Bash Shell Programming in Linux

(double-click any word to see its definition)

Revised 3/2006

Bash what?

Okay, I grant that this page might represent a leap from the familiar to the alien without much warning. Here are some explananatory notes:

- Under Linux, there are some powerful tools that for all practical purposes are unavailable under Windows (I can imagine all the old Linux hands saying "Duh!").
- One of these tools is something called "shell programming". This means writing code that a command shell executes.
- There is something like this under Windows, but as usual, the Windows version is a weak imitation.
- The most common Linux shell is named "Bash". The name comes from "Bourne Again SHell," which, in turn ... (imagine a lengthy recursion terminating in a caveman's grunt).
- There are many other shells available. Unless there is a compelling reason not to, I recommend that people stick to the Bash shell, because this increases the chance that your scripts will be portable between machines, distributions, even operating systems.
- I'll be showing some very basic examples of Bash shell programming on this page, and I want to say at the outset that shell programming is an art, not a science. That means there is always some other way to do the same thing.
- Because shell programming is an art, please don't write to say, "Wow, that was a really inefficient way to do such-and-such." Please do write (message page) to report actual errors.
- If this page seems too sketchy and elementary for your taste, you can choose from among the more advanced resources in this list.

Introduction

- Early computers had a teletype machine with a keyboard for I/O. Later, glass terminals became the norm, but the behavior was much the same — a keyboard, a screen, a text display. A program was responsible for mediating the transaction between the operator and the machine, and as the years passed this program (the command interpreter or shell) became more sophisticated.
- At this stage the command shell has become rather too sophisticated, typically having a dozen ways to do any particular thing. In this page I will try to limit myself to describing a handful of useful operations, based not on listing everything that can be done, but on solving specific problems. There are some links at the bottom of this page for those wanting more depth.

Preliminaries

- There are two primary ways to use the shell: interactively and by writing shell scripts.
  - In the *interactive mode*, the user types a single command (or a short string of commands) and the result is printed out.
  - In *shell scripting*, the user types anything from a few lines to an entire program into a text editor, then executes the resulting text file as a shell script.
  - It is often the case that an interactive session becomes a shell scripting session, once things get too complicated for simple interactive line entries, or because a specific sequence of commands appears to be generally useful and worth preserving.
- In a modern Linux environment the user can have more than one shell open at a time, either by moving between a sequence of independent "virtual terminals" in a text-only environment, or by opening any number of shell windows in the X Windows environment.
- The advantage of having more than one shell available is that one shell can be used for testing one command at a time, while another might provide a text editor for assembling single commands into a shell program.
- I don't want to get too distribution-specific, but if you are not hosting X Windows and want more than one simultaneous shell session, with many current distributions you can switch between "virtual terminals" by pressing Ctrl+Alt+F(n), n typically between 1 and 6.
- In an environment that supports X Windows, simply open any desired number of command shell windows and move between them.

Simple Stuff

- First, a convention. I'll list things for you to type in this format:

```
$ date
```

I will list the computer's reply like this:

```
Tue Dec 23 10:52:51 PST 2003
```

Notice the "$" symbol in the user entry above. This is a generic shell prompt, and yours will almost certainly look different (but it will include a similar symbol). I'll be using one of two prompts (this is a common convention, worth

remembering): I'll use "$" to refer to a normal user session, and "#" to refer to a root session.

- **NOTE:** *Avoid using root sessions and permissions unless it is required.* Misused root authority can cause very serious harm to your system. Since this is a tutorial in which you will want to experiment with different commands, limit the chance for harm by doing so as an ordinary user.
- To put this another way, enter this example:

```
# whoami
root
```

  If your session produced the result shown above, please — log out and become an ordinary user.

- *In shell programming, spaces matter.* If you see spaces between words and characters in these examples, be sure to include the spaces.
- *In shell programming, case matters also.* If you don't get the results shown on this page, look at the case of your entries.

Where are you?

- As you may be aware, a Linux filesystem is in the form of a large tree with many branches called "subdirectories". When you issue a shell command, it is often necessary to know where you are in the "tree". Type this example:

```
$ pwd
/path/path/path
```

  When you try this example ("pwd" means "print working directory"), your current working directory will be printed.

- You can decide where you are in the tree. Type this example:

```
$ cd ~
$ pwd
/home/username
```

  The symbol "~" is a special shortcut character that can be used to refer to your home directory. You could have typed this —

```
$ cd /home/username
```

  — and accomplished the same result, but if you think about it, the "~" character is more portable. Later, when you are writing shell scripts, you might want a command that moves to any user's home directory.

Listing Files

- Directories contain files, and you can list them in a simple, compact format:

```
$ ls
filename filename filename ...
```

  Or you can list them in more detail:

```
$ ls -la
(detailed list, one file per line)
```

  And, very important, to find out what a command's options are, use the "man" (manual) command:

```
$ man ls
(manual page for "ls")
```

  NOTE: *The "man" command allows you to learn a command's options. You still have to remember the command's name.*

  To find files by name:

```
$ find . -name '*.jpg'
(list of files with .jpg suffix in current and all child directories)
```

  To create a text diagram of the directory tree:

```
$ tree -d .
(diagram of the directory tree from the current directory)
```

  The "tree" command is less useful now that directory trees have become so complicated, and now that most distributions support X Windows and sophisticated filesystem browsers.

Examining Files

- There are a number of things you can do to find out more about the files in the list. Here are just a few:

  The "file" command tries to identify files by examining their contents:

```
$ file tux_small.png
tux_small.png: PNG image data, 128 x 151, 8-bit/color RGB, non-interlaced
```

The next example uses the obscurely named "cat" command. It prints the contents of a file. Unfortunately if the file's contents are not readable, they get printed anyway.

```
$ cat zipcodes.txt
(prints the entire contents of a file named "zipcodes.txt")
```

If a file is too long to be viewed on one page, you can say:

```
$ more zipcodes.txt
(prints file one screenful at a time)
```

You can also use "grep" to print only those parts of a file you are interested in:

```
$ grep 10001 zipcodes.txt
(prints only those lines that have the character string "10001" in them)
```

The "grep" command is very useful, unfortunately it has a difficult-to-remember name. Be sure to:

```
$ man grep
```

There are many, many more shell commands to learn and to use. You may want to browse the list of Useful Links for more detail.

Pipelines and Redirection

- *You can use a pipeline (symbolized by "|") to make the output of one command serve as the input to another command.* This idea can be used to create a combination of commands to accomplish something no single command can do.

  Enter this command:

  ```
  $ echo "cherry apple peach"
  cherry apple peach
  ```

  Okay, let's say we want to sort these words alphabetically. There is a command "sort", but it sorts entire lines, not words, so we need to break this single line into individual lines, one line per word.

  Step one: pipe the output of "echo" into a translation (tr) command that will replace spaces with linefeeds (represented by "\n"):

  ```
  $ echo "cherry apple peach" | tr " " "\n"
  cherry
  apple
  peach
  ```

  Success: each word appears on a separate line. Now we are ready to sort.

  Step two: add the sort command:

  ```
  $ echo "cherry apple peach" | tr " " "\n" | sort
  apple
  cherry
  peach
  ```

  Let's try reversing the order of the sort:

  ```
  $ echo "cherry apple peach" | tr " " "\n" | sort -r
  peach
  cherry
  apple
  ```

- **Remember:** *A pipeline ("|") takes the output of one command and makes it the input to another command.*

- Normally the output from commands is printed on the screen. But using the symbol ">", you can redirect the output to a file:

  ```
  $ date > RightNow.txt
  $ cat RightNow.txt
  Tue Dec 23 14:43:33 PST 2003
  ```

  The above example used ">" to replace the content of any existing file having the name "RightNow.txt". To append new data to an existing file, use ">>" instead:

  ```
  $ date >> RightNow.txt
  $ cat RightNow.txt
  Tue Dec 23 14:43:33 PST 2003
  Tue Dec 23 14:46:10 PST 2003
  ```

- **Remember:** *Use ">" to overwrite any existing file, use ">>" to append to any existing file.* In both cases, if no file exists, one is created.

- Many commands have inputs as well as outputs. The input defaults to the keyboard, the output defaults to the screen.
- To redirect the output to a file, use ">" or ">>" as shown above.
- To make the output of a *command* serve as the input of another command, use "|".
- To make the contents of a *file* serve as the input to a command, use "<":

```
$ wc < RightNow.txt
2 12 58
```

As is so often the case in shell programming, there is at least one other way to produce the above result:

```
$ cat RightNow.txt | wc
2 12 58
```

Shell Script Basics

- A shell script is a plain-text file that contains shell commands. It can be executed by typing its name into a shell, or by placing its name in another shell script.
- To be executable, a shell script file must meet some conditions:
  - The file must have a special first line that names an appropriate command processor. For this tutorial, the following will work in most cases:

    ```
    #!/bin/bash
    ```

    If this example doesn't work, you will need to find out where your Bash shell executable is located and substitute that location in the above example. Here is one way to find out:

    ```
    $ whereis bash
    ```

  - The file must be made executable by changing its permission bits. An example:

    ```
    $ chmod +x (shell script filename)
    ```

- A shell script file may optionally have an identifying suffix, like ".sh". This only helps the user remember which files are which. The command processor responsible for executing the file uses the executable bit, plus the file's first line, to decide how to handle a shell script file.
- One normally executes a shell script this way:

  ```
  $ ./scriptname.sh
  ```

  This special entry is a way to tell the command processor that the desired script is located in the current directory. Always remember: if you cannot get your shell script to run, remember this trick to provide its location as well as its name.

First Shell Script

- This will get you past the details of writing and launching a simple script.
  1. Choose a text editor you want to use. It can be a command-line editor like emacs, pico or vi, or an X Windows editor if you have this option.
  2. Run your choice of editor and type the following lines:

     ```
     #!/bin/bash
     echo "Hello, world."
     ```

     NOTE: *Be sure to place a linefeed at the end of your script.* Forgetting a terminating linefeed is a common beginner's error.
  3. Save the file in the current working directory as "myscript.sh".
  4. Move from the text editor to a command shell.
  5. From the command shell, type this:

     ```
     $ chmod +x myscript.sh
     ```

  6. To execute the script, type this:

     ```
     $ ./myscript.sh
     Hello, world.
     ```

- These steps will become second nature soon enough.

Tests and Branching

- Bash shell scripts can perform, and act on, various kinds of tests. This will be just the most basic introduction — see the reference material at Useful Links for more on this rather baroque topic.
- *To follow these and other examples that involve multiline shell scripts, please set up to edit and run a test script file (let's call it "myscript.sh") that you can use to enter and test various options. And remember that the examples won't include the all-important first line of the script (see script examples above) — it will be assumed to exist in each case.*

  Also, to save time, you may want to copy some of the shell code examples from this page directly into your editor.

- Here is an example of a test and branch:

```
if [ -e . ]
then
        echo "Yes."
else
        echo "No."
fi
```

Run the test script:

```
$ ./myscript.sh
Yes.
```

We created a test (the part of the script between "[" and "]") which tested whether a particular element existed ("-e"). Because the symbol "." in this context means the current directory, the test succeeded. Try replacing the "." with something that is not present in the current directory, example "xyz". See how the outcome changes.

It is important to realize that "[" is an alias for the command "test". The script could have been written as:

```
if test -e .
then
        echo "Yes."
else
        echo "No."
fi
```

NOTE: Be sure to read the "test" man page to find out all the different tests that are available:

```
$ man test
```

Before we move on, there is a perversity about tests in Bash shells that I want to discuss. It turns out, because of a historical accident that now might as well be cast in concrete, when a test is conducted or a command returns a result value, the numerical value for "true" is 0, and "false" is 1. Those of you who have some programming experience will likely find this reversal of intuition as annoying as I do.

Here is a way to get the result of the most recent logical test (and to show the weird reversal described above):

```
$ test -e .
$ echo $?
0

$ test -e xyz
$ echo $?
1
```

Please remember this reversal, because it confounds the process of thinking through, and constructing, logical tests. For example, you may want to write a shortcut form for a test that only acts on one kind of result:

```
$ test -e . && echo "Yes."
Yes.
```

This sort of shorthand relies on some knowledge of logical processing — if the left-hand part of an AND test yields "true", then the right-hand part must also be evaluated, and so it is. But the numerical "true" value for the left-hand test is 0, which would argue for the opposite logic.

Just to show how perverse this all is, here is an example of Bash logical testing that comes out the opposite way:

```
$ echo $(( 0 && 0 ))
0

$ echo $(( 1 && 0 ))
0

$ echo $(( 0 && 1 ))
0

$ echo $(( 1 && 1 ))
1
```

Yes, just as you would expect. So do be on guard against this shell "gotcha", which only affects the outcome of tests and command result values. It probably will not surprise you to learn that no one mentions this strange anomaly in the official Bash documentation.

A couple of rules about logical operators used as branches:

- If you write "test && command", the command will only be executed if the test *succeeds*.
- If you write "test || command", the command will only be executed if the test *fails*.

Run these tests:

```
$ true && echo "Yes."
Yes.
```

```
$ false || echo "Yes."
Yes.
```

Notice that the outcomes are entirely in keeping with one's intuition about such logical comparisons, and all is well as long as you don't think about the fact that true equals 0. :)

Here's another scheme commonly seen in shell script programming and interactive sessions:

```
$ command1 && command2 && command3 && command4
```

This line of code will not run the next command in the sequence unless the prior command has returned "true", meaning no errors. It is a way to avoid running a command if a required prior outcome is not present.

Loops and Repetition

- Here are some examples of loop operators:

```
for fn in *; do
        echo "$fn"
done
```

In this example, the "*" is expanded by the shell to a list of all the files in the current directory, then each filename is applied to the loop control area. In such a construct, any whitespace-delimited list will do:

```
for fn in tom dick harry; do
        echo "$fn"
done
```

```
$ ./myscript.sh
tom
dick
harry
```

This method will work for any list that uses spaces as delimiters. But what happens if you must parse a list that must be delimited by linefeeds instead of spaces, such as the case of a list of filenames or paths that contain spaces as part of their names?

You can solve such a problem this way (there are other solutions):

```
ls -1 | while read fn; do
        echo "$fn"
done
```

This example uses an option to "ls" (note: the option is "-" followed by the numerical digit "1", *not* a lowercase "L") that formats file listings with one name per line, then this list is pipelined to a routine that reads lines until there are no more to read. This meets the requirement that linefeeds become the delimiters between list elements, not spaces.

There is plenty more to this topic. Please refer to the list of Useful Links for more.

Using Numbers in Scripts

- Contrary to a sometimes-expressed view, numbers can easily be accommodated in scripts. Example:

```
n=1
while [ $n -le 6 ]; do
        echo $n
        let n++
done
```

```
$ ./myscript.sh
1
2
3
4
5
6
```

Notice the "let" command, which treats its argument in a way meant to accommodate numbers.

Here is a somewhat more complex example:

```
y=1
while [ $y -le 12 ]; do
        x=1
        while [ $x -le 12 ]; do
                printf "% 4d" $(( $x * $y ))
                let x++
        done
        echo ""
        let y++
done
```

```
$ ./myscript.sh
```

```
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

Coping with user input

- Here is an example that relies on user input to decide what to do. It exploits a shell feature as an easy way to create a menu of choices:

```
PS3="Choose (1-5):"
echo "Choose from the list below."
select name in red green blue yellow magenta
do
        break
done
echo "You chose $name."
```

When run, it looks like this:

```
$ ./myscript.sh
```

```
Choose from the list below.
1) red
2) green
3) blue
4) yellow
5) magenta
Choose (1-5):4
You chose yellow.
```

As written, this menu code won't catch some kinds of errors (like a number that is out of range). In any application where the user choice must fall into defined bounds, be sure to perform a test on the result before using it. Example:

```
if [ "$name" = "" ]; then
        echo "Error in entry."
        exit 1
fi
```

An advanced example with numbers and user input

- Here is an example guessing game that ties together some of the elements we've covered so far:

```
secretNumber=$(( ((`date +%N` / 1000) % 100) +1 ))
guess=-1
while [ "$guess" != "$secretNumber" ]; do
        echo -n "I am thinking of a number between 1 and 100. Enter your guess:"
        read guess
        if [ "$guess" = "" ]; then
                echo "Please enter a number."
        elif [ "$guess" = "$secretNumber" ]; then
                echo -e "\aYes! $guess is the correct answer!"
        elif [ "$secretNumber" -gt "$guess" ]; then
                echo "The secret number is larger than your guess. Try again."
        else
                echo "The secret number is smaller than your guess. Try again."
        fi
done
```

Please study this example carefully, and refer to the reference materials in Useful Links to understand some of the methods.

Creating and using arrays

- Shell arrays are relatively easy to construct. Example:

```
array=(red green blue yellow magenta)
len=${#array[*]}
echo "The array has $len members. They are:"
i=0
while [ $i -lt $len ]; do
        echo "$i: ${array[$i]}"
        let i++
done
```

Run this example:

```
$ ./myscript.sh
```

```
The array has 5 members. They are:
0: red
1: green
2: blue
3: yellow
4: magenta
```

Now, before you decide this is a silly, rather useless example, replace one line of the script and run it again:

```
array=(`ls`)
```

See what difference this makes (and think of all the kinds of lists you might create for this line).

Strings and substrings

- It's useful to be able to take strings apart and put them together in different ways. Here is how to select a substring from a string:

```
string="this is a substring test"
substring=${string:10:9}
```

In this example, the *variable* "substring" contains the *word* "substring". Remember this rule:

```
substring=${string_variable_name:starting_position:length}
```

The string starting position is zero-based.

Searching and Replacing Substrings within Strings

- In this method you can replace one or more instances of a string with another string. Here is the basic syntax:

```
alpha="This is a test string in which the word \"test\" is replaced."
beta="${alpha/test/replace}"
```

The string "beta" now contains an edited version of the original string in which the *first* case of the word "test" has been replaced by "replace". To replace *all cases*, not just the first, use this syntax:

```
beta="${alpha//test/replace}"
```

Note the double "//" symbol.

Here is an example in which we replace one string with another in a multi-line block of text:

```
list="cricket frog cat dog"
poem="I wanna be a x\n\
A x is what I'd love to be\n\
If I became a x\n\
How happy I would be.\n"
for critter in $list; do
        echo -e ${poem//x/$critter}
done
```

Run this example:

```
$ ./myscript.sh
```

```
I wanna be a cricket
A cricket is what I'd love to be
If I became a cricket
How happy I would be.
I wanna be a frog
A frog is what I'd love to be
If I became a frog
How happy I would be.
I wanna be a cat
A cat is what I'd love to be
If I became a cat
How happy I would be.
I wanna be a dog
A dog is what I'd love to be
If I became a dog
How happy I would be.
```

Silly example, huh? It should be obvious that this search & replace capability could have many more useful purposes.

More obscure but useful string operations

- Here is a way to isolate something useful from a large, even multi-line, string. As above, this method relies on enclosing a variable name in curly braces, then aplying a special operator to achieve a particular result.

  Here is a list of four such operators:

  - Operator "#" means "delete from the left, to the first case of what follows."

    ```
    $ x="This is my test string."
    $ echo ${x#* }

    is my test string.
    ```

  - Operator "##" means "delete from the left, to the last case of what follows."

```
$ x="This is my test string."
$ echo ${x##* }

string.
```

- Operator "%" means "delete from the right, to the first case of what follows."

```
$ x="This is my test string."
$ echo ${x% *}

This is my test
```

- Operator "%%" means "delete from the right, to the last case of what follows."

```
$ x="This is my test string."
$ echo ${x%% *}

This
```

I find these operators particularly useful in parsing multi-line strings. Let's say I want to isolate a particular IP address from the output of the "ifconfig" command. Here's how I would proceed:

```
$ x=`/sbin/ifconfig`
$ echo $x

eth0      Link encap:Ethernet  HWaddr 00:0D:56:0C:8D:10
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::20d:56ff:fe0c:8d10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:253339 errors:0 dropped:0 overruns:0 frame:0
          TX packets:423729 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:36150085 (34.4 MiB)  TX bytes:496768499 (473.7 MiB)
          Base address:0xecc0 Memory:fe4e0000-fe500000
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:109394 errors:0 dropped:0 overruns:0 frame:0
          TX packets:109394 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12372380 (11.7 MiB)  TX bytes:12372380 (11.7 MiB)
```

There's lots of information, more than we need. Let's say for the sake of argument that I want the IP of "lo", the loopback interface. I could specify this:

```
$ y=${x#*inet addr:}
```

But, while gobbling text from the left, this search would stop at the IP address of eth0, not the desired interface. So I can specify it this way:

```
$ y=${x#*lo *inet addr:}
```

As a last step I'll trim off all remaining text to the right:

```
$ y=${y%% *}
```

Leaving only the desired address.

It seems the "#" and "%" operators, and their variants, are able to accept a rather complex argument and sort through the content of large strings, including strings with line breaks. This means I can use the shell to directly filter content in some simple cases where I might have considered using sed or Perl.

Bash Version 3

I have always thought the inability to test for the presence of a string or pattern (without using grep, sed or something similar) was a conspicuous weakness in shell programming. Bash version 3, present on must current Linux distributions, addresses this lack by allowing regular expression matching.

Let's say we need to establish whether variable $x appears to be a social security number:

```
if [[ $x =~ [0-9]{3}-[0-9]{2}-[0-9]{4} ]]
then
        # process SSN
else
        # print error message
fi
```

Notice the Perlish "=~" syntax and that the regular expression appears within *double brackets*. A substantial number of regular expression metacharacters are supported, but not some of the Perl-specific extensions like \w, \d and \s.

Another Bash 3 feature is an improved brace expansion operator:

```
$ echo {a..z}

a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
for n in {0..5}
do
        echo $n
done

0
1
2
3
4
5
```

Useful Links

Well, as long-winded as it turned out to be, this page is supposed to be an introduction to shell programming, one that just touches the highlights. The linked references below will provide a deeper understanding of the covered topics.

- A quick guide to writing scripts using the bash shell (Rutgers)
- Advanced Bash Scripting Guide (Linux Documentation Project)
- Bash Reference Manual (GNU Project, downloadable versions)