Advanced Bash−Scripting HOWTO

Table of Contents

Table of Contents

Advanced Bash−Scripting HOWTO

A guide to shell scripting, using Bash

Mendel Cooper

thegrendel@theriver.com

v0.1, 14 June 2000

This document is both a tutorial and a reference on shell scripting with Bash. It assumes no previous knowledge of scripting or programming, but progresses rapidly toward an intermediate/advanced level of instruction. The exercises and heavily−commented examples invite active reader participation. This is essentially a synopsis of a complete book on the subject.

Table of Contents

Chapter 1. Why Shell Programming?

The shell is a command interpreter. It is the insulating layer between the operating system kernel and the user. Yet, it is also a fairly powerful programming language. A shell program, called a *script* , is an easy−to−use tool for building applications by "gluing" together system calls, tools, utilities, and compiled binaries. Virtually the entire repertoire of UNIX commands, utilities, and tools is available for invocation by a shell script. If that were not enough, internal shell commands, such as testing and loop constructs, give additional power and flexibility to scripts. Shell scripts lend themselves exceptionally well to to administrative system tasks and other routine repetitive jobs not requiring the bells and whistles of a full−blown tightly structured programming language.

A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in `/etc/rc.d` to restore the system configuration and set up services. A detailed understanding of these scripts is important for analyzing the behavior of a system, and possibly modifying it.

Writing shell scripts is not hard to learn, since the scripts can be built in bite−sized sections and there is only a fairly small set of shell−specific operators and options to learn. The syntax is simple and straightforward, similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" to learn. Most short scripts work right the first time, and debugging even the longer ones is straightforward.

A shell script is a "quick and dirty" method of prototyping a complex application. Getting even a limited subset of the functionality to work in a shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl.

Shell scripting hearkens back to the classical UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities. Many consider this a better, or at least more esthetically pleasing approach to problem solving than using one of the new generation of high powered all−in−one languages, such as Perl, which attempt to be all things to all people, but at the cost of forcing you to alter your thinking processes to fit the tool.

When not to use shell scripts

- resource−intensive tasks, especially where speed is a factor
- complex applications, where structured programming is a necessity
- file handling (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line−by−line fashion)
- need to generate or manipulate graphics or GUIs
- need direct access to system hardware
- need port or socket I/O

- need to use libraries or interface with legacy code

If any of the above applies, consider a more powerful scripting language, perhaps Perl, Tcl, Python, or even a high−level compiled language such as C, C++, or Java. Even then, prototyping the application as a shell script might still be a useful development step.

We will be using Bash, an acronym for "Born−Again Shell" and a pun on Stephen Bourne's now classic Bourne Shell. Bash has become the de facto standard for shell scripting on all flavors of UNIX. Most of the principles dealt with in this document apply equally well to scripting with other shells, such as the Korn Shell, from which Bash derives some of its features, and the C Shell and its variants. (Note that C Shell programming is not recommended due to certain inherent problems, as pointed out in a news group posting by Tom Christiansen in October of 1993).

The following is a tutorial in shell scripting. It relies heavily on examples to illustrate features of the shell. As far as possible, the example scripts have been tested, and some of them may actually be useful in real life. The reader should cut out and save the examples, assign them appropriate names, give them execute permission (**`chmod u+x scriptname`**), then run them to see what happens. Note that some of the scripts below introduce features before they are explained, and this may require the reader to temporarily skip ahead for enlightenment.

Unless otherwise noted, the author of this document wrote the example scripts that follow.

Chapter 2. Starting Off With a Sha−Bang

In the simplest case, a script is nothing more than a list of system commands stored in a file. At the very least, this saves the effort of retyping that particular sequence of commands each time it is invoked.

Example 2−1. cleanup: A script to clean up the log files in /var/log

```
# cleanup
# Run as root, of course.

cd /var/log
cat /dev/null 62; messages
cat /dev/null 62; wtmp
echo "Logs cleaned up."
```

There is nothing unusual here, just a set of commands that could just as easily be invoked one by one from the command line on the console or in an xterm. The advantages of placing the commands in a script go beyond not having to retype them time and again. The script can easily be modified, customized, or generalized for a particular application.

Example 2−2. cleanup An enhanced and generalized version of above script.

```
#!/bin/bash
# cleanup, version 2
# Run as root, of course.

if [ −n $1 ]
# Test if command line argument present.
then
  lines=$1
else
  lines=50
  # default, if not specified on command line.
fi


cd /var/log
tail −$lines messages 62; mesg.temp
# Saves last section of message log file.
mv mesg.temp messages

# cat /dev/null 62; messages
# No longer needed, as the above method is safer.

cat /dev/null 62; wtmp
echo "Logs cleaned up."

exit 0
# A zero return value from the script upon exit
# indicates success to the shell.
```

Since you may not wish to wipe out the entire system log, this variant of the first script keeps the last section of the message log intact. You will constantly discover ways of refining previously written scripts for increased effectiveness.

The *sha−bang* (#!) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two byte " magic number" that marks an executable shell script (**`man magic`** gives more info on this fascinating topic). It also gives the path to the program that the script invokes, whether this be the shell, a programming language, or a utility. This enables the specific commands and directives embedded in the shell or program called.

```
#!/bin/sh
#!/bin/bash
#!/bin/awk
#!/usr/bin/perl
#!/bin/sed
#!/usr/bin/tcl
```

Each of the above script header lines calls a different command interpreter, be it `/bin/sh`, the default shell (**bash** in a Linux system) or otherwise. Using **`#!/bin/sh`**, the default Bourne Shell in most commercial variants of UNIX, makes the script portable to non−Linux machines, though you may have to sacrifice a few bash−specific features (the script will conform to the POSIX **sh** standard).

Note that the path given at the "sha−bang" must be correct, otherwise an error message, usually Command not found will be the only result of running the script.

#! can be omitted if the script consists only of a set of generic system commands, using no internal shell directives. Example 2, above, requires the initial #!, since the variable assignment line, **`lines=50`**, uses a shell−specific construct. Note that **`#!/bin/sh`** invokes the default shell interpreter, which defaults to `/bin/bash` on a Linux machine.

2.1. Invoking the script

Having written the script, you can invoke it by **`sh scriptname`**, or alternately **`bash scriptname`**. (Not recommended is using **`sh <scriptname`**, since this effectively disables reading from input within the script.) Much more convenient is to make the script itself directly executable by

> *Either:*
>
> > **`chmod 755 scriptname`** (gives everyone execute permission)
>
> *or*
>
> > **`chmod +x scriptname`** (gives everyone execute permission)
> > **`chmod u+x scriptname`** (gives only the script owner execute permission)

In this case, you could try calling the script by **`./scriptname`**.

As a final step, after testing and debugging, you would likely want to move it to `/usr/local/bin` (as root, of course), to make the script available to yourself and all other users as a system−wide executable. The script could then be invoked by simply typing **scriptnamereturn**' from the command line.

2.2. Shell wrapper, self−executing script

A **sed** or **awk** script would normally be invoked from the command line by a **`sed −e`** ***`commands`*** or **`awk −e`** ***`commands`***. Embedding such a script in a bash script permits calling it more simply, and makes it "reusable". This also permits combining the functionality of **sed** and **awk**, for example piping the output of a set of **sed** commands to **awk**. As a saved executable file, you can then repeatedly invoke it in its original form or modified, without retyping it on the command line.

Example 2−3. wrapper

```
#!/bin/bash

# This is a simple script
# that removes blank lines
# from a file.
# No argument checking.

# Same as
# sed −e '/^$/d $1' filename
# invoked from the command line.

sed −e /^$/d $1
# '^' is beginning of line,
# '$' is end,
# and 'd' is delete.
```

Example 2−4. A slightly more complex script wrapper

```
#!/bin/bash

# "subst", a script that substitutes one pattern for
# another in a file,
# i.e., "subst Smith Jones letter.txt".

if [ $# −ne 3 ]
# Test number of arguments to script
# (always a good idea).
then
  echo "Usage: `basename $0` old−pattern new−pattern filename"
  exit 1
fi

old_pattern=$1
new_pattern=$2

if [ −f $3 ]
then
    file_name=$3
else
    echo "File \"$3\" does not exist."
    exit 2
fi

# Here is where the heavy work gets done.
sed −e "s/$old_pattern/$new_pattern/" $file_name
# 's' is, of course, the substitute command in sed,
# and /pattern/ invokes address matching.
# Read the literature on 'sed' for a more
# in−depth explanation.

exit 0
# Successful invocation of the script returns 0.
```

Exercise. Write a shell script that performs a simple task.

Chapter 3. Tutorial / Reference

Your (manu)script is both good and original, but the part that is good is not original and the part that is original is not good.

−−Samuel Johnson

3.1. exit and exit status

The **exit** command may be used to terminate a script, as in a C program It may also return a value, which can be read by the shell.

Every command returns an *exit status* (sometimes referred to as a *return status*). A successful command returns a 0, while an unsuccessful one returns a non−zero value that usually may be interpreted as an error code.

Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status. Within a script, an **`exit nn`** command may be used to deliver an nn exit status to the shell (*`nn`* must be a decimal number in the 0 − 255 range).

`$?` reads the exit status of script or function

Example 3−1. exit / exit status

```
#!/bin/bash

echo hello
echo $?
# exit status 0 returned
# because command successful.

lskdf
# bad command
echo $?
# non−zero exit status returned.

echo

exit 143
# Will return 143 to shell.
# To verify this, type $? after script terminates.

# By convention, an 'exit 0' shows success,
# while a non−zero exit value indicates an error.
```

3.2. Special characters used in shell scripts

#

Comments. Lines beginning with a # (with the exception of #!) are comments.

```
# This line is a comment.
```

Comments may also occur at the end of a command.

```
echo "A comment will follow." # Comment here.
```

Comments may also follow white space at the beginning of a line.

```
# A tab precedes this comment.
```

;

Command separator. Permits putting two or more commands on the same line
`echo hello; echo there`
Note that the ; sometimes needs to be escaped (\).

.

"dot" command. Equivalent to source, explained further on

:

null command. Exit status 0, alias for true, see below
Endless loop:
`while : do    operation−1    operation−2    ...    operation−n done`
Placeholder in if/then test:
`if condition then :    # Do nothing and branch ahead else    take−some−action fi`
Evaluate string of variables using "parameter substitution" (explained later on):
`: ${HOSTNAME?} ${USER?} ${MAIL?}`
Prints error message if one or more of essential environmental variables not set.

${}

Parameter substitution.

`${parameter−default}`

If parameter not set, use default

${parameter=default}

If parameter not set, set it to default

${parameter+otherwise}

If parameter set, use 'otherwise", else use null string

${parameter?err_msg}

If parameter set, use it, else print err_msg

Example 3−2. Using param substitution and :

```
#!/bin/bash

: ${HOSTNAME?} {USER?} {MAIL?}
  echo $HOSTNAME
  echo $USER
  echo $MAIL
  echo Critical env. variables set.

exit 0
```

Parameter substitution and/or expansion. The following are the equivalent of **match** in **expr** string operations (see below). These are used mostly in parsing file path names.

${var#pattern}***, ***${var##pattern}

Strip off shortest/longest part of *pattern* if it matches the front end of *variable*.

${var%pattern}***, ***${var%%pattern}

Strip off shortest/longest part of *pattern* if it matches the back end of *variable*.

Version 2 of bash adds additional options.

${var:pos}

variable *var* expanded, starting from offset *pos*.

${var:pos:len}

expansion to a max of *len* characters of variable *var*, from offset *pos*.

${var/patt/replacement}

first match of *pattern*, within *var* replaced with *replacement*.

${var//patt/replacement}

all matches of *`pattern`*, within *`var`* replaced with *`replacement`*.

Example 3−3. Using pattern matching to parse arbitrary strings

```
#!/bin/bash

var1=abcd−1234−defg
echo "var1 = $var1"

t=${var1#*−*}
echo "var1 (with everything, up to and including first − stripped out) = $t"
t=${var1%*−*}
echo "var1 (with everything from the last − on stripped out) = $t"

echo

path_name=/home/bozo/ideas/thoughts.for.today
echo "path_name = $path_name"
t=${path_name##/*/}
# Same effect as   t=`basename $path_name`
echo "path_name, stripped of prefixes = $t"
t=${path_name%/*.*}
# Same effect as   t=`dirname $path_name`
echo "path_name, stripped of suffixes = $t"

echo

t=${path_name:11}
echo "$path_name, with first 11 chars stripped off = $t"
t=${path_name:11:5}
echo "$path_name, with first 11 chars stripped off, length 5 = $t"

echo

t=${path_name/bozo/clown}
echo "$path_name with bozo replaced = $t"
t=${path_name//o/O}
echo "$path_name with all o's capitalized = $t"

exit 0
```

()

command group.
`(a=hello; echo $a)`

{}

block of code. This, in effect, creates an anonymous function.
The code block enclosed in braces may have I/O redirected to and from it.
Example 3−4. Code blocks and I/O redirection
```
#!/bin/bash  { read fstab } 60; /etc/fstab  echo "First line in /etc/fstab is:" echo "$fsta
```

/{}

file pathname. Mostly used in 'find' constructs.

>>><&

redirection.
`scriptname >filename` redirects the output of *`scriptname`* to file *`filename`*. If *`filename`* already existed, it is overwritten.
`command >&2` redirects output of *`command`* to stderr.
`scriptname >>filename` appends the output of *`scriptname`* to file *`filename`*. If *`filename`* already existed, the output of the script will be added at the end of the file.

<<

redirection used in "here document". See below.

|

pipe. Passes the output of previous command to next one, or to shell.
```
echo ls −l | sh
```
`cat *.lst | sort | uniq` sorts the output of all the `.lst` files and deletes duplicate lines.

>|

force redirection (even if `noclobber` environmental variable is in effect). This will forcibly overwrite an existing file.

−

redirection from/to stdin or stdout.
```
(cd /source/directory 38;38; tar cf − . ) | (cd /dest/directory 38;38; tar xvfp −) # Move 
bunzip2 linux−2.2.15.tar.bz2 | tar xvf − # −−uncompress tar file−−    | −−then pass it to
```

White space

functions as a separator, separating commands or variables. White space consists of either spaces, tabs, blank lines, or any combination thereof. In some contexts, such as variable assignment, white space is not permitted, and results in a syntax error.

Blank lines

Blank lines have no effect on the action of a script, and are therefore useful for visually separating functional sections of the script.

3.3. Variables

$

variable substitution. **`$variable`** is a reference to the *value* of the variable. Variables will always begin with $, except when assigned (or at the head of a loop). Note that enclosing a referenced value in double quotes (" ") does not interfere with the variable substitution, but enclosing it in single quotes (' ') causes the variable name to be used literally, and no substitution will take place.

Note that **`$variable`** is actually a simplified alternate form of **`${variable}`**. In complex cases where the **`$variable`** syntax causes an error, the longer form may work.

Example 3−5. Variable substitution

```
#!/bin/bash

a=37.5
hello=$a
# No space permitted on either side of = sign.

echo hello

echo $hello
echo ${hello} #Identical as above.

echo "$hello"
echo "${hello}"

echo '$hello'
# Variable referencing disabled by single quotes.

# Notice the effect of different
# types of quoting.

exit 0
```

Note that an uninitialized variable has a "null" value (no assigned value at all). Using a variable before assigning a value to it will cause problems.

3.4. Quoting

Quoting means just that, bracketing a string in quotes. This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning, such as the wild card character, *.)

When referencing a variable, it is generally advisable in enclose it in double quotes (" "). This preserves spaces and special characters within the variable name, but still allows referencing it, that is, replacing the variable with its value (see Example 3−5, above). Enclosing the arguments to an **echo** statement in double quotes is usually a good practice.

Single quotes (' ') operate similarly to double quotes, but do not permit referencing variables, since the special meaning of $ is turned off. Special characters, such as $ are not translated, but interpreted literally. Consider single quotes to be a stricter method of quoting than the double quotes.

Escaping is a method of quoting single characters. The escape (\) preceding a character will either toggle on or turn off a special meaning for that character, depending on context.

\n

: means newline

\r

: means return

\t

: means tab

\v

: \vmeans vertical tab

\b

: means backspace

\a

: means "alert" (beep or flash)

\0xx

: translates to the octal ASCII equivalent of *0xx*

```
# Use the −e option with 'echo' to print these. echo −e "\v\v\v\v"   # Prints 4 vert
```

\"

gives the quote its literal meaning

\$

gives the dollar sign its literal meaning (variable name following \$ will not be referenced)
```
echo "\$variable01"  # results in $variable01
```

The escape also provides a means of writing a multi−line command. Normally, each separate line constitutes a different command, but an escape at the end of a line continues the command sequence onto the next line.

```
(cd /source/directory 38;38; tar cf − . ) | \
(cd /dest/directory 38;38; tar xvfp −)
# Repeating Alan Cox's directory tree copy command,
# but split into two lines for increased legibility.
```

3.5. Tests

The if/then construct tests whether a condition is true, and if so, executes one or more commands. Note that in this context, `0` (zero) will evaluate as true, as will Why?

Example 3−6. What is truth?

```
#!/bin/bash

if [ 0 ]
#zero
then
  echo "0 is true."
else
  echo "0 is false."
fi

if [ ]
#NULL (empty condition)
then
  echo "NULL is true."
else
  echo "NULL is false."
fi

if [ xyz ]
#string
then
  echo "Random string is true."
else
  echo "Random string is false."
fi

if [ $xyz ]
#string
then
  echo "Undeclared variable is true."
else
  echo "Undeclared variable is false."
fi

exit 0
```

Exercise. Explain the behavior of Example 3−6, above.

```
if [ condition−true ]
then
   command 1
   command 2
   ...
else
   # Optional (may be left out if not needed).
   # Adds default code block executing if original condition
   # tests false.
   command 3
   command 4
   ...
fi
```

Add a semicolon when 'if' and 'then' are on same line.

```
if [ −x filename ]; then
```

elif

This is a contraction for else if. The effect is to nest an inner if/then construction within an outer one.
`if [ condition ] then    command    command    command elif # Same as else if then`

The **`test condition−true`** construct is the exact equivalent of **`if [condition−true ]`**. The left bracket [is, in fact, an alias for test. (The closing right bracket] in a test should not therefore be strictly necessary, however newer versions of bash detect it as a syntax error and complain.)

Example 3−7. Equivalence of [ ] and test

```
#!/bin/bash

echo


if test −z $1
then
  echo "No command−line arguments."
else
  echo "First command−line argument is $1."
fi

# Both code blocks are functionally identical.

if [ −z $1 ]
# if [ −z $1
# also works, but outputs an error message.
then
  echo "No command−line arguments."
else
  echo "First command−line argument is $1."
fi


echo

exit 0
```

3.5.1. File test operators

Returns true if...

−e

file exists.

−f

file is a regular file.

−s

file is not zero size.

−d

file is a directory.

−r

file is readable (has read permission).

−w

file has write permission.

−x

file has execute permission.

−g

group−id flag set on file.

−u

user−id flag set on file.

−O

you are owner of file

−G

gid of file same as yours

f1 −nt f2

file `f1` is newer than `f2`

f1 −ot f2

file `f1` is older than `f2`

!"not", reverses the sense of the tests above (returns true if condition absent).

Example 3−8. Tests, command chaining, redirection

```
#!/bin/bash

# This line is a comment.

filename=sys.log

if [ ! −f $filename ]
then
  touch $filename; echo "Creating file."
else
  cat /dev/null 62; $filename; echo "Cleaning out file."
fi

# Of course, /var/log/messages must have
# world read permission (644) for this to work.
tail /var/log/messages 62; $filename
echo "$filename contains tail end of system log."

exit 0
```

3.5.2. Comparison operators (binary)

integer comparison

−eq

is equal to (**`$a −eq $b`**)

−ne

is not equal to (**`$a −ne $b`**)

−gt

is greater than (**`$a −gt $b`**)

−ge

is greater than or equal to (**`$a −ge $b`**)

−lt

is less than (**`$a −lt $b`**)

−le

is less than or equal to (**`$a −le $b`**)

string comparison

=

is equal to (**`$a = $b`**)

!=

is not equal to (**`$a != $b`**)

−z

string is "null", that is, has zero length

−n

string in not "null". Note that this test does *not* work reliably (a bash bug?). Use **`! −z`** instead.

Example 3−9. arithmetic and string comparisons

```
#!/bin/bash

a=4
b=5

# Here a and b can be treated either as integers or strings.
# There is some blurring between the arithmetic and integer comparisons.
# Be careful.

if [ $a −ne $b ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo

if [ $a != $b ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
fi

echo

exit 0
```

Example 3−10. zmost

```
#!/bin/bash

#View gzipped files with 'most'

NOARGS=1
```

```
if [ $# = 0 ]
# same effect as:  if [ −z $1 ]
then
  echo "Usage: `basename $0` filename" 62;38;2
  # Error message to stderr.
  exit $NOARGS
  # Returns 1 as exit status of script
  # (error code)
fi

filename=$1

if [ ! −f $filename ]
then
  echo "File $filename not found!" 62;38;2
  # Error message to stderr.
  exit 2
fi

if [ ${filename##*.} != "gz" ]
# Using bracket in variable substitution.
then
  echo "File $1 is not a gzipped file!"
  exit 3
fi

zcat $1 | most

exit 0

# Uses the file viewer 'most'
# (similar to 'less')
```

3.6. Operations

=

All−purpose assignment operator, which works for both arithmetic and string assignments.
```
var=27 category=minerals
```
May also be used in a string comparison test.
```
if [ $string1 = $string2 ] then    command fi
```

The following are normally used in combination with **expr** or **let**.

arithmetic operators

+

plus

−

minus

*

multiplication

/

division

%

modulo, or mod

+=

"plus−equal" (increment variable by a constant)
`` `expr $var+=5` `` results in `var` being incremented by `5`.

−=

"minus−equal" (decrement variable by a constant)

*=

"times−equal" (multiply variable by a constant)
`` `expr $var*=4` `` results in `var` being multiplied by `4`.

/=

"slash−equal" (divide variable by a constant)

The bitwise logical operators seldom make an appearance in shell scripts. Their chief use seems to be manipulating and testing values read from ports or sockets. "Bit flipping" is more relevant to compiled languages, such as C and C++, which run fast enough to permit its use on the fly.

<<

bitwise left shift (multiplies by `2` for each shift position)

<<=

"left−shift−equal"
`let "var <<= 2"` results in `var` left−shifted `2` bits (multiplied by `4`)

>>

bitwise right shift (divides by `2` for each shift position)

>>=

"right−shift−equal" (inverse of <<=)

&

bitwise and

&=

"bitwise and−equal"

|

bitwise OR

|=

"bitwise OR−equal"

~

bitwise negate

!

bitwise NOT

^

bitwise XOR

^=

"bitwise XOR−equal"

relational tests

<

less than

>

greater than

<=

less than or equal to

>=

greater than or equal to

==

equal to (test)

!=

not equal to

&&

and (logical)
```
if [ $condition1 38;38; $condition2 ] # if both condition1 and condition2 hold true.
```

||

or (logical)
```
if [ $condition1 || $condition2 ] # if both condition1 or condition2 hold true...
```

3.7. Variables Revisited

Internal (builtin) variables

environmental variables affecting bash script behavior

`$IFS`

input field separator
This defaults to white space, but may be changed, for example, to parse a comma−separated data file.

`$HOME`

home directory of the user (usually `/home/username`)

`$PATH`

path to binaries (usually `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, etc.)
Note that the "working directory", `./`, is usually omitted from the `$PATH` as a security measure.

`$PS1`

prompt

`$PS2`

secondary prompt

`$PWD`

working directory (directory you are in at the time)

`$EDITOR`

the default editor invoked by a script, usually **vi** or **emacs**.

`$BASH`

the path to the **bash** binary itself, usually `/bin/bash`

`$BASH_ENV`

an environmental variable pointing to a bash startup file to be read when a script is invoked

`$0`, `$1`, `$2`, etc.

positional parameters (passed from command line to script, passed to a function, or **set** to a variable)

$#

number of command line arguments or positional parameters

$$

process id of script, often used in scripts to construct temp file names

$?

exit status of command, function, or the script itself

*$**

All of the positional parameters

$@

Same as $*, but each parameter is a quoted string, that is, the parameters are passed on intact, without interpretation or expansion

$−

Flags passed to script

$!

PID of last job run in background

=

variable assignment (*no space before & after*)
Do not confuse this with == and −eq, which test, rather than assign!

Example 3−11. Variable Assignment

```
#!/bin/bash

#When is a variable "naked", i.e., lacking the '$' in front?

# Assignment
a=879
echo $a

# Assignment using 'let'
let a=16+5
echo $a

# In a 'for' loop (really, a type of disguised assignment)
for a in 7 8 9 11
```

```
do
  echo $a
done

exit 0
```

Example 3−12. Variable Assignment, plain and fancy

```
#!/bin/bash

a=23
# Simple case
echo $a
b=$a
echo $b

# Now, getting a little bit fancier...

a=`echo Hello!`
# Assigns result of 'echo' command to 'a'
echo $a

a=`ls −l`
# Assigns result of 'ls −l' command to 'a'
echo $a

exit 0
```

Variable assignment using the $() mechanism (a newer method than using back quotes)

```

# From /etc/rc.d/rc.local
R=$(cat /etc/redhat−release)
arch=$(uname −m)
```

local variables

variables visible only within a code block or function (see Section 3.16)

environmental variables

variables that affect the behavior of the shell and user interface, such as the path and the prompt If a script sets environmental variables, they need to be "exported", that is, reported to the environment itself. This is the function of the **export** command.

`$0`, `$1`, `$2`, `$3`, etc.

positional parameters ($0 is the name of the script itself)
Example 3−13. Positional Parameters
`#!/bin/bash  echo  echo The name of this script is $0 # Adds ./ for current directory echo`
Some scripts can perform different operations, depending on which name they are invoked by. For this to work, the script needs to check `$0`, the name it was invoked by. There also have to be symbolic links present to all the alternate names of the same script.
Example 3−14. wh, whois domain name lookup
`#!/bin/bash  # Does a 'whois domain−name' lookup # on any of 3 alternate servers: # ripe.n`

The **shift** command reassigns the positional parameters, in effect shifting them to the left one notch.
`$1` <−−− `$2`, `$2` <−−− `$3`, `$3` <−−− `$4`, etc.
The old `$1` disappears, but `$0` does not change. If you use a large number of positional parameters to a script, **shift** lets you access those past `10`.
Example 3−15. Using shift
```
#!/bin/bash  # Name this script something like shift000, # and invoke it with some paramete
```

3.7.1. Typing variables: declare or typeset

The **declare** or **typeset** keywords (they are exact synonyms) permit restricting the properties of variables. This is a very weak form of the typing available in certain programming languages. The **declare** command is not available in version 1 of **bash**.

−rreadonly

```
declare −r var1
```
(**`declare −r var1`** works the same as **`readonly var1`**)
This is the rough equivalent of the C **const** type qualifier. An attempt to change the value of a readonly variable fails with an error message.

−iinteger

```
declare −i var2
```
The script treats subsequent occurences of `var2` as an integer. Note that certain arithmetic operations are permitted for declared integer variables without the need for **expr** or **let**.

−aarray

```
declare −a indices
```
The variable `indices` will be treated as an array.

−ffunctions

```
declare −f  # (no arguments)
```
A **`declare −f`** line within a script causes a listing of all the functions contained in that script.

−xexport

```
declare −x var3
```
This declares a variable as available for exporting outside the environment of the script itself.

Example 3−16. Using declare to type variables

```
#!/bin/bash

declare −f
# Lists the function below.
```

```
func1 ()
{
echo This is a function.
}

declare −r var1=13.36
echo "var1 declared as $var1"
# Attempt to change readonly variable.
var1=13.37
# Generates error message.
echo "var1 is still $var1"

echo

declare −i var2
var2=2367
echo "var2 declared as $var2"
var2=var2+1
# Integer declaration eliminates the need for 'let'.
echo "var2 incremented by 1 is $var2."
# Attempt to change variable declared as integer
echo "Attempting to change var2 to floating point value, 2367.1."
var2=2367.1
# results in error message, with no change to variable.
echo "var2 is still $var2"

exit 0
```

3.7.2. RANDOM: generate random integer

Example 3−17. Generating random numbers

```
#!/bin/bash

# Prints different random integer
# at each invocation.

a=$RANDOM
echo $a

exit 0
```

3.8. Loops

for (in)

This is the basic looping construct. It differs significantly from its C counterpart.
for [*`arg`*] in [*`list`*]
do
 `command`...
done
Note that *`list`* may contain wild cards.
Note further that if **do** is on same line as **for**, there needs to be a semicolon before list.
for [*`arg`*] in [*`list`*] ; do

Example 3−18. Simple for loops

```
#!/bin/bash

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet
done

echo

# Entire 'list' enclosed in quotes creates a single variable.
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
do
  echo $planet
done

exit 0
```

Omitting the **`in [list]`** part of a **for** loop causes the loop to operate on $#, the list of arguments given on the command line to the script.

Example 3−19. Missing `in [list]` in a for loop

```
#!/bin/bash

# Invoke both with and without arguments,
# and see what happens.

for a
do
 echo $a
done

# 'in list' missing, therefore
# operates on '$#'
# (command−line argument list)

exit 0
```

Example 3−20. Using efax in batch mode

```
#!/bin/bash

if [ $# −ne 2 ]
# Check for proper no. of command line args.
then
   echo "Usage: `basename $0` phone# text−file"
   exit 1
fi


if [ ! −f $2 ]
then
  echo "File $2 is not a text file"
  exit 2
fi


# Create fax formatted files from text files.
fax make $2

for file in $(ls $2.0*)
# Concatenate the converted files.
# Uses wild card in variable list.
do
  fil="$fil $file"
done

# Do the work.
efax −d /dev/ttyS3 −o1 −t "T$1" $fil

exit 0
```

while

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true.

while [*condition*]
do
 command...
done

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

while [*condition*] ; do

Note that certain specialized **while** loops, as, for example, a **getopts** construct, deviate somewhat from the standard template given here.

Example 3−21. Simple while loop

```
#!/bin/bash  var0=0  while [ "$var0" −lt 10 ] do   echo −n "$var0 "   # −n suppresses newl
```

Example 3−22. Another while loop

```
#!/bin/bash  while [ "$var1" != end ] do   echo "Input variable #1 "   echo "(end to exit)
```

until

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of **while** loop).

until [*condition−is−true*]

do
 command...
done
Note that an **until** loop tests for the terminating condition at the top of the loop, differing from a similar construct in some programming languages.
As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.
until [*condition−is−true*] ; do
Example 3−23. until loop

```
#!/bin/bash  until [ "$var1" = end ] # Tests condition at top of loop. do   echo "Input va
```

break*, *continue

The **break** and **continue** loop control commands correspond exactly to their counterparts in other programming languages. The **break** command terminates the loop (breaks out of it), while **continue** causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.
Example 3−24. Effects of break and continue in a loop

```
#!/bin/bash  echo echo Printing Numbers 1 through 20.  a=0  while [ $a −le 19 ]  do  a=$((
```

case (in) / esac

The **case** construct is the shell equivalent of **switch** in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.
case "$*variable*" in
 "$*condition1*")
 command...
 ;;
 "$*condition2*")
 command...
 ;;
 esac
Note:
Quoting the variables is recommended.
Each test line ends with a left paren).
Each condition block ends with a *double* semicolon ;;.
The entire **case** block terminates with an **esac** (*case* spelled backwards).
Example 3−25. Using case

```
#!/bin/bash  echo echo "Hit a key, then hit return." read Keypress  case "$Keypress" in
```

Example 3−26. Creating menus using case

```
#!/bin/bash  # Crude rolodex−type database  clear # Clear the screen.  echo "          Con
```

select

The **select** construct, adopted from the Korn Shell, is yet another tool for building menus.
select*variable* [in *list*]
do
 command...
 break
done

This prompts the user to enter one of the choices presented in the variable list. Note that **select** uses the `PS3` prompt (`#? `) by default, but that this may be changed.

Example 3−27. Creating menus using select

```
#!/bin/bash  PS3='Choose your favorite vegetable: ' # Sets the prompt string.  echo  selec
```

If ***in list*** is omitted, then **select** uses the list of command line arguments (`$@`) passed to the script or to the function in which the **select** construct is embedded. (Compare this to the behavior of a **for***`variable`* [in *`list`*] construct with the ***in list*** omitted.)

Example 3−28. Creating menus using select in a function

```
#!/bin/bash  PS3='Choose your favorite vegetable: '  echo  choice_of() { select vegetable
```

3.9. Internal Commands and Builtins

A *builtin* is a command contained in the bash tool set, literally built in.

getopts

This powerful tool parses command line arguments passed to the script. This is the bash analog of the **getopt** library function familiar to C programmers. It permits passing and concatenating multiple flags[1] and options to a script (for example **`scriptname -abc -e /usr/local`**).
The **getopts** construct uses two implicit variables. `$OPTIND` is the argument pointer (*OPTion INDex*) and `$OPTARG` (*OPTion ARGumnet*) the (optional) argument attached to a flag. A colon following the flag name in the declaration tags that flag as having an option.
A **getopts** construct usually comes packaged in a **while** loop, which processes the flags and options one at a time, then decrements the implicit `$OPTIND` variable to step to the next.

Note:

1. The arguments must be passed from the command line to the script preceded by a minus (`-`) or a plus (`+`), else **getopts** will not process them, and will, in fact, terminate option processing at the first argument encountered lacking these modifiers.
2. The **getopts** template differs slightly from the standard **while** loop, in that it lacks condition brackets.
3. The **getopts** construct replaces the obsolete **getopt** command.

```
while getopts ":abcde:fg" Option
# Initial declaration.
# a, b, c, d, e, f, and g are the flags expected.
# The : after flag 'e' shows it will have an option passed with it.
do
  case $Option in
    a ) # Do something with variable 'a'.
    b ) # Do something with variable 'b'.
    ...
    e)  # Do something with 'e', and also with $OPTARG,
        # which is the associated argument passed with 'e'.
    ...
    g ) # Do something with variable 'g'.
  esac
done
shift $(($OPTIND - 1))
# Move argument pointer to next.
# All this is not nearly as complicated as it looks 60;grin62;.
```

Example 3−29. Using getopts to read the flags/options passed to a script

```
#!/bin/bash

# 'getopts' processes command line args to script.

# Usage: scriptname −options
# Note: dash (−) necessary

# Try invoking this script with
# 'scriptname −mn'
# 'scriptname −oq qOption'
# (qOption can be some arbitrary string.)

OPTERROR=33

if [ −z $1 ]
# Exit and complain if no argument(s) given.
then
  echo "Usage: `basename $0` options (−mnopqrs)"
  exit $OPTERROR
fi

while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scenario #1: option −m−";;
    n | o ) echo "Scenario #2: option −$Option−";;
    p     ) echo "Scenario #3: option −p−";;
    q     ) echo "Scenario #4: option −q−, with argument \"$OPTARG\"";;
    # Note that option 'q' must have an additional argument,
    # otherwise nothing happens.
    r | s ) echo "Scenario #5: option −$Option−"'';;
    *     ) echo "Unimplemented option chosen.";;
  esac
done

shift $(($OPTIND − 1))
# Decrements the argument pointer
# so it points to next argument.

exit 0
```

exit

Unconditionally terminates a script. The **exit** command may optionally take an integer argument, which is returned to the shell as the *exit status* of the script. It is a good practice to end all but the simplest scripts with an **`exit 0`**, indicating a successful run.

set

The **set** command changes the value of internal script variables. One use for this is to toggle flags which help determine the behavior of the script (see Section 3.22). Another application for it is to reset the positional parameters that a script sees as the result of a command (**`set `command``**). The script can then parse the fields of the command output.

Example 3−30. Using set with positional parameters

```
#!/bin/bash  # script "set−test"  # Invoke this script with three command line parameters,
```

unset

The **unset** command deletes an internal script variable. It is a way of negating a previous **set**. Note that this command does not affect positional parameters.

readonly

Same as **`declare −r`**, sets a variable as read−only, or, in effect, as a constant. Attempts to change the variable fail with an error message. This is the shell analog of the C language **const** type qualifier.

basename

Strips the path information from a file name, printing only the file name. The construction **`basename $0`** lets the script know its name, that is, the name it was invoked by. This can be used for "usage" messages if, for example a script is called with missing arguments:

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Strips the **basename** from a file name, printing only the path information.
Note: basename and **dirname** can operate on any arbitrary string. The filename given as an argument does not need to refer to an existing file.

Example 3−31. basename and dirname

```
#!/bin/bash  a=/home/heraclius/daily−journal.txt  echo "Basename of /home/heraclius/daily−
```

read

"Reads" the value of a variable from stdin, that is, interactively fetches input from the keyboard. The `−a` option lets **read** get array variables (see Example 3−63).

Example 3−32. Variable assignment, using read

```
#!/bin/bash  echo −n "Enter the value of variable 'var1': " # −n option to echo suppresses
```

true

A command that returns a successful (zero) exit status, but does nothing else.

```
# Endless loop while true # alias for : do    operation−1    operation−2    ...    operati
```

false

A command that returns an unsuccessful exit status, but does nothing else.

```
# Null loop while false do    # The following code will not execute.    operation−1    ope
```

factor

Factor an integer into prime factors.

```
bash$ factor 27417R7417: 3 13 19 37
```

hash [cmds]

Record the path name of specified commands (in the shell hash table), so the shell or script will not need to search the `$PATH` on subsequent calls to those commands. When **hash** is called with no arguments, it simply lists the commands that have been hashed.

pwd

Print Working Directory. This gives the user's (or script's) current directory.

pushd, ***popd***, ***dirs***

This command set is a mechanism for bookmarking working directories, a means of moving back and forth through directories in an orderly manner. A pushdown stack is used to keep track of directory names. Options allow various manipulations of the directory stack.
`pushd dir−name` pushes the path *`dir−name`* onto the directory stack and simultaneously changes the current working directory to *`dir−name`*
popd removes (pops) the top directory path name off the directory stack and simultaneously changes the current working directory to that directory popped from the stack.
dirs lists the contents of the directory stack. A successful **pushd** or **popd** will automatically invoke **dirs**.
Scripts that require various changes to the current working directory without hard−coding the directory name changes can make good use of these commands. Note that the implicit `DIRSTACK` array variable, accessible from within a script, holds the contents of the directory stack.
Example 3−33. Changing the current working directory

```
#!/bin/bash  dir1=/usr/local dir2=/var/spool  pushd $dir1 # Will do an automatic 'dirs' #
```

source, . (**dot** *command*), ***dirs***

This command, when invoked from the command line, executes a script. Within a script, a **`source file−name`** loads the file *`file−name`*. This is the shell scripting equivalent of a C/C++ **`#include`** directive. It is useful in situations when multiple scripts use a common data file or function library.
Example 3−34. "Including" a data file

```
#!/bin/bash  # Load a data file. . data−file # Same effect as "source data−file"  # Note t
```

File `data−file` for Example 3−34, above. Must be present in same directory.

```
# This is a data file loaded by a script. # Files of this type may contain variables, func
```

3.9.1. Job Control Commands

wait

Stop script execution until all jobs running in background have terminated, or until the job number specified as an option terminates.

Example 3−35. Waiting for a process to finish before proceeding

```
#!/bin/bash

if [ −z $1 ]
```

```
then
  echo "Usage: `basename $0` find−string"
  exit 1
fi

echo "Updating 'locate' database..."
echo "This may take a while."
updatedb /usr 38;
# Must be run as root.

wait
# Don't run the rest of the script
# until 'updatedb' finished.
# In this case, you want the the database updated
# before looking up the file name.

locate $1


exit 0
```

suspend

This has the same effect as **Control−Z**, pausing a foreground job.

stop

This has the same effect as **suspend**, but for a background job.

disown

Remove job(s) from the shell's table of active jobs.

jobs

Lists the jobs running in the background, giving the job number. Not as useful as **ps**.

times

Gives statistics on the system time used in executing commands, in the following form: `0m0.020s 0m0.020s` This capability is of very limited value, since it is uncommon to profile and benchmark shell scripts.

kill

Forcibly terminate a process. Note that **`kill -l`** lists all the "signals".

3.10. External Filters, Programs and Commands

This is a descriptive listing of standard UNIX commands useful in shell scripts.

ls

The basic file "list" command. It is all too easy to underestimate the power of this humble command. For example, using the `−R`, recursive option, **ls** provides a tree−like listing of a directory structure.

Example 3−36. Using ls to create a table of contents for burning a CDR disk

```
#!/bin/bash

# Script to automate burning a CDR.

# Uses Joerg Schilling's "cdrecord" package
# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

# If this script invoked as an ordinary user, need to suid cdrecord
# (chmod u+s /usr/bin/cdrecord, as root).

if [ −z $1 ]
then
  IMAGE_DIRECTORY=/opt
# Default directory, if not specified on command line.
else
    IMAGE_DIRECTORY=$1
fi

ls −lR $IMAGE_DIRECTORY 62; $IMAGE_DIRECTORY/contents
echo "Creating table of contents."

mkisofs −r −o cdimage.iso $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image (cdimage.iso)."

cdrecord −v −isosize speed=2 dev=0,0 cdimage.iso
# Change speed parameter to speed of your burner.
echo "Burning the disk."
echo "Please be patient, this will take a while."

exit 0
```

chmod

Changes the attributes of a file.
```
chmod +x filename # Makes "filename" executable for all users.
chmod 644 filename # Makes "filename" readable/writable to owner, readable to # others # (
chmod 1777 directory−name # Gives everyone read, write, and execute permission in director
```

umask

Set the default file attributes (for a particular user).

find

exec *`COMMAND`*
Carries out *`COMMAND`* on each file that **find** scores a hit on. *`COMMAND`* is followed by {} \; (the ; is escaped to make certain the shell reads it literally and terminates the command sequence). This causes *`COMMAND`* to bind to and act on the path name of the files found (see Example 3−53)

xargs

A filter for feeding arguments to a command, and also a tool for assembling the commands themselves. It breaks a data stream into small enough chunks for filters and commands to process. Consider it as a powerful replacement for backquotes. In situations where backquotes fail with a too many arguments error, substituting **xargs** often works. Normally, xargs reads from 'stdin' or from a pipe, but it can also be given the output of a file.
`ls | xargs -p -l gzip` gzips every file in current directory, one at a time, prompting before each operation.
One of the more interesting xargs options is `-n` *`XX`*, which limits the number of arguments passed to *`XX`*.
`ls | xargs -n 8 echo` lists the files in the current directory in 8 columns.
Example 3−37. Log file using xargs to monitor system log
```
#!/bin/bash  # Generates a log file in current directory # from the tail end of /var/log me
```
Example 3−38. copydir, copying files in current directory to another, using xargs
```
#!/bin/bash  # Copy (verbose) all files in current directory # to directory specified on co
```

`eval arg1, arg2, ...`

Translates into commands the arguments in a list (useful for code generation within a script).
Example 3−39. Showing the effect of eval
```
#!/bin/bash  y=`eval ls −l` echo $y  y=`eval df` echo $y # Note that LF's not preserved  ex
```
Example 3−40. Forcing a log−off
```
#!/bin/bash  y=`eval ps ax | sed −n '/ppp/p' | awk '{ print $1 }'` # Finding the process nu
```

`expr arg1 operation arg2 ...`

All−purpose expression evaluator: Concatenates and evaluates the arguments according to the operation given (arguments must be separated by spaces). Operations may be arithmetic, comparison, string, or logical.

`expr 3 + 5`

returns 8

`expr 5 % 3`

returns 2

``y=`expr $y + 1` ``

incrementing variable, same as **`let y=y+1`** and **`y=$(($y+1))`**, as discussed elsewhere

``z=`expr substr $string28 $position $length` ``

Note that external programs, such as **sed** and **Perl** have far superior string parsing facilities, and it might well be advisable to use them instead of the built−in bash ones.

Example 3−41. Using expr

```
#!/bin/bash

# Demonstrating some of the uses of 'expr'
# +++++++++++++++++++++++++++++++++++++++

echo

# Arithmetic Operators

echo Arithmetic Operators
echo
a=`expr 5 + 3`
echo 5 + 3 = $a

a=`expr $a + 1`
echo
echo a + 1 = $a
echo \(incrementing a variable\)

a=`expr 5 % 3`
# modulo
echo
echo 5 mod 3 = $a

echo
echo

# Logical Operators

echo Logical Operators
echo

a=3
echo a = $a
b=`expr $a \62; 10`
echo 'b=`expr $a \62; 10`, therefore...'
echo "If a 62; 10, b = 0 (false)"
echo b = $b

b=`expr $a \60; 10`
echo "If a 60; 10, b = 1 (true)"
echo b = $b


echo
echo

# Comparison Operators

echo Comparison Operators
echo
a=zipper
echo a is $a
if [ `expr $a = snap` ]
# Force re−evaluation of variable 'a'
then
```

```
   echo "a is not zipper"
fi

echo
echo

# String Operators

echo String Operators
echo

a=1234zipper43231
echo The string being operated upon is $a.
# index: position of substring
b=`expr index $a 23`
echo Numerical position of first 23 in $a is $b.
# substr: print substring, starting position 38; length specified
b=`expr substr $a 2 6`
echo Substring of $a, starting at position 2 and 6 chars long is $b.
# length: length of string
b=`expr length $a`
echo Length of $a is $b.
# 'match' operations similarly to 'grep'
b=`expr match $a [0−9]*`
echo Number of digits at the beginning of $a is $b.
b=`expr match $a '\([0−9]*\)'`
echo The digits at the beginning of $a are $b.

echo

exit 0
```

Note that : can substitute for **match**. **`b=`expr $a : [0−9]*``** is an exact equivalent of **`b=`expr match $a [0−9]*``** in the above example.

let

The **let** command carries out arithmetic operations on variables. In many cases, it functions as a less complex version of **expr**.

Example 3−42. Letting let do some arithmetic.

```
#!/bin/bash  echo  let a=11 # Same as 'a=11' let a=a+5 # Equivalent to let "a = a + 5" # (
```

printf

The **printf**, formatted print, command is an enhanced **echo**. It is a limited variant of the C language `printf`, and the syntax is somewhat different.

printf*`format−string`*... *`parameter`*...

See the **printf** man page for in−depth coverage.

Note: Older versions of **bash** may not support **printf**.

Example 3−43. printf in action

```
#!/bin/bash  # printf demo  PI=3.14159265358979 DecimalConstant=31373 Message1="Greetings,
```

at

The **at** job control command executes a given set of commands at a specified time. This is a user version of **cron**.

`at 2pm January 15` prompts for a set of commands to execute at that time.
Using the `−f` option, **at** reads a command list from a file, which can be useful in a non−interactive script.

ps

Lists currently executing jobs by owner and process id. This is usually invoked with `ax` options, and may be piped to **grep** to search for a specific process.
`ps ax | grep sendmail` results in:

```
295 ?        S      0:00 sendmail: accepting connections on port 25
```

batch

The **batch** job control command is similar to **at**, but it runs a command list when the system load drops below `.8`. Like **at**, it can read commands from a file with the `−f` option.

sleep

This is the shell equivalent of a wait loop. It pauses for a specified number of seconds, doing nothing. This can be useful for timing or in processes running in the background, checking for a specific event every so often.

```
sleep 3 # Pauses 3 seconds.
```

dd

This is the somewhat obscure and much feared "data duplicator" command. It simply copies a file (or stdin/stdout), but with conversions. Possible conversions are ASCII/EBCDIC, upper/lower case, swapping of byte pairs between input and output, and skipping and/or truncating the head or tail of the input file. A **`dd --help`** lists the conversion and other options that this powerful utility takes.
The **dd** command can copy raw data and disk images to and from devices, such as floppies. It can even be used to create boot floppies.
`dd if=kernel-image of=/dev/fd0H1440` One important use for **dd** is initializing temporary swap files (see Example 3−69).

sort

File sorter, often used as a filter in a pipe. See the man page for options.

diff

Simple file comparison utility. The files must be sorted (this may, if necessary be accomplished by filtering the files through **sort** before passing them to **diff**). **`diff file−1 file−2`** outputs the lines in the files that differ, with carets showing which file each particular line belongs to. A common use for **diff** is to generate difference files to be used with **patch** (see below). The `−e` option outputs files suitable for **ed** or **ex** scripts.

```
patch −p1 60;patch−file # Takes all the changes listed in 'patch−file' and applies them #
```

comm

Versatile file comparison utility. The files must be sorted for this to be useful.
comm ***`−optionsfirst−filesecond−file`***

`comm file−1 file−2` outputs three columns:
column 1 = lines unique to *`file−1`*
column 2 = lines unique to *`file−2`*
column 3 = lines common to both.
The options allow suppressing output of one or more columns.
`−1` suppresses column `1`
`−2` suppresses column `2`
`−3` suppresses column `3`
`−12` suppresses both columns `1` and `2`, etc.

uniq

This filter removes duplicate lines from a sorted file. It is often seen in a pipe coupled with sort.

```
cat list−1 list−2 list−3 | sort | uniq 62; final.list
```

expand

A filter than converts tabs to spaces, often seen in a pipe.

cut

A tool for extracting fields from files. It is similar to the **`print $N`** command set in **awk**, but more limited. It may be simpler to use **cut** in a script than **awk**. Particularly important are the `−d` (delimiter) and `−f` (field specifier) options.
Using **cut** to obtain a listing of the mounted filesystems:

```
cat /etc/mtab | cut −d ' ' −f1,2
```

Using **cut** to list the OS and kernel version:

```
uname −a | cut −d" " −f1,3,11,12
```

`cut −d ' ' −f2,3 filename` is equivalent to **`awk '{ print $2, $3 }' filename`**

colrm

Column removal filter. This removes columns (characters) from a file and writes them, lacking the specified columns, back to stdout. **`colrm 2 3 <filename`** removes the second and third characters from each line of the text file *`filename`*.

paste

Tool for merging together different files into a single, multi−column file. In combination with **cut**, useful for creating system log files.

join

Consider this a more flexible version of **paste**. It works on exactly two files, but permits specifying which fields to paste together, and in which order.

cpio

This specialized archiving copy command is rarely used any more, having been supplanted by **tar**/**gzip**. It still has its uses, such as moving a directory tree.
Example 3−44. Using cpio to move a directory tree

```
#!/bin/bash  # Copying a directory tree using cpio.  if [ $# −ne 2 ] then   echo Usage: `ba
```

cd

The familiar **cd** change directory command finds use in scripts where execution of a command requires being in a specified directory.
`(cd /source/directory 38;38; tar cf − . ) | (cd /dest/directory 38;38; tar xvfp −)` [from the previously cited example by Alan Cox]

touch

Utility for updating access/modification times of a file to current system time or other specified time, but also useful for creating a new file. The command **`touch zzz`** will create a new file of zero length, named `zzz`, assuming that `zzz` did not previously exist.

split

Utility for splitting a file into smaller chunks. Usually used for splitting up large files in order to back them up on floppies or preparatory to e−mailing or uploading them.

rm

Delete (remove) a file or files. When used with the recursive flag `−r`, this removes files all the way down the directory tree (very dangerous!).

ln

Creates links to pre−existings files. Most often used with the `−s`, symbolic or "soft" link flag. This permits referencing the linked file by more than one name and is a superior alternative to aliasing.

cp

This is the file copy command. **`cp file1 file2`** copies *`file1`* to *`file2`*, overwriting *`file2`* if it already exists.

mv

This is the file move command. It is equivalent to a combination of **cp** and **rm**. It may be used to move multiple files to a directory.

rcp

"Remote copy", copies files between two different networked machines. Using **rcp** and similar utilities with security implications in a shell script may not be advisable. Consider instead, using an **expect** script.

yes

In its default behavior the **yes** command feeds a continuous string of the character `y` followed by a line feed to stdout. A **control−c** terminates the run. A different output string may be specified, as in **`yes different string`**, which would continually output `different string` to stdout.

One might well ask the purpose of this. From the command line or in a script, the output of **yes** can be redirected or piped into a program expecting user input. In effect, this becomes a sort of poor man's version of **expect**.

echo

prints (to stdout) an expression or variable (`$variable`).

```
echo Hello echo $a
```

Normally, each **echo** command prints a terminal newline, but the `−n` option suppresses this.

cat**, **tac

cat, an acronym for *concatenate*, lists a file to stdout. When combined with redirection (> or >>), it is commonly used to concatenate files.

```
cat filename cat file.1 file.2 file.3 62; file.123
```

tac, is the inverse of *cat*, listing a file backwards from its end.

head

lists the first `10` lines of a file to stdout.

tail

lists the last `10` lines of a file to stdout. Commonly used to keep track of changes to a system logfile, using the `−f` option, which outputs lines appended to the file.

tee

[UNIX borrows an idea here from the plumbing trade.]
This is a redirection operator, but with a difference. Like the plumber's *tee*, it permits "siponing off" the output of a command or commands within a pipe, but without affecting the result. This is useful for printing an ongoing process to a file or paper, perhaps to keep track of it for debugging purposes.

```
                 tee                    |------62; to file                    |    ===========
cat listfile* | sort | tee check.file | uniq 62; result.file
```

(The file `check.file` contains the concatenated sorted "listfiles", before the duplicate lines are removed by **uniq**.)

sed**, **awk

manipulation scripting languages in order to parse text and command output

sed

Non−interactive "stream editor", permits using many **ex** commands in batch mode.

awk

Programmable file extractor and formatter, good for manipulating and/or extracting fields (columns) in text files. Its syntax is similar to C.

wc

wc gives a "word count" on a file or I/O stream.

```
%
wc /usr/doc/sed−3.02/READMER0     127     838 /usr/doc/sed−3.02/README [20 lines  127 word
```

`wc −w` gives only the word count.
`wc −l` gives only the line count.
`wc −c` gives only the character count.
`wc −L` gives only the length of the longest line.

tr

character translation filter.
Note: must use quoting and/or brackets, as appropriate.
`tr "A−Z" "*" <filename` changes all the uppercase letters in *`filename`* to asterisks (writes to stdout).
`tr −d [0−9] <filename` deletes all digits from the file *`filename`*.
Example 3−45. toupper: Transforms a file to all uppercase.

```
#!/bin/bash  # Changes a file to all uppercase.  if [ −z $1 ] # Standard check whether com
```

fold

A filter that wraps inputted lines to a specified width.

fmt

Simple−minded file formatter.

pr

Print formatting filter. This will paginate a file (or stdout) into sections suitable for hard copy printing. A particularly useful option is `−d`, forcing double−spacing.
Example 3−46. Formatted file listing.

```
#!/bin/bash  # Get a file listing...  b=`ls /usr/local/bin`  # ...40 columns wide. echo $b
```

date

Simply invoked, **date** prints the date and time to stdout. Where this command gets interesting is in its formatting and parsing options.
Example 3−47. Using date

```
#!/bin/bash  #Using the 'date' command  # Needs a leading '+' to invoke formatting.  echo
```

time

Outputs very verbose timing statistics for executing a command.
`time ls −l /` gives something like this:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)kPinputs+0output
```

grep

A multi−purpose file search tool that uses regular expressions. Originally a command/filter in the ancient **ed** line editor, **`g/re/p`**, or *global − regular expression − print*.

grep*`pattern`* [*`file`*...] search the files *`file`*, etc. for occurrences of *`pattern`*.
`ls −l | grep '.txt'` has the same effect as **`ls −l *.txt`**.

script

This utility records (saves to a file) all the user keystrokes at the command line in a console or an xterm window. This, in effect, create a record of a session.

tar

The standard UNIX archiving utility. Originally a *Tape ARchiving* program, from whence it derived its name, it has developed into a general purpose package that can handle all manner of archiving with all types of destination devices, ranging from tape drives to regular files to even stdout. GNU tar has long since been patched to accept **gzip** options, see below.

gzip

The standard GNU/UNIX compression utility, replacing the inferior and proprietary **compress**.

shar

Shell archiving utility. The files in a shell archive are concatenated without compression, and the resultant archive is essentially a shell script, complete with #!/bin/sh header, and containing all the necessary unarchiving commands. Shar archives still show up in Internet newsgroups, but otherwise **shar** has been pretty well replaced by **tar**/**gzip**. The **unshar** command unpacks **shar** archives.

file

A utility for identifying file types. The command **`file file−name`** will return a file specification for *`file−name`*, such as `ascii text` or `data`. It references the magic numbers found in `/usr/share/magic`, `/etc/magic`, or `/usr/lib/magic`, depending on the Linux/UNIX distribution.

uuencode

This utility encodes binary files into ASCII characters, making them suitable for transmission in the body of an e−mail message or in a newsgroup posting.

uudecode

This reverses the encoding, decoding uuencoded files back into the original binaries.
Example 3−48. uuencoding encoded files

```
#!/bin/bash  lines=35 # Allow 35 lines for the header (very generous).  for File in * # Te
```

more, ***less***

Pagers that display a text file or text streaming to stdout, one page at a time.

jot, ***seq***

These utilities emit a sequence of integers, with a user selected increment. This can be used to advantage in a **for** loop.

Example 3−49. Using seq to generate loop arguments

```
#!/bin/bash  for a in `seq 80` # Same as for a in 1 2 3 4 5 ... 80 (saves much typing!). #
```

3.11. System and Administrative Commands

The startup and shutdown scripts in `/etc/rc.d` illustrate the uses (and usefulness) of these comands. These are usually invoked by root and used for system maintenance or emergency filesystem repairs. Use with caution, as some of these commands may damage your system if misused.

uname

Output system specifications (OS, kernel version, etc.) to stdout. Invoked with the `−a` option, gives verbose system info.
`uname −a` outputs something like:

```
Linux localhost.localdomain 2.2.15−2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i586 unknown
```

env

Runs a program or script with certain environmental variables set or changed (without changing the overall system environment).

shopt

This command permits changing shell options on the fly. Works with version 2 of **bash** only.

```
shopt −s cdspell # Allows misspelling directory names with 'cd' command.
```

lockfile

This utility is part of the **procmail** package (www.procmail.org). It creates a *lock file*, a semaphore file that controls access to a file, device, or resource. The lock file serves as a flag that this particular file, device, or resource is in use by a particular process ("busy"), and permitting only restricted access (or no access) to other processes. Lock files are used in such applications as protecting system mail folders from simultaneously being changed by multiple users, indicating that a modem port is being accessed, and showing that an instance of Netscape is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts create a lock file that already exists, the script will likely hang.

cron

Administrative program scheduler, performing such duties as cleaning up and deleting system log files and updating the slocate database. This is the superuser version of **at**. It runs as a daemon (background process) and executes scheduled entries from `/etc/crontab`.

chroot

CHange ROOT directory. Normally commands are fetched from `$PATH`, relative to `/`, the default root directory. This changes the root directory to a different one (and also changes the working directory to there). A **`chroot /opt`** would cause references to `/usr/bin` to be translated to `/opt/usr/bin`, for example. This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those telnetting in, to a secured portion of the filesystem. Note that after a **chroot**, the execution path for system

binaries is no longer valid.
The **chroot** command is also handy when running from an emergency boot floppy (**chroot** to `/dev/fd0`), or as an option to **lilo** when recovering from a system crash. Other uses include installation from a different filesystem (an **rpm** option). Invoke only as root, and use with caution.

ldd

Show shared lib dependencies for an executable file.

```
bash$
ldd /bin/ls libc.so.6 =62; /lib/libc.so.6 (0x4000c000) /lib/ld−linux.so.2 =62; /lib/
```

who

Show all users logged on to the system.

w

Show all logged on users and the processes belonging to them. This is an extended version of **who**. The output of **w** may be piped to **grep** to find a specific user and/or process.

```
bash#
w | grep startx grendel  tty1     −                 4:22pm  6:41   4.47s  0.45s  sta
```

wall

This is an acronym for "write all", i.e., sending a message to all users every terminal logged on in the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem.

```
wall System going down for maintenance in 5 minutes!
```

fuser

Identifies the processes (by pid) that are accessing a given file, set of files, or directory.

logger

Appends a user−generated message to the system log (`/var/log/messages`).

```
logger Experiencing instability in network connection at 23:10, 05/21. # Now, do a '
```

free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using **grep**, **awk** or **Perl**.

```
bash$
free                 total       used       free     shared    buffers     cached
```

sync

Forces writing all updated data from buffers to hard drive. While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure. In the olden days, a **`sync sync`** was a useful precautionary measure before a system reboot.

init

The **init** command is the parent of all processes. Called in the final step of a bootup, **init** determines the runlevel of the system from `/etc/inittab`. Invoked by its alias **telinit**, and by root only.

telinit

Symlinked to **init**, this is a means of changing the system runlevel, usually done for system maintenance or emergency filesystem repairs. Invoked only by root. This command can be dangerous − be certain you understand it well before using!

runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel `0`), in single−user mode (`1`), in multi−user mode (`2` or `3`), in X Windows (`5`), or rebooting (`6`).

halt**, **shutdown**, **reboot

Command set to shut the system down, usually just prior to a power down.

exec

This is actually a system call that replaces the current process with a specified command. It is mostly seen in combination with **find**, to execute a command on the files found. When used as a standalone in a script, this forces an exit from the script when the **exec**'ed command terminates. An **exec** is also used to reassign file descriptors. **`exec <zzz−file`** replaces stdin with the file *`zzz−file`*.

Example 3−50. Effects of exec

```
#!/bin/bash

exec echo "Exiting $0."
# Exit from script.

# The following lines never execute.
echo "Still here?"

exit 0
```

ifconfig

Network interface configuration utility.

route

Show info about or make changes to the kernel routing table.

netstat

Show current network information and statistics, such as routing tables and active connections.

mknod

Creates block or character device files (may be necessary when installing new hardware on the system).

mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file `/etc/fstab` provides a handy listing of available filesystems, including options, that may be automatically or manually mounted. The file `/etc/mtab` shows the currently mounted filesystems (including the virtual ones, such as `/proc`).

```
mount −t iso9660 /dev/cdrom /mnt/cdrom # Mounts CDROM mount /mnt/cdrom # Shortcut, if /mnt
```

umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be **umount**'ed, else filesystem corruption may result.

```
umount /mnt/cdrom
```

lsmod

List installed kernel modules.

insmod

Force insertion of a kernel module. Must be invoked as root.

modprobe

Module loader that is normally invoked automatically in a startup script.

depmod

Creates module dependency file, usually invoked from startup script.

rdev

Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but **rdev** remains useful for setting up a ram disk. This is another dangerous command, if misused.

Using our knowledge of administrative commands, let us examine a system script. One of the shortest and simplest to understand scripts is **killall**, used to suspend running processes at system shutdown.

Example 3−51. killall, from `/etc/rc.d/init.d`

```
#!/bin/sh

# −−62; Comments added by the author of this HOWTO marked by "−−62;".
# −−62; This is part of the 'rc' script package
# −−62; by Miquel van Smoorenburg, 60;miquels@drinkel.nl.mugnet.org62;


# Bring down all unneeded services that are still running (there shouldn't
# be any, so this is just a sanity check)

for i in /var/lock/subsys/*; do
        # −−62; Standard for/in loop, but since "do" is on same line,
        # −−62; it is necessary to add ";".
        # Check if the script is there.
        [ ! −f $i ] 38;38; continue
        # −−62; This is a clever use of an "and list", equivalent to:
        # −−62; if [ ! −f $i ]; then continue

        # Get the subsystem name.
        subsys=${i#/var/lock/subsys/}
        # −−62; Match variable name, which, in this case, is the file name.
        # −−62; This is the exact equivalent of subsys=`basename $i`.

        # −−62; It gets it from the lock file name, and since if there
        # −−62; is a lock file, that's proof the process has been running.
        # −−62; See the "lockfile" entry, above.


        # Bring the subsystem down.
        if [ −f /etc/rc.d/init.d/$subsys.init ]; then
            /etc/rc.d/init.d/$subsys.init stop
        else
            /etc/rc.d/init.d/$subsys stop
        # −−62; Suspend running jobs and daemons
        # −−62; using the 'stop' shell builtin.
        fi
done
```

That wasn't so bad. Aside from a little fancy footwork with variable matching, there is no new material there.

Exercise. In `/etc/rc.d/init.d`, analyze the **halt** script. It is a bit longer than **killall**, but similar in concept. Make a copy of this script somewhere in your home directory and experiment with it (do *not* run it as root). Do a simulated run with the `−vn` flags (**`sh −vn scriptname`**). Add extensive comments. Change the "action" commands to "echos".

Now, look at some of the more complex scripts in `/etc/rc.d/init.d`. See if you can understand parts of them. Follow the above procedure to analyze them.

For those scripts needing a single do−it−all tool, a Swiss army knife, there is Perl. Perl combines the capabilities of **sed**, **awk**, and throws in a large subset of C, to boot. It is modular and contains support for everything ranging from object oriented programming up to and including the kitchen sink. Short Perl scripts can be effectively embedded in shell scripts, and there may even be some substance to the claim that Perl can totally replace shell scripting.

Example 3−52. Perl embedded in a bash script

```
#!/bin/bash

perl −e 'print "This is an embedded Perl script\n"'
```

```
# Some shell commands may follow.

exit 0
```

3.12. Backticks (`...`)

Command substitution

Use the output of the command within backticks as arguments to another to generate command line text.
`` rm `cat filename` `` (where *`filename`* contains list of files to delete)

Incrementing / decrementing variables

`` z=`expr $z + 3` `` Note that this use of backticks has been superseded by double parentheses **`$((...))`** or the **let** construction.
`z=$(($z+3))` or
`let z=z+3`
`let "z += 3"`

Example 3−53. Badname, eliminate file names in current directory containing bad characters and white space.

```
#!/bin/bash

# Delete filenames in current directory containing bad characters.

for filename in *
do
badname=`echo "$filename" | sed −n /[\+\{\;\"\\\=\?~\(\)\60;\62;\38;\*\|\$]/p`
# Files containing those nasties:   + { ; " \ = ? ~ ( ) 60; 62; 38; * | $
rm $badname 262;/dev/null
#           So error messages deep−sixed.
done

# Now, take care of files containing all manner of whitespace.
find . −name "* *" −exec rm −f {} \;
# The "{}" references the paths of all the files that "find" finds.
# The '\' ensures that the ';' is interpreted literally, as end of command.

exit 0
```

−

Where file name expected, redirects output to stdout (mostly seen with **`tar cf`**)
Example 3−54. Backup of all files changed in last day
`#!/bin/bash  # Backs up all files in current directory # modified within last 24 hours # i`

3.13. I/O Redirection

There are always three default "files" open, stdout (the screen), stderr (the screen, also) and stdin (the keyboard).

```
62;
62;62;
262;38;1
```

n<&−

close input file descriptor *n*

<&−

close stdin

n>&−

close output file descriptor *n*

>&−

close stdout

Recess Time

A bizarre little intermission whose purpose is to give the reader a chance to catch his/her breath and maybe giggle a little.

Fellow Linux user, greetings! You are reading a something which will bring you luck and good fortune. Just e−mail ten copies of this document to ten of your friends. Before you make the copies, send a 100−line 'bash' script to the first person on the list given at the bottom of this letter. Then delete their name and add yours to the bottom of the list.

Don't break the chain! Make the copy within 48 hours. Wilfred P. of Houston failed to send out his ten copies and woke the next morning to find his job description changed to "COBOL programmer." Howard L. of Newport News sent out his ten copies and within a month had enough hardware and software to build a 100−node Beowulf cluster dedicated to playing 'xbill'. Amelia V. of Chicago laughed at this letter and broke the chain. Shortly thereafter, a fire broke out in her terminal and she now spends her days writing documentation for MS Windows.

Don't break the chain! Send out your ten copies today!
−−Courtesy 'NIX "fortune cookies", with a few alterations and many apologies

3.14. Regular Expressions

In order to fully utilize the power of shell scripting, you need to master regular expressions.

3.14.1. A Brief Introduction to Regular Expressions

An expression is simply a set of characters that has an interpretation above and beyond its literal meaning. A quote symbol ("), for example, may denote speech by a character, ditto, or a meta−meaning for the symbols that follow. Regular expressions are a set of characters that UNIX endows with special features.

The main uses for regular expressions (REs) are text searches and manipulation. An RE *matches* a single character or a set of characters.

- The asterisk * matches any number of characters, including zero.
- The dot . matches any one character, except a newline.
- The question mark ? matches zero or one of the previous RE.
- The plus + matches one or more of the previous RE.
- The caret ^ matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.
- The dollar sign $ at the end of a an RE matches the end of a line.
- Brackets [] enclose a set of characters to match in a single RE.
- The backslash \ escapes a special character.

See "Sed & Awk", by Dougherty and Robbins (see *Bibliography*) for a complete treatment of REs.

3.14.2. Using REs in scripts

Sed, awk, and Perl, used as filters in scripts, take REs as arguments when "sifting" or transforming files or I/O streams.

3.15. Subshells

- ()
- {}

3.16. Functions

Like "real" programming languages, **bash** has functions, though in a somewhat limited implementation. A function is a subroutine, a code block that implements a set of operations. Whenever there is repetitive code, when a task repeats with only slight variations, then writing a function should be investigated.

function*`function−name`* {
`command`...
}
or

`function−name` () {
`command`...
}

The second form will cheer the hearts of C programmers.

The opening bracket in the function may optionally be placed on the second line, to more nearly resemble C function syntax.

`function−name` ()
{
`command`...
}

Functions are called, *triggered*, simply by invoking their names.

Note that a function itself must precede the first call to it. There is no method of "declaring" the function, as, for example, in C.

Example 3−55. Simple function

```
#!/bin/bash

funky ()
{
  echo This is a funky function.
  echo Now exiting funky function.
}

# Note: function must precede call.

# Now, call the function.

funky

exit 0
```

More complex functions may have arguments passed to them and may return exit values to the script for further processing.

```
function−name $arg1 $arg2
```

The function refers to the passed arguments by position (as if they were positional parameters), that is, `$1`, `$2`, etc.

Example 3−56. Positional Parameters

```
#!/bin/bash

func2 () {
   if [ −z $1 ]
   # Checks if any params.
   then
     echo "No parameters passed to function."
     return 0
   else
     echo "Param #1 is $1."
   fi

   if [ $2 ]
   then
     echo "Parameter #2 is $2."
   fi
}

func2
# Called with no params
echo

func2 first
# Called with one param
echo

func2 first second
# Called with two params
echo

exit 0
```

exit status

Functions return a value, called an *exit status*. The exit status may be explicitly specified by a **return** statement, otherwise it is the exit status of the last command in the function (0 if successful, and a non−zero error code if not). This exit status may be used in the script by referring to as `$?`.

return

Terminates a function. The **return** statement may optionally take an integer argument, which is returned to the calling script as the "exit status" of the function, and this exit status is assigned to the variable `$?`.

Example 3−57. Converting numbers to Roman numerals

```
#!/bin/bash

# Arabic number to Roman numeral conversion
# Range 0 − 200
# It's crude, but it works.

# Extending the range and otherwise improving the script
# is left as an exercise for the reader.

# Usage: roman number−to−convert

ARG_ERR=1
OUT_OF_RANGE=200

if [ −z $1 ]
then
  echo "Usage: `basename $0` number−to−convert"
  exit $ARG_ERR
fi

num=$1
if [ $num −gt $OUT_OF_RANGE ]
then
  echo "Out of range!"
  exit $OUT_OF_RANGE
fi

to_roman ()
{
number=$1
factor=$2
rchar=$3
let "remainder = number − factor"
while [ $remainder −ge 0 ]
do
  echo −n $rchar
  let "number −= factor"
  let "remainder = number − factor"
done

return $number
}

# Note: must declare function
#       before first call to it.

to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
num=$?
```

```
to_roman $num 1 I

echo

exit 0
```

local variables

A variable declared as *local* is one that is visible only within the block of code in which it appears. In a shell script, this means the variable has meaning only within the function it is internal to.

Example 3−58. Local variable visibility

```
#!/bin/bash  func () {   local a=23   echo   echo "a in function is $a"   echo }   func  
```

Local variables permit recursion (a recursive function is one that calls itself), but this practice can involve much computational overhead and is definitely *not* recommended in a shell script.

Example 3−59. Recursion, using a local variable

```
#!/bin/bash  # Does bash permit recursion? # Well, yes, but... # You gotta have rocks in y
```

3.17. List Constructs

The "and list" and "or list" constructs provide a means of processing a number of commands consecutively. These can effectively replace complex nested **if**/**then** or even **case** statements. Note that the exit status of an "and list" or an "or list" is the exit status of the last command executed.

and list

`command−1 38;38; command−2 38;38; command−3 38;38; ... command−n` Each command executes in turn provided that the previous command has given a return value of true. At the first false return, the command chain terminates (the first command returning false is the last one to execute).

Example 3−60. Using an "and list" to test for command−line arguments

```
#!/bin/bash

# "and list"

if [ ! −z $1 ] 38;38; echo "Argument #1 = $1" 38;38; [ ! −z $2 ] 38;38; echo "Argument #2 = $2"
then
  echo "At least 2 arguments to script."
  # All the chained commands return true.
else
  echo "Less than 2 arguments to script."
  # At least one of the chained commands returns false.
fi
# Note that "if [ ! −z $1 ]" works, but its supposed equivalent,
# "if [ −n $1 ]" does not. This is a bug, not a feature.


# This accomplishes the same thing, coded using "pure" if/then statements.
if [ ! −z $1 ]
then
  echo "Argument #1 = $1"
fi
if [ ! −z $2 ]
then
  echo "Argument #2 = $2"
  echo "At least 2 arguments to script."
else
  echo "Less than 2 arguments to script."
fi
# It's longer and less elegant than using an "and list".


exit 0
```

or list

`command−1 || command−2 || command−3 || ... command−n` Each command executes in turn for as long as the previous command returns false. At the first true return, the command chain terminates (the first command returning true is the last one to execute). This is obviously the inverse of the "and list".

Example 3−61. Using "or lists" in combination with an "and list"

```
#!/bin/bash  # "Delete", not−so−cunning file deletion utility. # Usage: delete filename  i
```

Clever combinations of "and" and "or" lists are possible, but the logic may easily become convoluted and require extensive debugging.

3.18. Arrays

Newer versions of **bash** support one−dimensional arrays. Arrays may be declared with the **`variable[xx]`** notation or explicitly by a **`declare −a variable`** statement. To dereference (find the contents of) an array variable, use *curly bracket* notation, that is, **`${variable[xx]}`**.

Example 3−62. Simple array usage

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

# Note that array members need not be consecutive
# or contiguous.

# Some members of the array can be left uninitialized.
# Gaps in the array are o.k.

echo −n "area[11] = "
echo ${area[11]}
echo −n "area[13] = "
echo ${area[13]}
# Note that {curly brackets} needed
echo "Contents of area[51] are ${area[51]}."

# Contents of uninitialized array variable print blank.
echo −n "area[43] = "
echo ${area[43]}
echo "(area[43] unassigned)"

echo

# Sum of two array variables assigned to third
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo −n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo −n "area[6] = "
echo ${area[6]}
# This doesn't work because
# adding an integer to a string is not permitted.

exit 0
```

Arrays variables have a syntax all their own, and even standard bash operators have special options adapted for array use.

Example 3−63. Some special properties of arrays

```
#!/bin/bash

declare −a colors
```

```
# Permits declaring an array without specifying size.

echo "Enter your favorite colors (separated from each other by a space)."

read −a colors
# Special option to 'read' command,
# allowing it to assign elements in an array.

echo

element_count=${#colors[@]}
# Special syntax to extract number of elements in array.
index=0

while [ $index −lt $element_count ]
do
  echo ${colors[$index]}
  let "index = $index + 1"
done

echo

exit 0
```

Arrays enable implementing a shell script version of the *Sieve of Erastosthenes*. Of course, a resource−intensive application of this nature should really be written in a compiled language, such as C. It runs excruciatingly slowly as a script.

Example 3−64. Complex array application: *Sieve of Erastosthenes*

```
#!/bin/bash

# sieve.sh
# Sieve of Erastosthenes
# Ancient algorithm for finding prime numbers.

# This runs a couple of orders of magnitude
# slower than equivalent C program.

LOWER_LIMIT=1
# Starting with 1.
UPPER_LIMIT=1000
# Up to 1000.
# (You may set this higher...
#  if you have time on your hands.)

PRIME=1
NON_PRIME=0

let SPLIT=UPPER_LIMIT/2
# Optimization:
# Need to test numbers only
# halfway to upper limit.


declare −a Primes
# Primes[] is an array.


initialize ()
```

```
{
# Initialize the array.

i=$LOWER_LIMIT
until [ $i −gt $UPPER_LIMIT ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
# Assume all array members guilty (prime)
# until proven innocent.
}

print_primes ()
{
# Print out the members of the Primes[] array
# tagged as prime.

i=$LOWER_LIMIT

until [ $i −gt $UPPER_LIMIT ]
do

  if [ ${Primes[i]} −eq $PRIME ]
  then
    printf "%8d" $i
    # 8 spaces per number
    # gives nice, even columns.
  fi

  let "i += 1"

done

}

sift ()
{
# Sift out the non−primes.

let i=$LOWER_LIMIT+1
# We know 1 is prime, so
# let's start with 2.

until [ $i −gt $UPPER_LIMIT ]
do

if [ ${Primes[i]} −eq $PRIME ]
# Don't bother sieving numbers
# already sieved (tagged as non−prime).
then

  t=$i

  while [ $t −le $UPPER_LIMIT ]
  do
    let "t += $i "
    Primes[t]=$NON_PRIME
    # Tag as non−prime
    # all multiples.
  done
fi

  let "i += 1"
done


}


# Invoke the functions sequentially.
initialize
sift
print_primes
echo
# This is what they call structured programming.

exit 0
```

3.19. Files

- `/etc/profile`
- `$HOME/.bashrc`

3.20. Here Documents

A *here document* is a way of feeding a command script to an interactive program, such as **ftp**, **telnet**, or **ex**. Typically, it consists of a command list to the program, delineated by a limit string. The special symbol << precedes the limit string. This has the same effect as redirecting the output of a file into the program, that is,

```
interactive−program 60; command−file
```

where `command−file` contains

```
command #1
command #2
...
```

The "here document" alternative looks like this:

```
#!/bin/bash
interactive−program 60;60;LimitString
command #1
command #2
...
LimitString
```

Choose a limit string sufficiently unusual that it will not occur anywhere in the command list and confuse matters.

Note that "here documents" may sometimes be used to good effect with non−interactive utilities and commands.

Example 3−65. dummyfile: Creates a 2−line dummy file

```
#!/bin/bash

# Non−interactive use of 'vi' to edit a file.
# Emulates 'sed'.

if [ −z $1 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi

TARGETFILE=$1

vi $TARGETFILE 60;60;x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23

# Note that ^[ above is a literal escape
# typed by Control−V Escape

exit 0
```

The above script could just as effectively have been implemented with **ex**, rather than **vi**. Here documents containing a list of **ex** commands are common enough to form their own category, known as *ex scripts*.

Example 3−66. broadcast: Sends message to everyone logged in

```
#!/bin/bash

wall 60;60;zzz23EndOfMessagezzz23
Dees ees a message frrom Central Headquarters:
Do not keel moose!
# Other message text goes here.
# Note: Comment lines printed by 'wall'.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
# wall 60;message−file

exit 0
```

Example 3−67. Multi−line message using cat

```
#!/bin/bash

# 'echo' is fine for printing single line messages,
#  but somewhat problematic for for message blocks.
#  A 'cat' here document overcomes this limitation.

cat 60;60;End−of−message
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
End−of−message

exit 0
```

Example 3−68. upload: Uploads a file pair to "Sunsite" incoming directory

```
#!/bin/bash

# upload
# upload file pair (filename.lsm, filename.tar.gz)
# to incoming directory at Sunsite


if [ −z $1 ]
then
  echo "Usage: `basename $0` filename"
  exit 1
fi


Filename=`basename $1`
# Strips pathname out of file name

Server="metalab.unc.edu"
```

```
Directory="/incoming/Linux"
# These need not be hard−coded into script,
# may instead be changed to command line argument.

Password="your.e−mail.address"
# Change above to suit.

ftp −n $Server 60;60;End−Of−Session
# −n option disables auto−logon

user anonymous $Password
binary
bell
# Ring 'bell' after each file transfer
cd $Directory
put $Filename.lsm
put $Filename.tar.gz
bye
End−Of−Session

exit 0
```

Note: Some utilities will not work in a "here document". The pagers, **more** and **less** are among these.

For those tasks too complex for a "here document", consider using the **expect** scripting language, which is specifically tailored for feeding input into non−interactive programs.

3.21. Miscellany

Uses of `/dev/null`

Think of `/dev/null` as a "black hole". It is the nearest equivalent to a write−only file. Everything written to it disappears forever. Attempts to read or output from it result in nothing. Nevertheless, `/dev/null` can be quite useful both from the command line and in scripts.
Suppressing stdout or stderr (from Example 3−70):
```
rm $badname 262;/dev/null #           So error messages [stderr] deep−sixed.
```
Deleting contents of a file, but preserving the file itself, with all attendant permissions (from Example 2−1 and Example 2−2):
```
cat /dev/null 62; /var/log/messages cat /dev/null 62; /var/log/wtmp
```
Automatically emptying the contents of a log file (especially good for dealing with those nasty "cookies" sent by Web commercial sites):
```
rm ~/.netscape/cookies ln −s /dev/null ~/.netscape/cookies # All cookies now get ser
```

Uses of `/dev/zero`

Like `/dev/null`, `/dev/zero` is a pseudo file, but it actually contains nulls (numerical zeros, not the ASCII kind). Output written to it disappears, and it is fairly difficult to actually read the nulls in `/dev/zero`, though it can be done with **od** or a hex editor. The chief use for `/dev/zero` is in creating an initialized dummy file of specified length intended as a temporary swap file.

Example 3−69. Setting up a swapfile using `/dev/zero`

```
#!/bin/bash

# Creating a swapfile.
# This script must be run as root.

FILE=/swap
BLOCKSIZE=1024
PARAM_ERROR=33
SUCCESS=0


if [ −z $1 ]
then
  echo "Usage: `basename $0` swapfile−size"
  # Must be at least 40 blocks.
  exit $PARAM_ERROR
fi

dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$1

echo "Creating swapfile of size $1 blocks (KB)."

mkswap $FILE $1
swapon $FILE

echo "Swapfile activated."
```

```
exit $SUCCESS
```

3.22. Debugging

The bash shell contains no debugger, nor even any debugging−specific commands or constructs. Syntax errors or outright typos in the script generate cryptic error messages that are often of no help in debugging a non−functional script.

Example 3−70. test23, a buggy script

```
#!/bin/bash

a=37

if [$a −gt 27 ]
then
  echo $a
fi

exit 0
```

Output from script:

```
./test23: [37: command not found
```

What's wrong with the above script (hint: after the **if**)?

What if the script executes, but does not work as expected? This is the all too familiar logic error.

Example 3−71. test24, another buggy script

```
#!/bin/bash

# This is supposed to delete all filenames
# containing embedded spaces in current directory,
# but doesn't.  Why not?


badname=`ls | grep ' '`

# echo "$badname"

rm "$badname"

exit 0
```

To find out what's wrong with Example 3−71, uncomment the **`echo "$badname"`** line. Echo statements are useful for seeing whether what you expect is actually what you get.

Summarizing the symptoms of a buggy script,

1. It bombs with an error message syntax error, or
2. It runs, but does not work as expected (logic error)

3. It runs, works as expected, but has nasty side effects (logic bomb.

Tools for debugging non−working scripts include

1. echo statements at critical points in the script to trace the variables, and otherwise give a snapshot of what is going on.
2. using the **tee** filter to check processes or data flows at critical points.
3. setting option flags `−n −v −x`

   `sh −n scriptname` checks for syntax errors without actually running the script. This is the equivalent of inserting **`set −n`** or **`set −o noexec`** into the script. Note that certain types of syntax errors can slip past this check.

   `sh −v scriptname` echoes each command before executing it. This is the equivalent of inserting **`set −v`** or **`set −o verbose`** in the script.

   `sh −x scriptname` echoes the result each command, but in an abbreviated manner. This is the equivalent of inserting **`set −x`** or **`set −o xtrace`** in the script.

   Inserting **`set −u`** or **`set −o nounset`** in the script runs it, but gives an unbound variable error message at each attempt to use an undeclared variable.
4. trapping at exit

   The **exit** command in a script actually sends a signal 0, terminating the process, that is, the script itself. It is often useful to trap the **exit**, forcing a "printout" of variables, for example. The **trap** must be the first command in the script.

   trap

   Specifies an action on receipt of a signal; also useful for debugging.

   ```
   trap 2 #ignore interrupts (no action specified)  trap 'echo "Control−C disabled."' 2
   ```

Example 3−72. trapping at exit

```
#!/bin/bash

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
# EXIT is the name of the signal generated
# upon exit from a script.

a=39

b=36

exit 0
# Note that commenting out the 'exit' command
```

```
# does not make a difference.
```

3.23. Options

Options are settings that change shell and/or script behavior. A script enables options by the **set** command.

The following are some useful options. They may be set in either abbreviated form or by complete name.

Table 3−1. bash options

Abbreviation	Name	Effect
`−C`	noclobber	Prevent overwriting of files by redirection (may be overridden by >|)
`−f`	noglob	Filename expansion disabled
`−p`	privileged	Script runs as "suid"
`−u`	nounset	Attempts to use undefined variables result in error message
`−v`	verbose	Print commands to stdout before executing
`−x`	xtrace	Similar to `−v`, but expands commands
`−`	(none)	End of options flag. All other args are positional parameters.
`−−`	(none)	Unset positional parameters. If arguments given (`−−arg1arg2`), positional parameters set to arguments.

3.24. Gotchas

Assigning reserved words or characters to variable names.

```
var1=case
# Causes problems.
var2=xyz((!*
# Causes even worse problems.
```

Using a hyphen or other reserved characters in a variable name.

```
var−1=23
# Use 'var_1' instead.
```

Using white space inappropriately (in contrast to other programming languages **bash** can be finicky about white space).

```
var1 = 23
# 'var1=23' is correct.
let c = $a − $b
# 'let c=$a−$b' or 'let "c = $a − $b"' are correct.
if [ $a −le 5]
# 'if [ $a −le 5 ]' is correct.
```

Using uninitialized variables (that is, using variables before a value is assigned to them). An uninitialized variable has a value of "null", *not* zero.

Commands issued from a script may fail to execute because the script owner lacks execute permission for them. If a user cannot invoke a command from the command line, then putting it into a script will likewise fail. Try changing the attributes of the command in question, perhaps setting the suid bit (as root, of course).

Using bash version 2 functionality (see below) in a script headed with **`#!/bin/bash`** may cause a bailout with error messages. Your system may still have an older version of bash as the default installation. Try changing the header of the script to **`#!/bin/bash2`**.

Making scripts "suid" is generally a bad idea, as it may compromise system security. Administrative scripts should be run by root, not regular users.

3.25. Bash, version 2

The current version of **bash**, the one you have running on your machine, is actually version 2. This update of the classic **bash** scripting language added array variables, string and parameter expansion, and indirect variable references, among other features.

Example 3−73. String expansion

```
#!/bin/bash

# String expansion.
# Introduced in version 2 of bash.

# Strings of the form $'xxx'
# have the standard escaped characters interpreted.

echo $'Ringing bell 3 times \a \a \a'
echo $'Three form feeds \f \f \f'
echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'

exit 0
```

Example 3−74. Indirect variable references

```
#!/bin/bash

# Indirect variable referencing.
# This has a few of the attributes of references in C++.


a=letter_of_alphabet
letter_of_alphabet=z

# Direct reference.
echo "a = $a"

# Indirect reference.
echo "Now a = ${!a}"

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"
table_cell_3=387
echo "Value of t changed to ${!t}"
# Useful for referencing members
# of an array or table,
# or for simulating a multi−dimensional array.
# An indexing option would have been nice (sigh).


exit 0
```

Example 3−75. Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards

```
#!/bin/bash2
```

```
# Must specify version 2 of bash, else might not work.

# Cards:
# deals four random hands from a deck of cards.

UNPICKED=0
PICKED=1

DUPE_CARD=99

LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUITE=13
CARDS=52

declare −a Deck
declare −a Suites
declare −a Cards
# It would have been easier and more intuitive
# with a single, 3−dimensional array. Maybe
# a future version of bash will support
# multidimensional arrays.


initialize_Deck ()
{
i=$LOWER_LIMIT
until [ $i −gt $UPPER_LIMIT ]
do
  Deck[i]=$UNPICKED
  let "i += 1"
done
# Set each card of "Deck" as unpicked.
echo
}

initialize_Suites ()
{
Suites[0]=C #Clubs
Suites[1]=D #Diamonds
Suites[2]=H #Hearts
Suites[3]=S #Spades
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
# Alternate method of initializing array.
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS"
if [ ${Deck[card_number]} −eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
  return $card_number
else
  return $DUPE_CARD
fi
}
```

```
parse_card ()
{
number=$1
let "suite_number = number / CARDS_IN_SUITE"
suite=${Suites[suite_number]}
echo −n "$suite−"
let "card_no = number % CARDS_IN_SUITE"
Card=${Cards[card_no]}
printf %−4s $Card
# Print cards in neat columns.
}

seed_random ()
{
# Seed random number generator.
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}

deal_cards ()
{
echo

cards_picked=0
while [ $cards_picked −le $UPPER_LIMIT ]
do
  pick_a_card
  t=$?

  if [ $t −ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # Change back to 1−based indexing (temporarily).
    let "u %= $CARDS_IN_SUITE"
    if [ $u −eq 0 ]
    then
     echo
     echo
    fi
    # Separate hands.

    let "cards_picked += 1"
  fi
done

echo

return 0
}


# Structured programming:
# entire program logic modularized in functions.

#================
seed_random
initialize_Deck
initialize_Suites
```

```
initialize_Cards
deal_cards

exit 0
#================



# Exercise 1:
# Add comments to thoroughly document this script.

# Exercise 2:
# Revise the script to print out each hand sorted in suites.
# You may add other bells and whistles if you like.

# Exercise 3:
# Simplify and streamline the logic of the script.
```

Chapter 4. Credits

Philippe Martin translated this document into DocBook/SGML. While not on the job at a small French company as a software developer, he enjoys working on GNU/Linux documentation and software, reading literature, playing music, and for his peace of mind making merry with friends. You may run across him somewhere in France or in the Basque Country, or email him at feloy@free.fr.

Bibliography

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1−156592−225−5.

To unfold the full power of shell scripting, you need at least a passing familiarity with **sed** and **awk**. This is the standard tutorial. It includes an excellent introduction to "regular expressions". Read this book.

Aeleen Frisch, *Essential System Administration*, 2nd edition, O'Reilly and Associates, 1995, 1−56592−127−5.

This excellent sys admin manual has a decent introduction to shell scripting for sys administrators and does a nice job of explaining the startup and initialization scripts. The book is long overdue for a third edition (are you listening, Tim O'Reilly?).

Stephen Kochan and Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

The standard reference, though a bit dated by now.

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1−56592−347−2.

This is a valiant effort at a decent shell primer, but somewhat deficient in coverage on programming topics and lacking sufficient examples.

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1−56592−260−3.

Contains a couple of sections of very informative in−depth articles on shell programming, but falls short of being a tutorial. It also reproduces much of the regular expressions tutorial from the Dougherty and Robbins book, above.

Ellen Siever, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1−56592−585−8.

The all−around best Linux command reference, even has a bash section.

The O'Reilly books on Perl. (Actually, any O'Reilly books.)

The man pages for **bash** and **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **patch**, **tar**, **tr**, **xargs**. The texinfo documentation on **bash**, **dd**, **gawk**, and **sed**.

The excellent "Bash Reference Manual", by Chet Ramey and Brian Fox, distributed as part of the "bash−2−doc" package (available as an rpm).

Appendix A. Copyright

The "Advanced Bash−Scripting HOWTO" is copyright, (c) 2000, by Mendel Cooper. This document may only be distributed subject to the terms and conditions set forth in the LDP License.

If this document is incorporated into a printed book, the author requests a courtesy copy (this is a request, not a requirement).

Notes

[1]
A flag is an argument that acts as a signal, switching script behaviors on or off.