To those who know nothing about UNIX, Linux, and Bash

Denis Jacob Machado

Contents

Table 1: What does BASH stands for?

Acronym	Definition
BASH	Bourne Again Shell
BASH	Bird/wildlife Aircraft Strike Hazard
BASH	Bridged Amplifier Switching Hybrid
BASH	Bandwidth Sharing
BASH	Blue Ash YMCA (Cincy)
BASH	Bay Area Siberian Husky Club
BASH	Bulimia Anorexia Self-Help, Inc.
BASH	Bikers Against Statewide Hunger (WA)
BASH	Baptists Are Saving Homosexuals
BASH	Building A Scholastic Heritage
BASH	Brit. Assoc. for the Study of Headache

1 What is Bash?

Bash (see the definition of the acronym on Table 1) is a UNIX shell and command language written by Brian Fox for the GNU Project (a mass collaborative initiative for the development of free software founded by Richard Stallman in 1978 at MIT; see Fig. 3) as a free software replacement for the Bourne shell.

The bash was first released in 1989. Since then, it has been distributed widely as the default login shell for most Linux distributions and Apple's macOS (formerly OS X). A version is also available for Windows 10.

You can learn some Bash programming at CodeAcademy.

What does GNU stands for?

$GNU = GNU\ is\ Not\ UNIX$

GNU is a *recursive acronym*. Its a play on words that is very common in the opensource community (*e.g.*, $PHP = PHP : Hypertext\ Preprocessor$ or $WINE = Wine\ Is\ Not\ an\ Emulator$).

Figure 1: The Linux logo is a penguin called "Tux", which stands for "**T**orvalds **U**ni**X**." A tattoo of Tux is called a "Tuxtoo."

2 What is Linux?

Just like Windows XP, Windows 7, Windows 8, and Mac OS X, Linux (see the logo on Fig. 1, and read more about it here) is an operating system (OS). An OS is a software that manages all of the hardware resources associated with your computer. In lay terms, the OS handles the communication between your software and your hardware. Without the OS, other software wouldn't function.

3 Main components of an OS

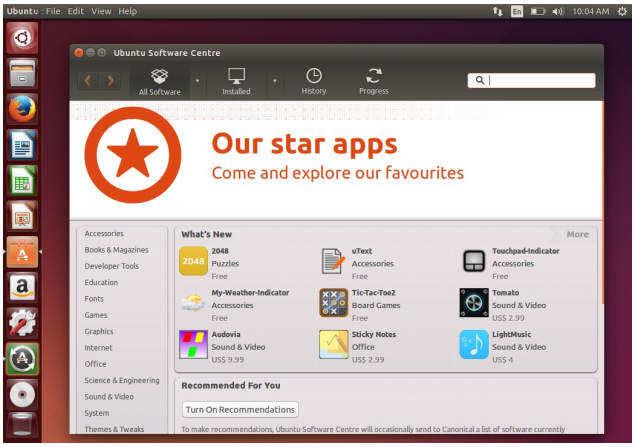
- **The Bootloader:** The software that manages the boot process of your computer. For most users, this will merely be a splash screen that pops up and eventually goes away to boot into the operating system.

- **The Kernel:** This is the one piece of the whole that is called "Linux." The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the "lowest" level of the OS.

- **Daemons:** These are background services (e.g., printing, sound, scheduling) that either start during boot or after you log into the desktop.

- **The Shell:** Its the Linux command line – a command process that allows you to control the computer via commands typed into a text interface. The shell is what, at one time, scared people away from Linux the most (assuming they had to learn a seemingly archaic command line structure to make Linux work). But this is no longer the case. With modern desktop Linux, there is no need ever to touch the command line.

Figure 2: Screenshot of an Ubuntu Desktop.



- **Graphical Server:** This is the sub-system that displays the graphics on your monitor. We often refer to the Graphical Server as the X server or just "X."

- **Desktop Environment:** This is the piece of the puzzle that interacts with the user. There are many desktop environments to choose from (e.g., Unity, GNOME, Cinnamon, Enlightenment, KDE, XFCE). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, and games).

- **Applications:** Desktop environments do not offer the full array of apps. Just like Windows and Mac, Linux gives thousands upon thousands of high-quality software titles that can be easily found and installed. Most modern Linux distributions (more on this in a moment) include App Store-like tools that centralize and simplify application installation. For example, Ubuntu Linux has the Ubuntu Software Center (Fig. 2) which allows you to quickly search among the thousands of apps and install them from one centralized location.

4 What is the relation between GNU and Linux?

GNU (GNU's Not Unix) is a Unix-like operating system entirely composed of free software and no Unix code.

Linux is a Unix-like Kernel developed by Linus Torvalds with a bit of help from people around the world.

This Linux kernel, when combined with GNU operating system, forms the so-called "Linux" Operating system or in more accurate terms "GNU/Linux" (read more here).

Figure 3: The GNU logo.

5 What is UNIX?

The Linux is a UNIX-like system. UNIX is a family of multitasking, multiuser computer operating systems that derive from the original AT&T NIX, development starting in the 1970s at the Bell Labs research center by Ken Thompson, Dennis Ritchie, and others.

Initially intended for use inside the Bell System, AT&T licensed UNIX to outside parties in the late 1970s, leading to a variety of both academic and commercial UNIX variants from vendors like the University of California, Berkeley (BSD), Microsoft (Xenix), IBM (AIX), and Sun Microsystems (Solaris). In the early 1990s, AT&T sold its rights in UNIX to Novell, which then sold its UNIX business to the Santa Cruz Operation (SCO) in 1995. The UNIX trademark passed to The Open Group, a neutral industry consortium, which allows the use of the mark for certified operating systems that comply with the Single UNIX Specification (SUS). As of 2014, the UNIX version with the largest installed base is Apple's macOS.

UNIX systems are characterized by a modular design that is sometimes called the "UNIX philosophy". This concept entails that the operating system provides a set of simple tools that each performs a limited, well-defined function, with a unified filesystem as the primary means of communication, and a shell scripting and command language to combine the tools to perform complex workflows. UNIX distinguishes itself from its predecessors as the first portable operating system: almost the operating system as a whole is written in the C programming language, thus allowing UNIX to reach numerous platforms.

Figure 4: The UNIX plate. "Live Free or Die" was New Hampshire's state motto when the plate was designed by Armando P. Stettner inspired by the license plate on Bill and Karen Shannon's Datsun 280ZX.

6 History

You may find helpful to know a little bit about UNIX and it's history. The following is a timeline full of gaps that tries to summarize the most important events of UNIX history. Click here for additional details. Also, check a simplify diagram of UNIX history on Fig. 5 (click here to see the original).

- **1970's:** The UNIX brand has traditionally been applied to the family of multitasking, multiuser computer operating systems that derive from the original AT&T UNIX operating system, developed in the 1970s at the Bell Labs research center by Ken Thompson, Dennis Ritchie, and others.

- **Early 1980's:** Since it began to escape from AT&T's Bell Laboratories in the early 1970's, the success of the UNIX operating system has led to many different versions: recipients of the (at that time free) UNIX system code all began developing their different versions in their own, different, ways for use and sale. Universities, research institutes, government bodies and computer companies all began using the powerful UNIX system to develop many of the technologies which today are part of a UNIX system. In the early 1980's, the market for UNIX systems had grown enough to be noticed by industry analysts and researchers. Now the question was no longer "What is a UNIX system?" but "Is a UNIX system suitable for business and commerce?" The UNIX plate is born (Fig. 4).

- **Mid-1980's:** Although UNIX was still owned by AT&T, the company did little commercially with it until the mid-1980's. Then the spotlight of X/Open showed clearly that a single, standard version of the UNIX system would be in the wider interests of the industry and its customers. The question now was, "which version?."

- **Late 1980's:** In a move intended to unify the market in 1987, AT&T announced a pact with Sun Microsystems, the leading proponent of the Berkeley derived strain of UNIX. However, the rest of the industry viewed the development with considerable concern. Believing that their own markets were under threat they clubbed together to develop their own "new" open systems operating system. Their new organization was called the Open Software Foundation (OSF; which is not associated to the Free Software Foundation, FSF). In response to this, the AT&T/Sun faction formed UNIX International. This resulted in the "UNIX wars" that divided the system vendors between these two camps clustered around the two dominant UNIX system technologies: AT&T's System V and the OSF system called OSF/1. In the meantime, X/Open Company held the center ground. It continued the process of standardizing the APIs necessary for an open operating system specification.
- **Early 1990's:** UNIX System Laboratories (USL) becomes a company - majority-owned by AT&T. Linus Torvalds commences Linux development. Solaris 1.0 debuts.
- **Mid-1990's:** The Open Group forms a merger of OSF and X/Open.
- **Late 1990's:** The UNIX system reached its 30th anniversary in 1999. Linux 2.2 kernel released. The Open Group and the IEEE commence joint development of a revision to POSIX and the Single UNIX Specification. First LinuxWorld conferences. Dotcom fever on the stock markets. Tru64 UNIX ships.
- **Currently:** Apple reports 50 million desktops and growing – these are Certified UNIX systems.

Do it yourself

Read this additional materials:

- The Cathedral and the Bazaar
- Why Open Source misses the point of Free Software

Answer the following questions:

1. What is the difference between open and free software?
2. Which Creative Commons license best represents the idea of open science?
3. Which GNU license best represents the idea of free science?

7 Basic Unix and shell commands

Here, we will see a few Tcsh shell commands, Unix commands and special key strokes.

Note

For getting help in Unix, try the "manual" pages: `$ man <COMMAND>`.

- Shell commands are particular to the shell (tcsh, in this case).
- Unix commands are common to all Unix systems, though options vary a bit.
- Special characters may apply to Unix in general, or be particular to a shell.

8 Shell Commands

A shell command is one that is processed internally by the shell. There is no corresponding executable program.

Take cd for instance. There is no /bin/cd program, say, and which cd specifies that it is a built-in command. It is part of the shell. The cd command is processed by the shell (tcsh or bash, say).

Contrast that with ls. There is a /bin/ls program, as noted by which ls. The shell does not process ls internally.

See also

For geting help from the shell, use `man`, *e.g.*

- `$ man tcsh`
- `$ man bash`

8.0.1 bg

Put the most recently accessed job (child process) in the background.

This is most commonly entered after **ctrl-z**. One might also use the output of **jobs**, and specify a job number.

8.0.2 cd

Change directories.

The *cd* command is used to change directories. Without any parameters, the shell will change to the user's home directory (`$HOME`). Otherwise, it will change to the single specified directory.

Examples

```
$ cd
$ cd $HOME
$ cd suma_demo
$ cd AFNI_data6/FT_analysis
$ cd ~username
$ cd /absolute/path/to/some/directory
$ cd relative/path/to/some/directory
$ cd ../../some/other/dir
```

Note: It is a good idea to follow a cd command with ls to list the contents of the new directory.

Since .. is the parent directory, including such entries will travel up the directory tree.

8.0.3 echo

Echo text to the terminal window.

The echo command displays the given text on the terminal window. This is often used to show the user what is happening or to prompt for a response. There are more devious uses, of course...

Example

```
$ echo hello there
$ echo "this section performs volume registration"
$ set my_var = 7
$ echo "the value of my_var = $my_var"
```

You should try the commands in every example on your own terminal window, and aks for help in case you do not understand something.

8.0.4 fg

Put the most recently accessed job (child process) in the foreground.

This is most commonly entered after **ctrl-z**. One might also use the output of **jobs**, and specify a job number.

While **bg** keeps a process running, but leaves the terminal window available for new commands, **fg** puts a process in the foreground, so commands would no longer be available.

8.0.5 jobs

List processes started from current shell.

The **jobs** command lists processes that have been started from the current shell (the current terminal window, probably) that are either suspended or running in the background. Processes are suspended via **ctrl-z**, and can then be put into the background using the **bg** command (or by using **&** in the first place).

Example

```
$ jobs
```

Probably shows nothing...

8.0.6 set

Assign a value to a shell variable.

The set command assigns a value to a variable (or multiple values to multiple variables). Without any options, all set variables are shown.

If a value has spaces in it, it should be contained in quotes. If a list of values is desired, parentheses () should be used for the assignment, while indivisual access is done via square brackets [].

Example

```
$ set
$ set value = 7
$ set new_val = "some number, like seven"
$ echo $new_val
$ set list_of_vals = ( 3 1 4 one five )
$ echo $list_of_vals
$ echo $list_of_vals[3]
$ set path = ( $path ~/abin )
```

A single set command can be used to set many variables, but such a use is not recommended.

Note

The **set** command is for setting shell variables, which **do not** propagate to child shells. To propagate to a child shell, use **environment variables**. A child shell would be created when a new shell is started, such as when running a script. Use **setenv** to set an environment variable.

Examples:

```
$ setenv DYLD_FALLBACK_LIBRARY_PATH $HOME/abin
```

8.0.7 unset

Delete a shell variable. This does not just clear the variable, but makes it “not exist”.

Example

```
$ set value = 7
$ echo $value
$ unset value
$ echo $value
```

The result of the last echo `$value` command would produce an error, since that variable no longer exists.

Note

Use **unsetenv** to delete an environment variable. This also does not just clear the variable, but makes it "not exist".

Examples:

```
$ unsetenv DYLD_FALLBACK_LIBRARY_PATH
$ unsetenv AFNI_NIFTI_DEBUG
```

9 Unix Commands

A Unix command is a command that refers to an actual file on disk. There is a `/bin/ls` file, but there is no file for `cd`.

Example commands to consider:

```
$ which cat
$ which cd
$ which ls
$ which afni
```

9.0.1 cat

It will display file contents in terminal window.

The **cat** command, short for *catenate*, is meant to dump all files on the command line to the terminal window.

Example

```
$ cat FT_analysis/s01.ap.simple
$ cat here are four files
$ cat here are four files | wc -l
```

Please note that none of the paths above are real. Test the commands with real paths from your computer. See also `$ man cat`.

9.0.2 gedit

It is a text editor for the GNOME Desktop.

The **gedit** program is a graphical text editor that works well across many Unix-like platforms. If you are not sure which editor to use, gedit is a good option. But it often needs to be explicitly installed.

Examples

```
$ gedit
$ gedit my_script.txt
$ gedit output.proc.subjFT.txt
```

See also:

- `$ man gedit`
- wiki.gnome.org/Apps/Gedit
- en.wikipedia.org/wiki/Gedit

9.0.3 less

A text file viewer.

The **less** is a terminal pager program on Unix, Windows, and Unix-like systems used to view (but not change) the contents of a text file one screen at a time. It is similar to **more**, but has the extended capability of allowing both forward and backward navigation through the file. Unlike most Unix text editors/viewers, less does not need to read the entire file before starting, resulting in faster load times with large files.

The less can be invoked with options to change its behaviour, for example, the number of lines to display on the screen. A few options vary depending on the operating system. While less is displaying the file, various commands can be used to navigate through the file. These commands are based on those used by both more and vi. It is also possible to search for character patterns in the file.

By default, less displays the contents of the file to the standard output (one screen at a time). If the file name argument is omitted, it displays the contents from standard input (usually the output of another command through a pipe). If the output is redirected to anything other than a terminal, for example a pipe to another command, less behaves like cat.

The command-syntax is:

```
$ less [options] [file_name]
```

Frequently used options:

- `-g`: Highlights just the current match of any searched string.
- `-I`: Case-insensitive searches.
- `-M`: Shows more detailed prompt, including file position.
- `-N`: Shows line numbers (useful for source code viewing).

- `-S`: Disables line wrap ("chop long lines"). Long lines can be seen by side scrolling.
- `-X`: Leave file contents on screen when less exits.
- `-?`: Shows help.
- `+F`: Follow mode for log.

9.0.4 ls

list directory contents

The **ls** command lists the contents of either the current directory or the directories listed on the command line. For files listed on the command line, it just lists them.

Multiple directories may be listed, in which case each directory is shows one by one.

Example

```
$ ls
$ ls $HOME
$ ls AFNI_data6/afni
$ ls AFNI_data6/afni FT_analysis/FT ~
$ ls -al
$ ls -ltr
$ ls -ltr ~
$ ls -lSr dir.with.big.files
```

Common options:

- `-a`: list all files/directories, including those starting with '.'
- `-l`: use the long format, which includes:
  - type
  - permissions
  - ownership
  - size
  - date (may vary per OS)
- `-t`: sort by time (modification time)
- `-r`: reverse sort order
- `-S`: sort by size of file (e.g. 'ls -lSr')

9.0.5 pwd

Display the full path to the "Present Working Directory".

9.0.6 tcsh

The **t-shell**. This shell (user command-line interface) is an expanded **c-shell**, with syntax akin to the `C` programming language. One can start a new **tcsh** shell by running this command, or one can tell the shell to interpret a script.

Example

```
$ tcsh
$ tcsh my.script.txt

$ tcsh -xef proc.subj1234 |& tee output.proc.subj1234
```

10 Special characters

Special characters and keystrokes.

Special **characters** refer to those that have special functions when used in **tcsh** command or scripts.

Special **keystrokes** refer to those that apply to a terminal window shell with sub-processes.

10.0.1 Ampersand (&)

Run some command in the background.

Putting a trailing **&** after a command will have it run in the background, akin to omitting it and typing **ctrl-z** followed by **bg**.

Example

```
$ suma -spec subj_lh.spec -sv SurfVol+orig &
$ tcsh run.my.script &
```

Some other uses for **&** include conditional (**&&**) and bitwise ANDs (**&**), as well as piping (**|&**) and redirection (**>&**) of stderr (standard error).

10.0.2 Backslash (\)

Line continuation (or escaping).

Putting a trailing \at the end of a command line tells the shell that the command continues on the next line. This could continue for many lines.

Examples:

```
$ echo this is all one long  \
     command, extending over \
     three lines
```

Note that the latter two lines were indented only to clarify that echo was the actual command name, while the other text items were just parameters.

Another use is to tell the shell not to interpret a special character or an alias.

More examples:

```
$ ls $HOME
$ ls \$HOME
$ echo *
$ echo \*
$ ls
$ \ls
```

Some programs allow for a similar interpretation (and other interpretations).

10.0.3 Hash (#)

The pound/ hash character: apply as comment or return list length.

The pound character has two main uses in a t-shell script: to start a comment or to return the length of an array.

In a shell script, if # is not hidden from the shell (in quotes or escaped with \), then from that character onward is ignored by the shell, as if it were not there. The point of this is to allow one to add comments to a script: text that merely tells a reader what the script is intending to do.

For example, if a t-shell script had these lines:

```
set greeting = pineapple
# check whether user wants to say "hi" or "hello"
if ( $greeting == hi ) then
  # the short greeting
  echo hi there
else
  echo hello there # strange place for a comment
endif
```

Then the "check whether user wants" line does not affect the script, nor does the comment "this is a strange place for a comment".

The output is simply, "hello there".

Note

Hash characters entered at the command line are not treated as comments, they are treated as any other simple text (possibly because the shell authors did not see any reason why one might want comments at the command line, such as for when cutting and pasting scripts).

Another use of # is to get the length of a shell array variable, such as `$PATH`.

Example

```
$ echo my path has $#path directories in it
$ echo the full list is: $path
```

Note: this use does not apply to environment variables, such as `$PATH`.

10.0.4 Single quotes (')

Enclosing text in single quotes tells the shell not to interpret (most of) the enclosed special characters. This is particularly important for cases where special characters need to be passed to a given program, rather than being interpreted by the shell.

With respect to scripting, the most important difference between single and double quotes is for enclosed $ characters, such as with `$HOME`, `$3` or something like `$value`. Such variable expansions would occur within double quotes, but not within single quotes.

Note

Back quotes ` are very different from single ' or double " quotes

Examples:

```
3dcalc -a r+orig'[2]' -expr 'atanh(a)' -prefix z
awk '{print $3}' some.file.txt
echo 'my home directory is $HOME'
```

The first example uses **3dcalc** to convert a volume of r-value (correlation values) via the inverse hyperbolic tangent function (a.k.a. Fisher's z-transformation). The first set of quotes around `[2]` hide the [and] characters from the shell passing them on to 3dcalc. Then the 3dcalc program knows to read in only volume #2, ignoring volumes 0, 1 and anything after 2.

If the [and] characters were not protected by the quotes, it would likely lead to a "No match" error from the shell, since the square brackets are used for wildcard file matching.

Alternatively, the quotes could alter go around the entire `r+orig[2]`.

The quotes around `atanh(a)` are to hide the (and) characters, again so that 3dcalc sees that entire expression.

The second example hides both the { and } and $ characters. Note that $ is most commonly used to access variable values, such as in `$HOME`.

The third example just clarifies that shell variables are not expanded, since the output shows `$HOME` and not `/home/rickr`, for example.

10.0.5 Double quotes (")

Double quotes.

Enclosing text in double quotes tells the shell not to interpret some of the enclosed special characters, but not as many as with single quotes. This is particularly important for cases where special characters need to be passed to a given program, rather than being interpreted by the shell.

With respect to scripting, the most important difference between single and double quotes is for enclosed $ characters, such as with `$HOME`, `$3` or something like `$value`. Such variable expansions would occur within double quotes, but not within single quotes.

Example

```
$ 3dcalc -a r+orig"[$index]" \
    -expr "atanh(a)" -prefixz
$ echo "my home directory is $HOME"
```

These examples just demonstrate use of variables within double quotes. The first one uses `$index` as a sub-brick selector with AFNI's *3dcalc* program. In this case, `$index` might expand to 2, as in the example using single quotes.

The second example (with `$HOME`) is similar to the one with single quotes. But the double quote output shows `$HOME` expanded to the home directory (e.g. `/home/rickr`), while the single quotes output does not (it still shows `$HOME`).

10.0.6 Back quote (`)

Back quotes (`) are very different from single(´) or double quotes ("). While single and double quotes are commonly used for hiding special characters from the shell, back quotes are used for command expansion.

When putting back quotes around some text, the shell replaces the quoted text with the output of running the enclosed command. Examples will make it more clear.

Example

```
$ echo my afni program is here: `which afni`
$ count -digits 2 1 6
$ set runs = "`count -digits 2 1 6`"
$ echo there are $#runs runs, \
    indexed as: $runs
$ set runs = ( `count -digits 2 1 6` )
$ echo there are $#runs runs, \
    indexed as: $runs
```

The first example runs the command `which afni`, and puts the result back into the echo command. Assuming `afni` is at `/home/rickr/abin/afni`, the first command is as if one typed the command:

```
$ echo my afni program is \
    here: /home/rickr/abin/afni
```

The second example line (`count -digits 2 1 6`) simply shows the output from the `afni` count program, zero-padded 2 digit numbers from 1 to 6.

The third line captures that output into a variable. Going off on a small tangent, that output is stored as a single value (because of the double quotes).

The fourth line displays that output in the terminal window. In this case, the `$runs` variable has only 1 (string) value, with spaces between the 6 run numbers.

The fifth line (again with set runs) sets the `$run` variable using parentheses, storing the output as a list (an array) of 6 values).

The final echo line shows the same output as the previous echo line, except that now it shows that there are indeed 6 runs.

This document was created on Nov. 13, 2018 and last updated on October 18, 2018.

Figure 5: The key Unix and Unix-like operating systems.