Win32 Shell Scripting Tutorial

Ashley J.S Mills

<ashley@ashleymills.com>

Table of Contents

1. Introduction to Win32 Shell Scripting

Time is precious. It is non-sense-ical to waste time typing a frequently used sequence of commands at a command prompt, more especially if they are abnormally long or complex. Scripting is a way by which one can alleviate this necessity by automating these command sequences in order to make ones life at the shell easier and more productive. Scripting is all about making the computer, *the tool* do the work. Hopefully by the end of this tutorial you should have a good idea of the kinds of scripting languages available for Windows and how to apply them to your problems.

2. The Environment

The environment is an area of memory associated with the command processor that provides upto 32KB of space for storing variables, the variables contain information about the operating environment that is used by the operating system and other programs in various ways, typically to inform a program of the location of a certain piece of information it requires. A few examples follow:

- `ComSpec`, specifies the location of the command interpreter.
- `PATH`, specifies the locations to search for commands typed at the command line.
- `Prompt`, specifies how the command prompt should appear to the user.
- `AGE`, a user-defined variable for indicating the age of something.
- `TEMP`, specifies the directories where temporary files should be placed.

Sometimes it is necessary to modify certain environment variables, for instance, we may have installed a new program in a directory such as `C:\tools\chipper\`, and its executable files may be stored in `C:\tools\chipper\bin\`. We may desire to execute this program from the command line but find that we get an error saying something like:

```
'chipper' is not recognized as an internal or external command, operable program or batch file.
```

Because we had not updated the `PATH` to point to the location of the binary files for this program.

Altering environment variables will depend on what version of Windows you are using, see *Configuring A Windows Working Environment* [../winenvars/winenvarshome.html].

3. Batch Programming

A batch file is a plain ASCII text file with the file extension `.bat`, it is interpreted by the command processor, usually `com-`

`mand.com` or `cmd.exe`. Batch files are used to automate repetitive command sequences in the command shell environment.

In the context of batch programming, the environment is exploited for it's ability to store text strings within definable variables known as environment variables as illustrated in the above example, these variables can be used in batch files in much the same way as you would use variables in other programming languages albeit with less freedom. The language used in batch files is usually referred to as a scripting language but it is in-fact Turing-complete hence is actually a programming language.

3.1. Auxiliary files for extended batch programming

Unfortunately, due to reasons unknown, many useful tools for batch programming have been omitted in recent versions of Windows and are only available from within costly 'resource kits' that one has to purchase for seemingly extortionate sums of money. However, there are free alternatives to some of these tools and/or hacks/kludges to circumvent their necessity.

3.1.1. `OldDos.exe`

ftp://ftp.microsoft.com/softlib/mslfiles/olddos.exe

This is a self-extracting archive containing some of the files that were distributed with versions of Windows prior-to and including Windows 95, programs such as **qbasic**.

3.1.2. `NT Resource Kit`

http://www.microsoft.com/ntserver/nts/downloads/recommended/ntkit/default.asp

This is a subset of the tools found in the full Windows NT Resource Kit. Only available for Windows 2000/NT/XP users.

3.1.3. `CHOICE.EXE`

`CHOICE.EXE` is a batch command which allows the requisition of a choice from a basic set of choices presented to the user.

3.2. Batch Basics

3.2.1. Command Redirection and Pipelines

If you want to get help on a command in Windows, the usual way is to postfix the command with a space and then */?*.

By default a normal command accepts input from standard input, which we abbreviate to stdin, standard input is the command line in the form of arguments passed to the command. By default a normal command directs its output to standard output, which we abbreviate to stdout, standard output is usually the console display. For some commands this may be the desired action but other times we may wish to get our input for a command from somewhere other than stdin and direct our output to somewhere other than stdout. This is done by redirection:

- We use > to redirect stdout to a file, for instance, if we wanted to redirect a directory listing generated by the **ls** we could do the following:

  ```
  ls > file
  ```

- We use < to specify that we want the command immediately before the redirection symbol to get its input from the source specified immediately after the symbol, for instance, we could redirect the input to **grep**(which searches for strings within files) so that it comes from a file like this:

  ```
                      grep searchterm < file
  ```

- We use >> to append stdout to a file, for instance, if we wanted to append the current directory listing to the end of a file we could redirect the output from **dir** like so:

  ```
                      dir >> file
  ```

- We can redirect standard error (stderr) to a file by using **2>**, if we wanted to redirect the standard error from **commandA** to a file we would use:

  ```
  commmandA 2>
  ```

Pipelines are another form of redirection that are used to chain commands so that powerful composite commands can be constructed, the pipe symbol '|' takes the stdout from the command preceding it and redirects it to the command following it:

```
dir | more
```

The example above firsts requests a directory listing of the current directory using the **dir** command, the output from this is then piped to **more** which displays the results in a page by page basis, pausing after each screen full until the user presses a key.

3.2.2. Variables

3.2.2.1. Variables

New variables can be instantiated like this:

```
set name=value
```

The name must only be made up of alphabetic characters, numeric characters and underscores, it cannot begin with a numeric character. You cannot use keywords like *for* as variable names.

Variables are referenced like this: *%name%*, here is an example:

```
@echo off
msg1=Hello
msg2=There!
echo %msg1% %msg%
```

This would echo "Hello There!" to the console display, here is another example:

```
@echo off
set msg1=one
set msg2=%msg1% two
set msg3=%msg2% three
echo %msg3%
```

Would echo "one two three" to the screen. If you would like to echo the '%' character, escape it with another '%' like this '%%'. You can echo to another file to create a new batch file if you like:

```
@echo off
echo set msg=Hello World! > hello.bat
echo echo %%msg%% >> hello.bat
hello.bat
```

This would cause "set msg=Hello World!" to be echoed and redirected to the file `hello.bat`, "echo %msg%" is then echoed and redirected to the file `hello.bat` but this time appended to the end. Notice the double '%' in the program-listing so that the '%' character is echoed and not interpreted as being the opening of a variable name. The final line executes **hello.bat** causing it output "Hello World!".

3.2.2.2. Command Line Arguments

Command line arguments are treated as special variables within the batch script, the reason I am calling them variables is because they can be changed with the **shift** command. The command line arguments are enumerated in the following manner *%0*, *%1*, *%2*, *%3*, *%4*, *%5*, *%6*, *%7*, *%8* and *%9*. *%0* is special in that it corresponds to the name of the batch file itself. *%1* is the first argument, *%2* is the second argument and so on. To reference after the ninth argument you must use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have exhausted the arguments list.

3.2.3. Control Contructs

The flow of control within batch scripts is essentially controlled via the *if* construct.

3.2.3.1. If

This construct takes the following three generic form, the parts enclosed within '[' and ']' are optional:

```
IF [NOT] ERRORLEVEL number command
IF [NOT] string1==string2 command
IF [NOT] EXIST filename command
```

When a Windows command exits it exits with what is known as an *error level*, this indicates to anyone who wants to know the degree of success the command had in performing whatever task it was supposed to do, usually when a command executes without error it terminates with an exit status of zero. An exit status of some other value would indicate that some error had occurred, the details of which would be specific to the command. This is what the error-level *if* is for.

```
@echo off
if "%1"=="1" echo The first choice is nice
if "%1"=="2" echo The second choice is just as nice
if "%1"=="3" echo The third choice is excellent
if "%1"==""  echo I see you were wise enough not to choose, You Win!
```

What this example does is compare the first command line argument with the strings "1", "2" and "3" using == which compares two strings for equality, if any of them match it prints out the corresponding message. If no argument is provided it prints out the final case. If you want to include more than one command, enclose it within braces:

```
@echo off
if exist hello.bat (
   echo Removing hello.bat...
   del  hello.bat
   echo hello.bat removed!
)
```

Deletes hello.bat if it exists and prints acknowledgments

3.2.3.2. For

The syntax of the for command is:

```
FOR %%variable IN (set) DO command [command-parameters]
```

```
@echo off
if exist bigtxt.txt rename bigtxt.txt bigtxt
for %%f in (*.txt) do type %%f >> bigtxt
rename bigtxt bigtxt.txt
```

In this example first of all we check to see if the file `bigtxt.txt` exists, if it does we rename it to bigtxt so that it is not included within the scope of the next line. The second line says that for every *%%f* in *(*.txt)*, which is every `*.txt` file in the current directory, we should perform the command **type** on the files (which prints the file to stdout) and then append these to the file `bigtxt`. The third line simply renames the file `bigtxt` to `bigtxt.txt`.

A more useful option of the *FOR* construct is to use the option `/L` which turns in to like a numeric mode whereby the following syntax applies:

```
FOR /L %%variable IN (start,step,end) DO command [command-parameters]
```

Here is an example:

```
@echo off
FOR /L %%i IN (0 2 100) DO echo %%i
```

Which will simply, echo all the numbers in between zero and 100 inclusive, incrementing by 2 each time. If you want to include more than one statement, use brackets after the *DO* part:

```
@echo off
FOR /L %%i IN (0 2 100) DO (
   echo ***
   echo %%i
   echo ***
)
```

To echo some lines above and below each number.

3.2.3.3. Loops

The batch language is so brain dead one has to employ ad-hoccery to get anything useful done, as an example I will illustrate how to create a simple counter known as a bang counter:

```
rem A Bang Counter
@echo off
set n=%1
set i=
: loop
set i=%i%!
  echo Bang!
  if %i%==%n% goto end
goto loop
:end
```

This takes a string of bangs, e.g "!!!", as an argument and then echos "Bang!" for each bang in the string. The loop compares the string of bangs to the string *i* which is set up and loops while the two strings are not equal, on each loop iteration a bang is added to *i* hence the loop will iterate for the number of bangs in the argument string. One could use this to control the number of times some action was executed. If this is ran with no arguments the program loops forever because the bangs will never match.

3.3. Simple Examples

3.3.1. Creating Aliases

If you always use the same options to a command or would like to create an alias for a command with a certain set of options, why not use a batch file to do so, simply create the batch file and place it somewhere in your path. Make sure that you do not create a batch file with the same name as an actual command unless you really want to override the command if the batch file comes earlier on in the `PATH`. A few examples are shown below:

```
@echo off
REM dir.bat, maps dir to Unix ls
dir %1
```

```
@echo off
REM antlr.bat, creates shortcut for antlr java program
java antlr.Tool %1
```

```
@echo off
REM home.bat, changes to your home directory and lists contents
c:
cd \home
cls
dir
```

3.4. Example batch file

The example below illustrates some of the features of batch programming.

Example 1. Batch file - Concatenate and zip Example

```
@echo off ❶
:: A batch file that concatenates all the files of a given extension in ❷
:: the current directory to a specified file and then zips that file.
IF "%1"=="" GOTO USAGE ❸
IF "%2"=="" GOTO USAGE
FOR %%i IN (*.%1) DO type %%i >> %2 ❹
SHIFT ❺
pkzip %~n1.zip %1 ❻
GOTO END
:USAGE ❼
echo.
echo USAGE: %0 extension outputfile ❽
echo.
echo Concatenates all the files of the given extension in the current directory
echo to the specified outputfile and then compresses that file into a zip file.
echo.
:END
```

❶

```
@echo off
```

The command **echo** followed by the option `off` sets batch echoing to off, this means that the batch commands will not be echoed to standard output, that is, you will not see them on your screen. The line is prefixed by a "@" character which is used to specify that the line it is present on should not be echoed to standard output, hence is needed so that we do not echo the command sequence that turns echoing off.

❷

```
:: A batch file that concatenates all the files of a given extension in
:: the current directory to a specified file and then zips that file.
```

Comments in batch files are officially meant to begin with the prefix *REM* which stands for *REM*ark, this tells the command interpreter to ignore this line. It turns out, however, that it is quicker to use *::* in order to markup a comment. *::* is used because *:* indicates the start of a label, the next *:* tells the parser to ignore this label.

❸

```
IF "%1"=="" GOTO USAGE
IF "%2"=="" GOTO USAGE
```

In batch programming, command line parameters are referenced by using the *%n* notation where *n* indicates the numerical weight of the parameter on the command line. *%0* refers to the name of the batch file itself, *%1* the first parameter, *%2* the second and so on. You may only reference a maximum of 9 parameters in this way, if you need more you should process these first ones and then use the **shift** command to access the rest, the **shift**, shifts the parameters left by one so that *%2* becomes *%1* etc, hence the non-existent *%10* would be shifted into *%9* and you could access the parameter. There is no way to recover a parameter once it has been shifted out completely, i.e *%0* will disappear upon being **shift**ed.

The program listing illustrates the use of the *IF* clause to take an alternative course of action if no either or both of the required parameters is missing. *%1* and *%2* are compared to an empty string which would indicate their omission from the command line if true, hence if the *IF* clause evaluates to *TRUE* the command after is executed which is *GOTO USAGE* which indicates that program control should jump to the *USAGE* label.

❹

```
FOR %%i IN (*.%1) DO type %%i >> %2
```

This for construct enables us to implement iterative behaviour in our batch files. It says that for each *%%i* within the set **.%1* we should perform the command *type %%i >> %2*.

%%i is a for-loop variable, these begin with *%%* and end with a sequence of alpha characters. The for-loop says to assign the values in our set to this variable in turn, which allows us to reference each of the values in the set with the same variable in the for-loop body allowing us to perform a specific operation on each of the values in the set. The specific operation in this case is to redirect the output of **type** (which dumps the contents of a file to stdout) to a file, specifically *%2* which is our second command line parameter. The use of two > characters in series ensures that the contents of the file are appended to the end of the target file and do not overwrite the output file, which would be the case if only one > had been used.

❺

```
SHIFT
```

The shift command shifts all command line parameters one parameter to their left, see number 3 for more details about shift. The reason it has been used here is explained in the next callout.

❻

```
pkzip %~n1.zip %1
```

Pkzip is a shareware file compression program that is executed from the command line, for details on obtaining PkKzip see *Auxiliary files for extended batch programming,* it takes the command form:

```
PKZIP zipfile file(s)...
```

Where *zipfile* is the name of the zipfile you wish to create and *file(s)...* are the files you want to put in the zipfile. In order that the zipfile be given the same name as the second parameter (without the extension) passed to the batch file, the notation *%~n1* was used to specify that *%~n1* should take the value of the name of *%1* but not the extension. The notation *%2~n1* is not allowed so to reference just the filename of the second parameter passed to the batch file and not the extension, the command line parameters were first shifted to the left by one using **SHIFT**, as explained previously, the second parameter became the first parameter and it's filename could be referenced with *%~n1*. Pkzip was told to use the filename *%~n1.zip* as the zipfile and *%1* as the file to compress, which (owing to the **SHIFT**) was the second parameter passed to the batch file.

❼

```
:USAGE
```

Labels are specified by prefixing an alpha-character identifier with a *':'* character, labels are jumped to using the **GOTO**.

❽

```
echo.
echo USAGE: %0 extension outputfile
echo.
echo Concatenates all the files of the given extension in the current directory
echo to the specified outputfile and then compresses that file into a zip file.
echo.
```

By using **echo** text may be echoed to standard output so that the user may see it. **echo.** echos an empty line to standard output. This section of the program is only executed if one or both of the command line parameters are omitted. The program shows that normally execution will be caused to circumvent this section by means of *GOTO END* immediately prior its declaration.

3.5. BatchMaker example - Unix for Windows (sort of)

Note

This works on Windows 2000, I have tested this on Windows 98 and it does not work.

I was fed up of accidentally typing **ls** instead of **dir** at the DOS prompt to get a directory listing so I started making some batch files to map the DOS commands like, **move**, **dir**, **copy** etc. to their Unix equivalents. For example, here is the mapping for **copy** to `cp.bat`:

```
@echo off
if "%1"=="" GOTO USAGE
copy %1 %2 %3 %4 %5 %6 %7 %8 %9
GOTO END
:USAGE
copy /?
:END
```

Which just see's if there is a command line argument present, if there is not, it prints the usage instructions for the command, otherwise it executes the command with the maximum number of command line arguments (9) (just in case). Most of the mappings I wanted to do had this similar format, only varying in the number of required arguments, this is the main drawback. For instance **ls** (dir) takes no arguments so you would not want to print the usage instructions every time somebody tried to get a directory listing (the person would not even get their directory listing). What I could have done was use some kind of contrived kludge of a counter, a bang counter or something and let the person specify the number of command line arguments on the command line but I wasn't prepared to waste my time doing so when this behaviour suited the majority of commands I had to match. OK, I might take a look at it when I have nothing better to do... like never ;) This got tedious after a while so I thought it would be a good idea to make a command to do the mappings for me.

All I had to do to create a batch file that generated this template was make it echo that template to another file, changing the words "copy" to whatever the command is and do various other things, here it is:

```
@echo off
if "%1"=="" GOTO USAGE
if "%2"=="" GOTO USAGE

echo @echo off
FOR /L %%i IN (1 1 %2) DO (
  echo if "%%%%i"=="" GOTO USAGE
)

echo %1 %%1 %%2 %%3 %%4 %%5 %%6 %%7 %%8 %%9
if "%2"=="0" GOTO END
echo GOTO END
echo :USAGE
echo %1 /?
echo :END
GOTO END

:USAGE
echo maps a command to an alias
echo map %%1 %%2
echo %%1 is command to map
echo %%2 is the number of command line args the command *needs*
echo e.g map copy 2 > cp.bat (to map copy to cp.bat)
:END
```

When referring to echoing, this will occur to stdout but the command is used by redirecting the output to a file so what is being echoed will end up in another batch file. This means the things being echoed are batch commands. The batch file first checks that it's own command line arguments are present, if they are not it echos it's command usage. If the command line args *are* present, it first echoes:

```
@echo off
```

Because this will be present at the top of all of the batch files created. It then enters this loop:

```
FOR /L %%i IN (1 1 %2) DO (
  echo if "%%%%i"=="" GOTO USAGE
)
```

Which starts at *1* and iterates by *1*until *%%i* exceeds the number specified as the second command line argument, so if *1* is specified as the second argument the body of the loop will be executed once, echoing:

```
if "%1"=="" GOTO USAGE
```

"*%%%%i*" echos *%* followed by the value of *%%i*, the loop counter. So this loop creates the checks for the command line arguments that the command being mapped should have. If zero is specified as the second command line argument, the loop is never executed which is the desired action since a command with zero arguments needs no argument checking.

```
echo %%1 %%2 %%3 %%4 %%5 %%6 %%7 %%8 %%9
```

Echoes

```
command %1 %2 %3 %4 %5 %6 %7 %8 %9
```

So that the command will then be executed (in the generated file) with the maximum possible number of arguments, this is so that when the command is executed, any redirection and args etc. will be executed too, otherwise the command would not do anything.

```
if "%2"=="0" GOTO END
echo GOTO END
echo :USAGE
echo %1 /?
echo :END
GOTO END
```

The batch then checks if the number of required arguments for the command being mapped was zero, if it was then there is no need to echo the rest of the stuff because a command that needs zero arguments should generate a file like this:

```
@echo off
copy %1 %2 %3 %4 %5 %6 %7 %8 %9
```

If the required arguments for the command being mapped is not zero then the rest of the stuff is echoed, the rest of the stuff just ensures that if the mapped command is ran with not enough arguments, that command's usage instructions will be printed by calling the generic DOS help function of the command with the /? switch. The last line skips the usage instructions of `map.bat` so that the batch terminates.

An example usage will clarify, the contents of `mv.bat` after executing

`map move 2 > mv.bat`
are:

```
@echo off
if "%1"=="" GOTO USAGE
if "%2"=="" GOTO USAGE
move %1 %2 %3 %4 %5 %6 %7 %8 %9
GOTO END
:USAGE
move /?
:END
```

So of course you can redirect the batch file to somewhere in your path or whatever. Here is another quick batch file that generates some aliases and puts them in a directory of your choice:

```
@echo off
if "%1"=="" GOTO USAGE

call map copy 2 > %1\cp.bat
call map move 2 > %1\mv.bat
call map del  1 > %1\rm.bat
call map type 1 > %1\cat.bat
call map dir  0 > %1\ls.bat

GOTO END

:USAGE
echo Creates a set of Unix-named aliases for common DOS commands:
echo map %%1
echo where %%1 is the output directory, e.g c:\windows\command
:END
```

Notice that the keyword *call* is used to call the *map*, this is necessary so that after *map* has finished, program control will return to the next statement in this batch file. If call had not been used then only the first statement would have executed because the batch would have terminated after *map* had. Feel free, of course, to add any other commands you want to.