Linux BASH Shell Scripting
Basics of Creating and Utilizing BASH Scripts

Pete Nesbitt
May 2006

The information presented here should act as a guide to creating quality scripts using the Linux built-in BASH scripting language. BASH (Bourne Again Shell) is a scripting language as well as the default command interpreter in most Linux distributions, including Red Hat Linux. The ability to create quality scripts is arguably the most time and error saving aspect of Linux Systems Administration. The fact that the default shell or user environment in Linux is BASH makes makes learning to create scripts both very valuable as well as simple. In the following document we will look at fundamentals of scripts, best practices creating and testing of scripts. Shell scripts can be very simple or extremely complex involving dozens of support files, but regardless of the complexity, they should be created in a consistent, self-explanatory and easy to read format.

We won't cover regular expressions, or shell utilities like loops since that information is readily available on the Internet and in books. If after looking at an example, you do not understand an aspect like "*while read x in `ls`*", be sure to research how these functions work, or ask a more experienced staff member for help. Mimicking a process without understanding is of little or no value, and misunderstanding the process can lead to dangerous scripts and frustration.

In order to achieve the best value from this document, you should be familiar with common commands, basic regular expressions and GNU utilities. Anything used within the examples can be researched using the *man* command or by searching Google. For example, you will want to be familiar with grep, sed, awk, and *cut*.

IMPORTANT NOTE: If you try to cut and paste the examples in this document, you will probably need to manually replace the special characters since the word processor modifies them for presentation. Characters copied from the command line will be okay, but ones taken directly from this document will need to be corrected. These include, but are not limited to ' " ` / \ |

The Bash Shell mantra:
Every *formal script* should begin with command line interpreter definition.
In BASH, it is a virtual poem:

Hash Bang Slash
Bin Slash Bash

Literally:
#!/bin/bash

In addition to what I call a *formal script*, you can achieve script like functionality by entering a series of command at the prompt.

Linux BASH Shell Scripting
Basics of Creating and Utilizing BASH Scripts

Pete Nesbitt
May 2006

What can I do with a script?
Any task you can run at the command line can be incorporated into a script. This is because working at the command line and writing a script are essentially the same thing. Unix and Linux utilities and commands tend to be very simple and singular in task. However, stringing a series of such utilities together can create very flexible, powerful and extensible command line or script. There is no other environment that can be manipulated to perform with such flexibility and adaptability as the command line and it's related script files.

Any task that is complicated, repetitive or needs to be performed numerous times should be scripted. This saves time, eliminates the chances of typos, and often can be reused later.

Creating a script:
1. start by laying out the flow with only comments of what each step will perform, don't worry about the actual code until you define the steps you want to perform.
2. use the command line to test pieces of your script
3. take small bits of code that you have tested at the command line, then move them into the script
4. always test in a safe area, not working on important data or files until you have confirmed the scripts works as expected.

Learning to write quality BASH Scripts:
1. Be consistent. Develop a layout for your scripts and stick to that style.
2. Take your time. Scripting is not overly complicated, but does rushing things will just result in errors and frustration.
3. Practice makes perfect. Try and script as many things as possible, especially complicated tasks. This will help develop your skills working with sets of commands as opposed to doing one thing, saving to a file, working on that file, then saving those results and so on. Even if it is only to be used once, unless time is of the essence, go script crazy. You will be a better, more valuable Systems Administrator in the long run.
4. Review the work of others. In scripting, like almost everything else Unix or Linux related, there are many ways to do any given task. See how others do things, then decide what makes sense for you.

Best Practices:
Always write your scripts in a way that others can easily follow the flow and functionality.
- give your scripts a meaningful name and include .sh as a suffix
- always define the command interpreter on the first line
- include your name and the date
- include the file name and a brief description of what the script will do

- define functions, variables and include files before the main code
- include comments (lines starting with a #) and blank lines for readability
- indent stanzas or code blocks for readability
- fully qualify your commands
  - this adds security as well as making it *cron safe*
  - although this may break portability, you can use another shell script to interrogate the system and ensure portability when defining commands as variables.
- add an 'End Of File' comment

What to do if your script doesn't work:

1. add some output to see how things progress
   - add an echo line at each main step, that will tell you what does work
     - example:
       after a line like "*let x=$x+1*"
       add "*echo "the value of x is now: $x"*" to see if the value is changing as expected
2. if you get an error stating "permission denied" the script is not executable
3. try snipits on the command line or in a smaller script to find and fix a failing line
4. use Google or other resources to confirm your code, such as a loop format

Sample Script Template: (this is not a physical template, but a general format you may want to adopt)

```
#!/bin/bash
#
# file: myscript.sh
# created by Pete Nesbitt May 2006
# this script will perform some fancy task.
#
# set some variables to be used throughout the script
TARGET=”remote.host.com”
DATE_STAMP=”`date +%y%m%d`”
EMAIL_RECIPIENTS=”nsa@riptown.com”

# here we will manipulate something
#
 some code goes here
# now we need to do this other thing
some other code goes here
# eof
```

Functions:
A function is simple a code block that can be called several times without the need to rewrite the whole thing every time.
In BASH a function is defined as:

```
my_function()
 {
  the code block
  }
```

Then can be used anywhere in the script like so:

```
do some stuff...
  my_function
some more code
```

See the Samples section at the end for real world examples.

Building a Functions Library:
Whether you use the *include* statement or just have an inventory of code snipits, you can be consistent and avoid recreating the wheel by accumulating a library of shell functions, tasks and variables.

A Simple Script to Try:
First lets make a quick script to create some files that we can later manipulate, then we will rename

them. Since these are simple examples, lets cheat a bit here, using the command line instead of a formal script.
Create the original files:

```
count=0;while [ $count -lt 10 ]; do touch file_$count;let count=${count}+1; done
```

We now have 10 files, named *file_0* through *file_9*
Lets rename them. We could do each separately using:
"mv file_0 new_name_0;mv file_1 new_name_1; ... mv file_9 new_name_9"
but that seems like a lot of work and a lot of opportunity for error.

A simpler and more efficient method would be to create a loop that would rename each file automatically. This is very similar to the way we created the files. This time though, we will do each step separately. Note the prompt changes when BASH expects more information before it can complete the command string. This is called the second level prompt (the primary prompt is a $ for most users and a # for the root user).

```
$ for x in `ls`
> do
> n="`echo $x|cut -d_ -f2`"
> mv $x new_file_$n
> done
```

We now have 10 files, named *new_file_0* thru *new_file_9*

If you now do an up-arrow, the previous command is displayed as:

```
for x in `ls`; do n="`echo $x|cut -d_ -f2`"; mv $x new_file_$n; done
```

Whenever doing mass changes, it is prudent to create a test directory to use when building the script or command line, so you do not damage your real source files in the case of an error.

EXIT CODES:

Whenever a command is executed, upon completions a numeric value set which indicates whether the command was successful or not. An exit value of *0* (*zero*) means success. Any non zero value indicates non-success, usually *1* (*one*) and usually failure. In the case of a command like grep, and exit code of *1* does not really mean the command failed, but just that it did not find anything that matched the query.

An exit code can be forced with a 'exit' string such as this example place within an 'if' statement:

```
if
   ...some code...
else
        echo "    Operation Failed!"
        exit 1
fi
```

If the '*else*' line is matched, then the script exits with a status of *1*

You can also test for success of a command by retrieving the exit code, which is represented by the variable "*$?*" using something like this:

```
...some command...
if [ $? -eq 0 ] ; then
...success, carry on
```

or, more you may prefer this, slightly more refined method:

```
...some command...
RET_VAL="$?"
if [ $RET_VAL -eq 0 ] ; then
...success, carry on
```

The thing to remember is that the variable "*$?*" will be overwritten with the next commands completion unless you save it into a unique variable as in the second example.

How to use INCLUDE FILES:

You can store functions in one or more files outside your main script. In order to utilize these functions, you must have an "include" statement, then you can just call them as you would any other locally defined function. The include line does not actually have a syntax, it simply reads in additional functions.
Here we are adding the file *install_lib*, which contains a series of functions.
. *src/install_libs* (note the leading 'dot space' meaning 'look in this script plus in...')

INTERIGATING FOR VARIABLES:

Although the BASH shell is included in most Unix and Linux systems, some of the system commands or utilities are located in different directories. If you want your script to be truly portable, you must take this into account. Below is one of many ways to do it.

We read a file if the command names, formatted one command per line.
It's contents may look like this:

```
rm
grep
sed
```

```
awk
```

Then we search for them and save the output to an include file:
(the shortfall here is the 'which' (or 'type') command must be available, a more complex method is used in the Flac Jacket scripts)

```
while read CMD_VAR
do
  CMD_NAME="`echo $CMD_VAR|\` which tr\` [:lower:]  [:upper:]`"
  echo "${CMD_NAME}=`which  ${CMD_VAR}`" >> command_include_file
done < command_list.txt
```

REAL WORLD SAMPLES AND EXAMPLES:

Note, some lines may wrap in the following examples. This can be avoided in the script with a "\" to escape the line return. Also, these are not necessarily following the 'Best Practices" model, however, I did not want to alter the examples.

For a complete package involving most things we have discussed here, download Flac-Jacket, a good example of **complex BASH scripting.**
http://linux1.ca/code/flac_jacket-current.tgz

A simple script to send a monthly email reminder:

```
#!/bin/bash
#
# send email reminder to visit yi monthly
# run from cron monthly

/usr/bin/printf "Time to update yi.org\n\nthe url is http://www.whyi.org/admin/\n"|/bin/mail -s "Update
YI" pete
```

This script backs up a local unix/linux system to a Windows server.

```
#!/bin/bash
#
# script to backup local files on remote system via ssh pub-key authentication.
# both systems must be setup for ssh-key auth.
# June 26, 2000. Pete Nesbitt

# file to read  list of source dirs from.
SOURCES=/etc/backup_src.txt
```

```
EXCLUDE=/tmp/tar_exclude.tmp

# target host and dir
TARGET_HOST=192.219.60.22
TARGET_DIR=$TARGET_HOST:e:/bu-pnesbitt

while read LINE
do
  TARFILE="/tmp/`echo $LINE| sed -e s/'\/'/_/g `.tar"
  find `echo $LINE` -type s > $EXCLUDE
  find `echo $LINE` -type p >> $EXCLUDE
   # check which tar, the P will either GNU="not remove the leading /" or Solaris="don't add a
trailing /"
  /bin/tar cPf $TARFILE -X $EXCLUDE $LINE
  /bin/gzip -9 $TARFILE
  # next line forces "open ssh" as that is what the server uses, open & com don't play well.
  su - backup -c "/usr/bin/scp $TARFILE.gz $TARGET_DIR/"
  rm -f $TARFILE.gz
done < $SOURCES
```

An example of a <u>FUNCTION</u>:
This is a clip of the first two of several possible error messages, they utilize a string defined later in the script, just before calling the function. This allows for more meaningful messages. Following the two functions, is a clip of how one is used later in the program.

```
error_1 ()
{
   echo " ERROR: There is a \"${ERR_TARGET}\" directory, but you can not write to it."
   error_mesg_perm
   echo ""
   exit 1
}

######

error_2 ()
{
   echo " ERROR: There is a file named \"${ERR_TARGET}\", but it does not appear to be a
directory."
   echo " Remove or rename the \"${ERR_TARGET}\" file, or choose a different location, and try
again."
   echo ""
   exit 1
}
```

Later, they can be called like this:
(note the locally defined “ERR_TARGET” as well as the function call such as “error_1”)

```
# does the thumbnails exist, if so is it a dir, if so can we write to it
ERR_TARGET="thumbnails"
if [ -e $THUMBNAILS ]; then
 if [ -d $THUMBNAILS ]; then
  # make sure we can write to it
  if ! [ -w  $THUMBNAILS ]; then
   error_1
  fi
 else
   error_2
 fi
else
  mkdir $THUMBNAILS
  #run a test for success
  if [ -d $THUMBNAILS ]; then
   if ! [ -w $THUMBNAILS ]; then
   error_3
   fi
  fi
fi
```

EOF