Using the Command Line

Josh Buchbinder <josh.buchbinder@obscuradigital.com>

What this is

What this will *not* be is a list of commands and explanations of what they do. This is intended to teach fundamental concepts common to most environments and programs and to teach how to get and understand the help available. It is assumed that the reader knows the very basics of command line navigation such as `cd, dir, ls, copy, cp`. The intent is to teach the fundamental terminology and understanding so that the reader can better search for and understand help on their own and become more comfortable using text interfaces by understanding what is actually happening in the operating system when commands are executed. It is important to remember that there are many different shells for every OS and that on different operating systems the shells or commands or programs may have slightly different functionality or usage or appearance or messages (or fonts).

History

As Linux evolved from Unix, so too Windows NT evolved from DOS (technically OS/2 evolved from DOS and NT evolved from OS/2.) One of the original shells or command processors for Unix was the Bourne Shell, /bin/sh. These days the most common default shell for most distributions on Linux and OSX is now bash, created as a replacement for sh. It is an acronym for Bourne Again Shell. (Get it?) Unix was not free software or often open source so bash was created to replace sh, incorporating the functionality of the Bourne shell and expanding on it.

The DOS shell or command processor was command.com. This persisted into the Windows 95 family of operating systems that were really just a program running under DOS. With the Windows NT family of operating systems a new default shell was introduced, CMD.exe, which like bash also maintained most of the functionality of its predecessor while also continuing to add more.

A shell is a program that offers the user an interface to the operating system allowing access to the file system and the ability to execute other programs. We will focus on text shells (AKA command processors or command line interfaces) which interface with the user via a text command line and can also interpret scripts. There are of course other shells, for instance the package **cygwin** is a collection of Linux like programs and environments for Windows and includes a bash shell for Windows. Windows now also has a more advanced and powerful text shell called **powershell**.

As MacOS is now based on a Linux or Unix type architecture from now on “Linux” applies to MacOS as well.

Current Working Directory

The **current working directory** (CWD) is an important concept as it represents “where we are now” in the file system, the current location that any program is working in at any time. Most shells show you the CWD in the **prompt**, the text that is displayed when the shell is awaiting user input:

```
D:\pics>
root@edison1:/usr/bin#
```

Later we will see how we can change the **prompt**. The command “`cd`” is common to most shells, short for “change directory” and is used to change the **CWD**. On Windows if you type “`cd`” by itself it will tell you the CWD. On Linux if you type “`cd`” by itself it will return you to the user home directory, but the “`pwd`” command will print your current working directory (print working directory).

```
D:\>cd
D:\

obscura@ubu:~$ pwd
/home/obscura
```

In cmd.exe you just put in the drive letter and a colon to change drives and Windows remembers the current directory for each drive letter. However, if you use the `cd` command with a drive letter that is not the current drive it does NOT change the current working directory, it instead changes the current directory for the other drive letter:

```
C:\Users\Josh>d:
D:\>cd c:\windows
D:\>c:
c:\Windows>
```

Paths

The file system is a container representing the data available to the user, which is represented a container containing files and directories/folders (*referred to as just ‘directories’ from now on*) and the sub directories are also containers containing other directories and files. The key difference between operating systems is that Windows has the concept of drive letters where in Linux there is a single path tree containing all of the drives in the location they were mounted. For Linux-like shells running on Windows this is usually represented as the drive letter at the root of the file system or in another subdirectory such `/c/` or `/cygdrive/c/` which equates to c: in cmd.exe and is the equivalent of where they were mounted.

A key difference between path representations between operating systems is the character that separates the elements in the paths, on Windows this is the **back slash** “\” and on Linux it is a **forward slash** “/”.

NOTE
In some cases Windows will accept paths with forward OR back slashes which makes copy/paste easier:

```
D:\>cd /pics
D:\pics>
```

ENDNOTE

Another key difference between paths is that Linux is **CASE SENSITIVE** while Windows is not:

```
C:\>cd windows
C:\Windows>cd ..
C:\>cd WINDOWS
C:\Windows>
```

This works on Windows, while on Linux:

```
root@edison1:/# cd usr
root@edison1:/usr# cd ..
root@edison1:/# cd USR
-sh: cd: can't cd to USR
```

A path is a string representing the location of a specific file or directory (referred to as a resource from now on), this is how the operating system can locate that specific resource. There are two ways to represent a path; a **fully qualified path** or a **relative path**. On Windows **fully qualified paths** begin with a drive letter and contain the whole path to the resource such as:

```
c:\windows\system32
```

On Linux a **fully qualified path** begins at the file system root “/” and contains the whole path to the resource such as:

```
/usr/bin
```

A **relative path** depends on the **current working directory** and can contain the special directory names “.” or “..” (common to both operating systems), representing the current and parent directories, respectively. To change to a subdirectory we use a command like:

```
C:\Windows>cd system32
C:\Windows\System32>
```

This works because “system32” is a relative path to the “windows” directory, contained within it. To change back to the parent directory we use a command like:

```
C:\Windows\System32>cd ..
C:\Windows>
```

This works because “..” represents the parent directory of the current directory. If we use the command:

```
C:\Windows>cd .
```

```
C:\Windows>
```

We effectively do NOT change directory because "." represents the current directory in the path which starts at our CWD. For demonstration purposes, you could get stupidly redundant with relative paths if you want:

```
C:\Windows>cd .\system32\..\system32\.
C:\Windows\system32>
```

This works because the **relative path** points to first the current directory then the system32 subdirectory then back to its parent (back to the CWD) then BACK to system32 then the current directory, which is now system32.

On Linux there is another special character that can be used in relative paths which is "~", which represents the home directory of the current user. Linux also has the behavior that if you use "`cd`" with no arguments, it will change the CWD to the home directory. When you are in your home directory the prompt will usually show:

```
root@edison1:~#
```

instead of

```
root@edison1:/home/root#
```

Another aspect of the file system that is common to Linux and Windows is **links** or **junctions**, basically an alias in the file system that points to a different resource. This topic will not be covered but it is important to be aware of them and to recognise when a directory or file is actually a link pointing to somewhere else in the file system.

TAB Completion

A very useful way to speed up using the command line is by using the TAB key to complete the name of a file or directory but it works slightly differently between Linux and Windows. On Windows pressing TAB will cycle through all of the names that match what you have already started typing on the command line and SHIFT-TAB will cycle backwards. For instance if you are in C:\ and you type:

```
C:\>cd p
```

and then hit TAB it will change to:

```
C:\>cd PerfLogs
```

Pressing TAB again it will change to:

```
C:\>cd "Program Files"
```

Pressing SHIFT-TAB now will change it back to:

```
C:\>cd PerfLogs
```

cmd.exe is finding all of the directories that begin with "p" because that is what was typed into the command prompt before pressing TAB.

On Linux pressing TAB will only auto-complete names that uniquely match what has already been typed in, if there is more than one match pressing TAB once will do nothing, but pressing TWICE will list the names that match:

```
root@edison1:/# cd s
```

Pressing TAB now has no effect but pressing it again gets this:

```
sbin/    sketch/  sys/
root@edison1:/# cd s
```
Now if you type a “y” and then press TAB it will autocomplete:
```
root@edison1:/# cd sys/
```
because “sys” is the only name that begins with “sy”. It will however auto-complete up to the character where it can no longer find a unique name, for instance if you are in a directory with files named “file13”, “file25” and “file31” and you type:
```
root@edison1:~# cp f
```
then hit TAB it will autocomplete to:
```
root@edison1:~# cp file
```
waiting for you to provide the next unique character. At this point you could hit tab and it would show you “`file13 file25 file31`”. Note that directories are shown with a “/” at the end of their name to distinguish them from files.

Command history

In most shells, Windows and Linux, the up and down arrows will scroll through commands you have previously entered. After cycling to a command you want to re-enter, but with modifications, the cursor will be at the end of the command, then you can use the left and right arrows to move through that command and make modifications, then press enter to execute the command. In some interfaces such as cmd.exe using CONTROL-left/right will jump to the beginning of the next word or delimiter (such as / on Linux).

Mouse interaction

If using a command line interface in a window on a GUI desktop the mouse can be used to speed up cutting and pasting. In Windows many find it useful to set cmd.exe to “QuickEdit mode” (in Properties->Options) which enables functionality similar to Linux text shells; on both operating systems right clicking will paste the text clipboard into the shell as if it were keyboard input, and dragging with the left button selects text. The difference between the text selection functionalities is that on Windows you select a block (rectangle) of text and the clipboard receives each row of text followed by an end of line where on Linux it copies the text between the start and end of the drag wrapping text where there is no end of line, as you would expect in any text editor.

Windows GUI and command prompt interaction

Windows has a useful command: `start`. Start will launch a console process (or GUI process) in a new window and return immediately, running start by itself will open another command prompt window. However, “`start .`” or “`start c:\windows`” will open that directory in a GUI folder view. When navigating folders in the GUI you can quickly open a command prompt at the location of a folder by entering “`cmd`” in the address bar. A quick keyboard shortcut to highlight the address bar is `ALT-D` so if you are in a folder pressing `ALT-D` and then typing `cmd` and hitting enter will open a command prompt at that location. This works because it launches cmd.exe with the GUI folder as the current working directory. This allows you to easily move back and forth between between the text and GUI interface.

MacOS GUI and command prompt interaction

In a MacOS text shell the command “`open .`” or “`open /home`” will open a GUI folder view of the current or other directory.

NOTE: Most shells can be terminated using the “`exit`” command.

Command Line Arguments

Most commands or programs executed in the shell require some arguments meaningful to that command or program. Help is available for most built in commands and programs by using “`help command`” on Windows or “`man command`” on Linux. Most programs or commands also have their own help by typing “`command /?`” on Windows or “`command --help`” or “`command -h`” on Linux.
There is a somewhat common syntax to the help provided for commands, for example:

```
C:\>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:]attributes]] [/B] [/C] [/D] [/L] [/N]
  [/O[[:]sortorder]] [/P] [/Q] [/R] [/S] [/T[[:]timefield]] [/W] [/X] [/4]
```

Argument portions enclosed in square brackets `[]` are optional and can contain other optional nested portions. Usually arguments listed not enclosed in anything are required arguments, though sometimes required arguments are enclosed in curly braces `{}`. If there is a choice of mutually exclusive options they are separated with a pipe `|` character meaning *or*. Generally arguments shown in CAPS are literal strings (such as “`/A`”, although on Windows it is not usually necessary to use the same case when the user enters it) while lowercase arguments are replaced with either a user supplied string (such as `filename`) or a specific value described later in the help (such as `attributes`):

```
/A          Displays files with specified attributes.
attributes   D  Directories                R  Read-only files
             H  Hidden files               A  Files ready for archiving
             S  System files               I  Not content indexed files
             L  Reparse Points             -  Prefix meaning not
```

Note that the `[[:]attributes]` portion is enclosed in square brackets so is optional, meaning `/A` can be used alone without anything following it, and that the `[:]` portion is also enclosed in square brackets so you can optionally include a colon but it is not required. In this case “`dir /a`” alone will list all files with any attribute, including hidden and system files which are not normally shown, while “`dir /ad`” (or “`dir /a:d`”) will list only directories. Note that there is also an attribute code “`-  Prefix meaning not`” which will cause the inverse meaning of other codes if put before them, so “`dir /a-d`” will list everything that is NOT a directory.

Spaces in arguments

Since arguments are normally separated by spaces, to pass an argument containing a space you must surround the argument with quotes. In general it doesn't hurt to put quotes around an argument; they will not be passed to the command.

Wildcards

Many path or filename arguments can contain **wildcards**, special characters that mean "any". The most common are the asterix "*" meaning "any matching string" and the question mark "?" meaning "any matching character" (or the end of a string). For example "`dir s*`" will list all names starting with "s" while "`dir s?`" will display all names that are two characters long with the first character "s" AND the name "s". Wildcards can be used anywhere in a string and more than once; "`*s*`" will match any name containing an s, "`*w?`" will match any names with "w" as the second to last character or ending in "w".

Linux also supports other wildcards such as square brackets `[]` that will match a range or list of characters. So "`file[123].txt`" will match any name that begins with "`file`" and then has a 1 2 or 3 before the file extension. You can use a dash for a range so the above is the same as "`file[1-3].txt`".

One important difference between the way Linux and Windows handle wildcards is that on Windows an argument containing a wildcard is passed directly to the command or program and it is up to that program or command to interpret it and find the names that match but on Linux the **shell** finds matching names and expands the the command line entered replacing the argument with the wildcard into a list of the matching names. On Windows using the `dir` command with the `/S` option will list matching names in subdirectories as well, so the command:

```
D:\>dir/s v*
```

will list every name on the entire drive beginning with the letter "V". On Linux the `-R` option for the `ls` command is similar to the `dir/s` command except that the shell will try to expand the wildcard resulting in:

```
obscura@ubu:/home$ ls -lR v*
ls: cannot access v*: No such file or directory
```

This also has an effect on the way commands are structured. For instance on Windows if a single argument is passed to the `copy` command it is assumed that the current directory is the destination, which is why if you tell it to copy a file in your current directory but don't give it a destination you get an error:

```
D:\>copy tst.txt
The file cannot be copied onto itself.
        0 file(s) copied.
```

On Linux using the `cp` command you must always give it a destination, failing to do so can get you into trouble when using wildcards:

```
obscura@ubu:~$ ls file*
file1  file2
obscura@ubu:~$ cp file*
```
The above command line was actually expanded into "`cp file1 file2`" so this actually overwrote file2 with file1 which was probably not the desired result!

Environment variables

Every program maintains a list of **environment variables**, a list of named strings. The command "`set`" is used to display and change these values on both operating systems:
```
C:\>set FOO=This is a test
```
Another command common to both operating systems is "`echo`" which will print back the command line passed to it:
```
C:\>echo This is a test
This is a test
```
On the command line you can expand **environment variables** into their strings using the syntax "`%NAME%`" on Windows and "`$NAME`" on Linux:
```
C:\>echo %FOO%
This is a test
```
Environment variables can be read by any program and used however they like. For instance, cmd.exe uses the PROMPT variable to decide what to display at the command prompt:
```
C:\>echo %PROMPT%
$P$G
```
The $ codes tell cmd.exe how to display the prompt, $P means the current drive and path, $G means a greater-than sign. We can change the prompt to have a parentheses instead of a greater than by replacing the $G with $F:
```
C:\>set prompt=$P$F
C:\)
```
Or we could put an environment variable in:
```
C:\)set prompt=$P$C%USERNAME%$F$G
C:\(Josh)>
```
Linux has similar but more complex functionality, but bash uses the variable `PS1` for the primary prompt:
```
obscura@ubu:~$ echo $PS1
\[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\u@\h:\w\$
```
Variables can be changed in Linux using several commands, here we use `export`:
```
obscura@ubu:~$ export PS1="This is a prompt> "
This is a prompt>
```

One important variable common to most operating systems is PATH, which contains a list of directories that will be searched (in order) when a command is entered. These could be relative paths but are usually absolute. On Windows they are separated by semicolons and on Linux they are separated by colons:

```
C:\(Josh)>echo %PATH%
C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;D:\Program
Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\bin\; ...

This is a prompt> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/gam
es:/usr/local/games
```

One important difference in behavior is that on Windows the current directory is searched first whereas on Linux it is not, which is why it is necessary to put "`./`" in front of a filename if you want to execute a program in the current directory. You could add the current directory to the beginning of your path:

```
This is a prompt> export PATH=.:$PATH
This is a prompt> echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/g
ames:/usr/local/games
```

Another important point is that changing these variables will only affect that particular instance of the shell and any programs executed by that shell after a variable was set or changed. This is because programs inherit their environment from their parent when they are first run. If you close the shell or open a second instance it will not have the changes to the environment made in a different shell. To permanently change a variable on Windows you can use the `SETX` command:

```
C:\(Josh)>setx /?
SetX has three ways of working:
Syntax 1:
    SETX [/S system [/U [domain\]user [/P [password]]]] var value [/M]
```

Windows stores two types of "base variables" in the registry, one for the system and one for each user which can add to or override the system variables. Using `SETX`, by default it will only change the current user's variables, unless you use the `/M` option:

```
/M                     Specifies that the variable should be set in
                       the system wide (HKEY_LOCAL_MACHINE)
                       environment. The default is to set the
                       variable under the HKEY_CURRENT_USER
                       environment.
```

Remember that changing the variables in the registry will have no effect on programs currently running because they inherited their variables when they first started, with one exception: `EXPLORER.EXE` (which is a *graphical* shell) will update its environment as soon as the registry is changed, so if you launch a new program from Explorer after changing the global variables they will inherit the new changes.

In bash the base variables are set in the file .bashrc located in the user directory, a script that is executed by bash when it is launched. This file can be edited and new `export` commands added or existing ones modified. Like Windows there is also a non-user system version `/etc/bash.bashrc`.

Redirection and piping

One of the powerful aspects of using a text based interface is that the output of any program or command can be **redirected** into either a file or even to the input of another program or command. The syntax is very similar between operating systems. Let's use the `echo` command we used earlier along with the **output redirection operator** greater than > to create a file using redirection:

```
D:\>echo This is text > file.txt
```

This same syntax works on linux as well:

```
obscura@ubu:~$ echo This is text > file.txt
```

Now to view the contents of the file we just created we can use the `type` command on Windows and the `cat` command on Linux:

```
D:\>type file.txt
This is text

obscura@ubu:~$ cat file.txt
This is text
```

The path variable can be quite long so it might be useful to open it in an editor so we could `echo` the variable to a file:

```
D:\>echo %PATH% > path.txt

obscura@ubu:~$ echo $PATH > path.txt
```

Another useful thing we could do is combine several files into one using the `type` or `cat` commands:

```
D:\>echo THIS IS FILE 1 > file1.txt
D:\>echo THIS IS FILE 2 > file2.txt
D:\>type file?.txt > file_all.txt
D:\>type file_all.txt
THIS IS FILE 1
THIS IS FILE 2

obscura@ubu:~$ echo THIS IS FILE 1 >file1.txt
obscura@ubu:~$ echo THIS IS FILE 2 >file2.txt
obscura@ubu:~$ cat file?.txt > file_all.txt
obscura@ubu:~$ cat file_all.txt
THIS IS FILE 1
THIS IS FILE 2
```

The command name `cat` is actually short for concatenate so this type of usage is part of what it was designed for.
Another redirection operator is the **append redirection operator** `>>` which is similar to the output operator but instead of overwriting the target file if it exists it will instead append to the file:

```
D:\>echo Line1 >> out.txt
D:\>echo Line2 >> out.txt
D:\>echo Line3 >> out.txt
D:\>type out.txt
Line1
Line2
Line3
```

In both Linux and Windows programs that output text to the terminal can actually output it to two different handles known as “standard out” (STDOUT) and “standard error” (STDERR), though both are normally just displayed to the user. STDERR is usually used to display error messages but can be used for other things, for instance on Windows the `type` command sends the filename to STDERR before it sends the contents of the file to STDOUT. To redirect STDERR to a file use the **standard error redirection operator** 2>. You can redirect STDOUT and STDERR to different files by putting both redirection commands on the command line:

```
D:\>type *.txt > contents.txt 2> filenames.txt
```

To redirect STDOUT and STDERR to the same file, ie to get all of the output into a single file the syntax is a little different:

```
D:\>type *.txt > all.txt 2>&1
```

This means redirect STDERR to file handle 1 (File handle 0 is STDIN, 1 is STDOUT, 2 is STDERR). This is all true for and works in Linux as well.

You can use the **input redirection operator** less than `<` to have a program use the contents of a file as input, as if it was typed in by the user. A common need is to search for a specific string in a large amount of text. Windows comes with a program `find.exe` to do this, Linux has `grep`. First, let’s create a file containing the entire contents of the C drive by using the `dir` command with the `/S` option and the `/B` option to get just the fully qualified path names and redirect it into a text file:

```
D:\>dir/s/b c:\ > c_drive.txt
```

Now let’s run the `find` command redirecting its input from this file:

```
D:\>find "icon" < c_drive.txt
c:\AI_RecycleBin\{A69142C6-9692-4FCE-A7C3-3901A8DAD42E}\0\TouchDesigner088\bin\
iconv.dll
c:\MinGW\bin\iconv.exe
c:\MinGW\bin\libiconv-2.dll
...
```

This example demonstrates **input redirection**, but we could have also just run "`find "icon" c_drive.txt`" for the same results.

We could also skip the step of creating an intermediate file by using **piping** to pipe the output of the `dir` command directly into the `find` command by using the **pipe operator** |:

```
D:\>dir/s/b c:\ | find "icon"
```

Or on Linux using the `grep` command:

```
obscura@ubu:~$ ls -laR / | grep "Script"
-rw-rw-r--  1 obscura obscura  78695 Jul 28 16:58 RenderScript.ContextType.html
-rw-rw-r--  1 obscura obscura  89437 Jul 28 16:58 RenderScript.html
-rw-rw-r--  1 obscura obscura  77130 Jul 28 16:58 RenderScript.Priority.html
```

You can also chain piped commands. Another command common to Windows and Linux is `more` which takes input and displays it one screen at a time:

```
D:\>dir/s/b c:\ | find "icon" | more
c:\AI_RecycleBin\{A69142C6-9692-4FCE-A7C3-3901A8DAD42E}\0\TouchDesigner088\bin\
iconv.dll
c:\MinGW\bin\iconv.exe
c:\MinGW\bin\libiconv-2.dll
…
c:\Program Files (x86)\Adobe Media Player\assets\icons\app_icon_48.png
c:\Program Files (x86)\Adobe Media Player\assets\icons\f4v_icon_128.png
-- More  --
```

On Linux:

```
obscura@ubu:~$ ls -laR / | grep "Script" | more
-rw-rw-r--  1 obscura obscura  78695 Jul 28 16:58 RenderScript.ContextType.html
-rw-rw-r--  1 obscura obscura  89437 Jul 28 16:58 RenderScript.html
-rw-rw-r--  1 obscura obscura  77130 Jul 28 16:58 RenderScript.Priority.html
-rw-rw-r--  1 obscura obscura  64009 Jul 28 16:58 Script.Builder.html
--More--
```

The `more` command knows the height of the terminal window it is running in so it will pause after displaying one page of output. At the "`more`" prompt you can press space to display the next page or press enter to display the next line.

A more useful example of chained piping would be say you were looking for filenames with the string "top" in them. On Linux you might use something like this:

```
odguest@ubu:/$ ls -laR | grep "top"
```

But say most of the results were actually names containing "stop" which you did not want to see. You could pipe the output of the first `grep` command into a second `grep` command using the `-v` option which says "Display lines that do NOT contain the string":

```
odguest@ubu:/$ ls -laR | grep "top" | grep -v "stop"
```

Scripting

A **shell script** is at its base a text file with commands in it that are executed in order by the shell. .BAT files (batch files) are command.com's script files, .CMD files are cmd.exe's script files, and .sh files are Linux shell scripts. In addition to executing commands, scripts can contain logic that can change the execution order. They can also set and check environment variables to use in their logic. An old DOS logic command that is still supported on Windows is "`IF ERRORLEVEL n`" which returns true if the ERRORLEVEL is equal to or higher than `n`, 0 indicates no error. Consider the following basic batch file:

```
D:\>type sample.bat
copy /Y %WINDIR%\directx.log .
echo Error level is %ERRORLEVEL%
IF ERRORLEVEL 1 goto errorexit
echo SUCCESS
goto finalexit
:errorexit
echo FAILURE
:finalexit
```

The first line tries to copy the file `directx.log` from the Windows directory to the current directory. `%WINDIR%` is an environment variable that points to the WIndows directory (c:\windows). The `/Y` option for the `copy` command tells it to overwrite the file if it exists without asking the user. The second line displays the error level returned by the `copy` command, The third line checks if the `ERRORLEVEL` value is equal to or higher than one and if true it will jump to the `errorexit` label. Lines beginning in colons **:** are labels that can be used with the `goto` command.
When we run the program we get:

```
D:\>sample.bat
D:\>copy /Y C:\Windows\directx.log .
        1 file(s) copied.
D:\>echo Error level is 0
Error level is 0
D:\>IF ERRORLEVEL 1 goto errorexit
D:\>echo SUCCESS
SUCCESS
D:\>goto finalexit
```

After copying the file successfully `ERRORLEVEL` is NOT 1 or higher so it continues to the next line `echo`ing the text `SUCCESS` then jumps to the label `finalexit` to avoid displaying `FAILURE`. If we change the copy command to look for a file that doesn't exist and run the program we get this:

```
D:\>copy /Y C:\Windows\doesnotexist.txt .
The system cannot find the file specified.
D:\>echo Error level is 1
```

```
Error level is 1
D:\>IF ERRORLEVEL 1 goto errorexit
D:\>echo FAILURE
FAILURE
```

This time the `copy` command failed and returned an `ERRORLEVEL` value of 1, causing the `IF ERRORLEVEL` operation to return true causing it to jump to the `errorexit` label. In this example I allowed the script to display all of the commands it was executing for readability but most scripts start with the command “`echo off`” which prevents the shell from displaying the commands it is executing. You will however still see the output from those commands. If you wanted to hide the output of a command from the user you can use the **output redirection operator** to redirect the output to an empty void known as the NULL device, “`NUL`” on Windows and “`/dev/null`” on Linux.

```
D:\>copy /Y C:\Windows\doesnotexist.txt . > nul

obscura@ubu:/$ cp /doesnotexist.txt . > /dev/null
cp: cannot stat â/doesnotexist.txtâ: No such file or directory
obscura@ubu:/$ cp /doesnotexist.txt . 2> /dev/null
```

Notice that on Windows the error message went to STDOUT but on Linux it went to STDERR so it was necessary to redirect the output using `2>` instead of `>`.

Special devices and names

Both Windows and Linux have built in ‘devices’ that can be accessed as