`	`"$1" "$2" "$3" ...`

- The Positional parameter `$0` and all other variables stay global within the shell unless the command `local` *`VariableName`* is given within the function. Within a function, the variable `FUNCNAME` is used instead of the `$0`.
- Global shell or exported variables can be changed within the function.
- Functions do not return variables except for the `return` number, eg. `return 5.` `return` command will also terminate the function immediately. The `return` number can then be read as a normal *exit code* using the `$?`.
- In scripts normally functions are included at the top so that they are read in first.
- Environment functions can be put into a file and read in with the `'.'` command.
- Functions may be recursive. No limit is imposed on the number of recursive calls.
- Functions can be exported, using the command: `export -f` *`FunctionName`*
- Function syntax:

```
FunctionName() {   or          function FunctionName {
     commands ;                      commands ;

}                              }
```

- The command: `unset -f` *`FunctionName`* Deletes an existing function.

ALIASES

- Aliases are normally used to create command shortcuts(short names).
- Aliases are NOT exportable: not passed-on to sub-shells or child process.
- Aliases are not recognized in scripts.
- An alias can call another alias within the command.
  eg. `alias li="ls -l"; alias al="li -a"` : `al` calls the alias '`li`'
- Parameters added to `alias` will be added at the end of the real command.
- The parameters variables (`$1`, `$2`, `$3` ...etc.) cannot be used within aliasses.
- Aliases are often defined in a file run within a script (eg. `~/.bashrc` or `~/.profile`) with the dot '`.`' command.
  - Alias commands:

`alias` *`AliasName`*`="`*`command`*`(s)..."`	Sets a new alias value
eg. `alias cp="cp -i"`	replaces the original command `cp` with `cp -i` for interactive copying.(asks before overwriting files)
`unalias` *`AliasName`*	Un-sets(deletes) the `alias`.
`alias`	Displays all the current shell aliases.

COMMAND SEARCH PRIORITY

When a command is run, bash tries to find the command in the following sequence:
- Aliases
- Functions
- Built-in commands
- PATH

the first command found is the one which is run.
To force using a builtin command instead of an alias or a function (in the case the same command name exists as alias or function), use the command `builtin`.
eg. `builtin cat /etc/fstab`

FILES

Files read	**Interactive-login Bash** (eg. `bash --login` or `su - username` or from `login` program)
`/etc/profile`	Executed first from interactive login shell. It contains system-wide environment settings. If existent, it is read in and executed before `$HOME/.profile`.
`/etc/bash.bashrc`	Executed first from interactive login shell. (SuSE 9.2 and up use it) Same purpose as `/etc/profile`
`~/.bash_profile`	Individual users shell settings. If exist is executed after `/etc/profile.`
`~/.bash_login`	Executed if `~/.bash_profile` doesn't exist.
`~/.profile`	Executed if `~/.bash_login` or `~/.bash_profile` doesn't exist.
	Interactive NON-Login Bash (eg. `su username` or `bash -c cmd`)
`~/.bashrc`	The only script executed when started. And inherits from parent bash environment.
	NON-Interactive NON-Login Bash(forked when scripts are run)
	No above scripts are executed but inherits env. from parent.
`BASH_ENV`	Reads file in the variable `BASH_ENV`.
`ENV`	Reads file in the variable `ENV` if `BASH_ENV` doesn't exist.
	Extra files
`/etc/inputrc`	System readline initialization file
`~/.inputrc`	Individual readline initialization file
`~/.bash_logout`	Executed when a login shell exits.

SET and UNSET commands

set

Syntax: `set [--abefhkmnptuvxBCHP] [-o option] [arg ...]`

The `set` command is used to:

- Set bash operating attributes(using options)
- To assign values to positional parameters: eg.

`set -a` Automatically mark variables and functions which are modified or created for export to the environment of subsequent commands.

`set aaa bbb ccc`
Assigns the value `aaa` to `$1`, `bbb` to `$2` and `ccc` to `$3`.

unset

Syntax: `unset [-fv] [name ...]`

For each name, remove the corresponding variable or function.
Each unset variable or function is removed from the environment passed to subsequent commands. If any of `RANDOM`, `SECONDS`, `LINENO`, `HISTCMD`, `FUNCNAME`, `GROUPS`, `DIRSTACK` are unset, they lose their special properties, even if they are subsequently reset.
The exit status is true unless a name does not exist or is readonly.

`-v` If no options are supplied, or the `-v` option is given, each name refers to a shell variable.
Read-only variables may not be unset.

`-f` Each name refers to a shell function, and the function definition is removed.

```
eg.  unset DISPLAY       : Deletes the variable DISPLAY
     unset -f startx     : Deletes the function startx
```

REGULAR EXPRESSIONS

c	non-special character *c*
c	special character *c*
^	beginning of line
$	end of line
.	any single character
[*abc*]	any character *a*, *b*, or *c*
[*a-c*]	any character in range *a* through *c*
[^*abc*]	any character except *a*, *b*, or *c*
[^*a-c*]	any character except characters in *a-c*
n	*nth* \(...\) match (**grep** only)
*rexp**	zero or more occurrences of *rexp*
rexp+	one or more occurrences of *rexp*
rexp?	zero or one occurrence of *rexp*
rexp1 | *rexp2*	regular expressions *rexp1* or *rexp2*
\(*rexp*\)	tagged regular expression *rexp* (**grep**)
(*rexp*)	regular expression *rexp* (**egrep**)

echo COMMAND:

`echo`	`-e`	`"...\a..."`	Alert (bell) --Note: only in Virtual Terminal(not in `xterm`)
""	""	`"...\b..."`	Backspace
""	""	`"...\c..."`	Suppress trailing new line
""	""	`"...\f..."`	Form Feed
""	""	`"...\n..."`	New Line `echo -e "\012"`
""	""	`"...\r..."`	Carriage Return
""	""	`"...\t..."`	Horizontal Tab `echo -e "\011"`
""	""	`"...\v..."`	Vertical Tab
""	""	`"...\\..."`	Litteral Baskslash \
""	""	`"...\'..."`	Single quote
""	""	`"...\`*nnn*`..."`	The eight-bit character whose value is the octal value *nnn* (one to three digits)
""	""	`"...\x`*HH*`..."`	The eight-bit character whose value is the hexadecimal value HH (one or two hex digits)
""	""	`"...\cx..."`	A <Control-x> character

PROMPT MANIPULATION

The shell PROMPT display can be modified by changing the value of the **PS1** variable to any desired text. The following special character combinations(*x*) introduces the corresponding entry into the PROMPT as well.

`\a`	a bell character.
`\d`	the date, in "Weekday Month Date" format (e.g., "Tue May 26").
`\e`	an escape character.
`\h`	the hostname, up to the first '.'
`\H`	the hostname.
`\n`	newline.
`\s`	the name of the shell, the basename of `$0` (the portion following the final slash).
`\t`	the time, in 24-hour HH:MM:SS format.
`\T`	the time, in 12-hour HH:MM:SS format.
`\@`	the time, in 12-hour am/pm format.
`\v`	the version of Bash (e.g., 2.00)
`\V`	the release of Bash, version + patchlevel (e.g., 2.00.0)
`\w`	the current working directory.
`\W`	the basename of $PWD.
`\u`	your username.
`\!`	the history number of this command.
`\#`	the command number of this command.
`\$`	if the effective UID is 0, #, otherwise $.
`\nnn`	the character corresponding to the octal number nnn.
`\\`	a backslash.
`\[`	begin a sequence of non-printing characters. This could be used to embed a terminal control sequence into the prompt.
`\]`	end a sequence of non-printing characters.

eg. `PS1=\u@\h:\w >` Could display a prompt as follows:

```
mario@topserver:/root >
```

Job control (disown) Exercise:

- - Start `xterm` and in this `xterm` start another `xterm` (**`xterm &`**)
  - close first `xterm`.....the second is **not** closed.

- - Start `xterm`
  - in `xterm` start `wterm` in background (**`wterm &`**)
  - Close `xterm`.....the wterm is also closed (owned by xterm)

- - Start `xterm`
  - in `xterm` start `wterm` (**`wterm &`**)
  - in `xterm` > **`jobs`** ....shows the background job
  - in `xterm` > **`disown`** .... the last active job is disowned
  - Close `xterm`.....the `wterm` is NOT closed.

Bash session recording:

A bash session(commands and results) can be recorded into a file by entering the command '`script` ***`filename`***' before starting to record . A new shell will then start and all the commands typed and their results will be saved into the file ***`filename.`*** To stop the recording of the session, compose the `<Ctgrl-D>` key combination.

Monitoring a bash session from one or more users:

This above method can also be used for monitoring/teaching purposes if other users read live this recorded file using the command `tail -f` ***`filename`***. There will be a 1 second time delay between the original and the file read.
Another variation of this technique is to send the output of `script` into a pipe and to read it from one user only via the `cat` command.
eg.
IN THE ORIGINAL TERMINAL:

```
mkfifo /tmp/session
script /tmp/session
```

start typing commands
.....
`<Ctgrl-D>` to terminate `script`

IN THE LISTENING TERMINAL:

```
cat /tmp/session
```

Note: If in the original terminal **mc** is started, then some strange display of **mc** will occur in the listening terminal unless the dimensions and fonts are the same as the original terminal.

Bash options:

Bash can be started with different options which alter the way bash works.

`SHELLOPTS`	Environment variabble storing the current bash options
`set -o` *`option`*	Command used to turn a current bash option ON.
`set +o` *`option`*	Command used to turn a current bash option OFF.

eg.1

`set -o emacs`	Sets the `emacs` editing keys/commands:default
`set -o vi`	Sets the `vi` editing keys/commands

eg.2 `set -o noclobber`
Prevents commands from overwriting files when redirections (>) are used.
eg.

```
set -o noclobber    (or set -C)
touch xxxlog
ls /home > xxxlog
bash: xxxlog: cannot overwrite existing file
ls /home >| xxlog   (>| can override the overwriting restrictions)
```

eg.3 `set -x`
Sets bash in debugging mode. It will display the commands as they are really executed by bash after bash has done its first scanning of the command. This first scanning of the command is normally done to allow bash to expand the file globing characters.

Command History and command line editing:

Command history navigation:

`set +o history`	Turns history recording OFF
`set -o history`	Turns history recording ON
`$HISTFILE`	Variable containing the history file name. Normally `~/.bash_history`
`$HISTFILESIZE`	Variable containing the maximum number of commands the history file can contain. Default=`500`
`$HISTSIZE`	Variable containing the maximum number of commands in history. Default=`500`
`history`	Displays the whole history
`history 10`	Displays the last 10 lines of history
`fc -l -10`	Displays the last 10 lines of history
`fc -l` *`Pattern`*	Search the history for `Pattern` & display the result
`<Ctrl>-r`	Reverse search in history
`history -c`	Clears the whole history
`!!`	Most recent command
`!`*`n`*	Command *n* in the history
`!-`*`n`*	Backwards command *n* in history
`!` *`string`*	Last recent command starting with *`string`*
`!?` *`string`*	Last recent command containing with *`string`*
`^`*`string1`*`^`*`string2`*	Quick substitution *`string1`* to *`string2`*
`<Ctrl>-p`	Previous Line in history (also up-arrow)
`<Ctrl>-n`	Next Line in history (also down arrow)
`<Alt>-<`	Go to beginning of History
`<Alt>->`	Go to end of History

Command Line Editing commands (E-macs editing cmds -`readline` library)

`<Ctrl>-l`	Clear screen
`<Ctrl>-b`	Back one character (also left arrow)
`<Ctrl>-f`	Foreward one character (also right arrow)
`<Ctrl>-a`	Go to beginning of line (also `Pos1` key)
`<Ctrl>-e`	Go to end of line (also `Ende` key)
`<Ctrl>-k`	Delete text from cursor to end of line
`<Ctrl>-d`	Delete a character on the right (or under cursor)
`<Alt>-d`	Delete from crursor to end of current word
`<Ctrl>-y`	Paste text previously cut (deleted)

EXAMPLE COMMANDS

```
# Execute multiple commands on one line
   pwd ; ls tmp ; echo "Hello world"
# Run the find command in the background
   find . -name tmp.out -print &
# Connect the output of who to grep
   who | grep fred
# Talk to fred if he is logged on
   { who | grep fred ;  } && talk fred
# Send ls output to ls.out
   ls > ls.out
# Append output of ls to ls.out
   ls >> ls.out
# Send invite.txt to dick, jane, and spot
   mail dick jane spot < invite.txt
# Send the standard error of xsend to stderr.out
   xsend file 2>stderr.out
# List file names that begin with z
   ls z*
# List two, three, and four character file names
   ls ?? ??? ????
# List file names that begin with a, b, or c
   ls [a-c]*
# List file names that do not end with .c
   ls *[!.c]
# Set NU to the number of users that are logged on
   NU=`who | wc -l`  or  NU=$(who | wc -l)
# Set TOTAL to the sum of 4 + 3
   TOTAL=`expr 4 + 3`  or   TOTAL=$[4+3]
# Set and export the variable LBIN
   LBIN=/usr/lbin; export LBIN
# Unset variable LBIN
   unset LBIN
# Set SYS to the Operating System Name if not set, then display its value
   echo ${SYS:=`uname -o`}
# Display an error message if XBIN is not set
   : $X{BIN:?}
# Display $HOME set to /home/anatole
   echo '$HOME set to' $HOME
# Display the value of $TERM
   echo $TERM
# Bring background job 3 into the foreground
   fg %3
# Stop the find job
   stop %find
# Display the number of positional parameters
   echo "There are $# positional parameters"
```

```
# Display the value of positional parameter 2
   echo $2
# Display all information about current jobs
   jobs -l
# Terminate job 5
   kill %5
# Increment variable X
   X=`expr $X + 1`   or   let X++   or   X=$[$X+1]
# Set variable X to 20 modulo 5
   X=`expr 20 % 5`
# Set diagnostic mode
   set -x
# Run the dbscript in noexec mode
   sh -n dbscript
# Check for new mail every 2 minutes
   MAILCHECK=120; export MAILCHECK
# Set the primary prompt string PS1
   PS1='Good morning!'; export PS1
# Check if VAR is set to null
   [-z "$VAR"] && echo "VAR is set to null"
# Check if VAR is set to ABC
   [ "$VAR" = ABC ]
# Check if xfile is empty
   test ! -s xfile
# Check if tmp is a directory
   [ -d tmp ]
# Check if file is readable and writable
   test -r file -a -w file
# Display an error message, then beep(doesn't work inside xterm)
   echo "Unexpected error!\007"
# Display a message on standard error
   echo "This is going to stderr" >&2
# Display a prompt and read  the reply into ANSWER
    echo "Enter response: \c"; read ANSWER
or  echo -n "Enter response: "; read ANSWER
# Create a function md that creates a directory and cd's to it
   md() { mkdir $1 && cd $1 ; pwd ; }
# Set a trap to ignore signals 2 and 3
   trap "" 2 3
# Set X to 1 and make it readonly
   X=1 ; readonly X
# Set VAR to 1 and export it
   VAR=1 ; export VAR or export VAR=1
# Set the positional parameters to A B C
   set A B C
# Set the file size creation limit to 1000 blocks
   ulimit 1000
# Disable core dumps
   ulimit -c 0
```

```
# Add group write permission to the file creation mask
   umask 013
# Display the first and third fields from file
   awk '{print $1, $3}' filename
or sed 's/ */ /' filename | cut -d" " -f1,3
# Display the first seven characters of each line in tfile
   cut -c1-7 tfile
# Display the first and third fields from the /etc/passwd file
   cut -f1,3 -d":" /etc/passwd
# Display lines in names that begin with A, B, C, or Z
   egrep '[A-C,Z]*' names
# Display lines from dict that contain four character words
   egrep '....' dict
# Display password entries for users with the Korn shell
   grep ":/bin/ksh$" /etc/passwd
# Display number of lines(-c) in ufile that contain unix ; ignore case(-i)
   grep -ci 'unix' ufile
# Display the lengths of field 1 from file
   nawk'{TMP=length($1);print $TMP}' file
# Display the first 10 lines of tfile
   nawk '{for (i=1; i<10; i++) printf "%s\n", getline}' tfile
or head tfile
# List the contents of the current directory in three columns
   ls | paste d" " - - -
# Sort the /etc/passwd file by group id in numerical order(-n).
      sort -t":" -n +3 -4 /etc/passwd
or    sort -t":" -nk4 /etc/passwd
# Translate lower case letters in file to upper case
   cat file | tr a-z A-Z
# Display adjacent duplicate lines in file
   uniq -d file
# Display the numbers of lines in file
   wc -l file
# Display the number of .c files in the current directory
   ls *.c | wc -l
# Substitutes all instances of '/' in a variable to '\/'.
  Preparing for use in a sed command.
  variable2=$(echo $variable | sed 's/\//\\\//g')
# Display file with all occurrences of The substituted with A
   sed 's/The/A/g' file
# Display your user name only
   id | sed 's/).*//' | sed 's/.*(//'
# Display file with lines that contain unix deleted
   sed '/unix/d' file
# Display the first 75 lines of file
   sed 75q file   or  head -n75 file
-----------------------------------------------------
```

ANSI/VT100 Terminal Control

Many computer terminals and terminal emulators support color and cursor control through a system of escape sequences. One such standard is commonly referred to as ANSI Color. Several terminal specifications are based on the ANSI color standard, including VT100.

The following is a partial listing of the VT100 control set.

`\033` represents the ANSI "escape" character, 0x1B. Bracketed tags represent modifiable decimal parameters; eg. **`{ROW}`** would be replaced by a row number.

Device Status

The following codes are used for reporting terminal/display settings, and vary depending on the implementation:

`Query Device Code` **`echo -e \033[c`**

- Requests a **`Report Device Code`** response from the device.

`Report Device Code` **`echo -e \033[{code}0c`**

- Generated by the device in response to **`Query Device Code`** request.

`Query Device Status` **`echo -e \033[5n`**

- Requests a **`Report Device Status`** response from the device.

`Report Device OK` **`echo -e \033[0n`**

- Generated by the device in response to a **`Query Device Status`** request; indicates that device is functioning correctly.

`Report Device Failure` **`echo -e \033[3n`**

- Generated by the device in response to a **`Query Device Status`** request; indicates that device is functioning improperly.

`Query Cursor Position` **`echo -e \033[6n`**

- Requests a **`Report Cursor Position`** response from the device.

`Report Cursor Position` **`echo -e \033[{ROW};{COLUMN}R`**

- Generated by the device in response to a **`Query Cursor Position`** request; reports current cursor position.

Terminal Setup

The **`h`** and **`l`** codes are used for setting terminal/display mode, and vary depending on the implementation. Line Wrap is one of the few setup codes that tend to be used consistently:

`Reset Device` **`echo -e \033c`**

- Reset all terminal settings to default.

Enable Line Wrap **echo -e \033[7h**

- Text wraps to next line if longer than the length of the display area.

Disable Line Wrap **echo -e \033[7l**

- Disables line wrapping.

Fonts

Some terminals support multiple fonts: normal/bold, swiss/italic, etc. There are a variety of special codes for certain terminals; the following are fairly standard:

Font Set G0 **echo -e \033(**

- Set default font.

Font Set G1 **echo -e \033)**

- Set alternate font.

Cursor Control

Cursor Home **echo -e \033[{ROW};{COLUMN}H**

- Sets the cursor position where subsequent text will begin. If no row/column parameters are provided (ie. **echo -e \033[H**), the cursor will move to the *home* position, at the upper left of the screen.

Cursor Up **echo -e \033[{COUNT}A**

- Moves the cursor up by *COUNT* rows; the default count is 1.

Cursor Down **echo -e \033[{COUNT}B**

- Moves the cursor down by *COUNT* rows; the default count is 1.

Cursor Forward **echo -e \033[{COUNT}C**

- Moves the cursor *forward* by *COUNT* columns; the default count is 1.

Force Cursor Position **echo -e \033[{ROW};{COLUMN}f**

- Identical to **Cursor Home**.

Save Cursor **echo -e \033[s**

- Save current cursor position.

Unsave Cursor **echo -e \033[u**

- Restores cursor position after a **Save Cursor**.

Save Cursor & Attrs **echo -e \0337**

- Save current cursor position.

Restore Cursor & Attrs **echo -e \0338**

- Restores cursor position after a **Save Cursor**.

Scrolling

Scroll Screen **echo -e \033[r**

- Enable scrolling for entire display.

Scroll Screen **echo -e \033[{start};{end}r**

- Enable scrolling from row **{start}** to row **{end}**.

Scroll Down **echo -e \033D**

- Scroll display down one line.

Scroll Up **echo -e \033M**

- Scroll display up one line.

Tab Control

Set Tab **echo -e \033**

- Sets a tab at the current position.

Clear Tab **echo -e \033[g**

- Clears tab at the current position.

Clear All Tabs **echo -e \033[3g**

- Clears all tabs.

Erasing Text

Erase End of Line **echo -e \033[K**

- Erases from the current cursor position to the end of the current line.

Erase Start of Line **echo -e \033[1K**

- Erases from the current cursor position to the start of the current line.

Erase Line **echo -e \033[2K**

- Erases the entire current line.

Erase Down **echo -e \033[J**

- Erases the screen from the current line down to the bottom of the screen.

Erase Up **echo -e \033[1J**

- Erases the screen from the current line up to the top of the screen.

Erase Screen **echo -e \033[2J**

- Erases the screen with the background color and moves the cursor to *home*.

Printing

Some terminals support local printing:

Print Screen **echo -e \033[i**

- Print the current screen.

Print Line **echo -e \033[1i**

- Print the current line.

Stop Print Log **echo -e \033[4i**

- Disable log.

Start Print Log **echo -e \033[5i**

- Start log; all received text is echoed to a printer.

Define Key

Set Key Definition **echo -e \033[{key};"{string}"p**

- Associates a *string* of text to a keyboard key. **{key}** indicates the key by its ASCII value in decimal.

Set Display Attributes

Set Attribute Mode **echo -e \033[{attr1};...;{attrn}m**

- Sets multiple display attribute settings. The following lists standard attributes:

```
0       Reset all attributes
1       Bright
2       Dim
4       Underscore
5       Blink
7       Reverse
8       Hidden

        Foreground Colors
30      Black
31      Red
32      Green
33      Yellow
34      Blue
```

```
35      Magenta
36      Cyan
37      White

        Background Colors
40      Black
41      Red
42      Green
43      Yellow
44      Blue
45      Magenta
46      Cyan
47      White
```