A Windows Batch File Programming Primer

v1.0.2 / 01 jun 14 / greg goebel / public domain

* The Microsoft Windows operating system provides a useful tool for customizing the operation of a PC computer, known as "batch file" programming. Batch files are an extension of the Windows "CMD.EXE" program. CMD.EXE is a type of "shell" or "command interpreter program": it interprets commands and executes them. CMD.EXE can execute commands interactively or it can execute the same commands stored in an ordinary text file as a batch file. Batch files are an antique technology by modern standards, batch files having been introduced with the original MS-DOS operating system in the 1980s, but they still remain useful.

This document provides a quick introduction to Windows batch file programming. It assumes a basic knowledge of how to use a Windows PC, including familiarity with directories, file systems, mouse operations, and the Windows Explorer file browser.

---

[1] UNDERSTANDING CMD.EXE

* Understanding how batch files work involves understanding how the CMD.EXE shell works. CMD.EXE is effectively a holdover from the MS-DOS operating system, in its original form being known as "COMMAND.COM"; in those days it was effectively the only way to run programs, navigate through directories, create or delete directories, delete or copy files, and so on. These days such tasks are usually done with the Windows Desktop and Windows Explorer.

To tinker with CMD.EXE, use Windows Explorer create a directory for test and experimenting, say named "Test", then find CMD.EXE and make a copy of it -- using the mouse "right click" button on the CMD.EXE icon to get a popup menu with a "Copy" entry near the bottom. Typically it's found in the "C:\Windows\System32" directory; if not, look in some other "System" directory under "C:\Windows". In any case, once copied, paste a shortcut to CMD.EXE in the "Test" directory by performing a "right click" on Windows Explorer and selecting "Paste Shortcut".

Clicking on the shortcut icon in the test directory brings up CMD.EXE, popping up a bland window with a black background and white text, displaying a "command prompt" -- which, incidentally, is used in Windows documentation in some places as an alternate name for CMD.EXE. The prompt by default will be:

```
C:\\Windows\\System32>
```

That's because we're using a shortcut and CMD.EXE actually still resides in that directory. Not a problem, however; it's straightforward to change the shortcut to use the test directory. At the top of Windows Explorer, there's a bar that lists the current directory in the form of, say:

```
Bugs > My Documents > Workspace > Test
```

Click on the bar and the listing in it changes to a proper directory name:

```
C:\\Users\\Bugs\\Documents\\Workspace\\Test
```

Copy the name using the mouse, then right-click on the CMD.EXE shortcut to get a menu. Select "Properties" from the menu, and when the "Properties" panel comes up, paste the directory name into the "Start in" field. Save the changes. Now when the CMD.EXE shortcut is clicked, it comes up in the "Test" directory with the prompt:

```
C:\\Users\\Bugs\\Documents\\Workspace\\Test>
```

* Now that we've got CMD.EXE working, the next question is: What do we do with it? Actually, CMD.EXE is really nothing more than an ancestral version of Windows Explorer, being exactly the way users made their way around a PC in the days of MS-DOS and before windowed graphical user interfaces. Like Windows Explorer, CMD.EXE is generally used to copy, rename, or delete files; create or delete directories; change from one directory to another; and so on. A list of basic commands includes:

```
CD      Displays the name of or changes the current directory.
CLS     Clears the screen.
DATE    Displays or sets the date.
DEL     Deletes one or more files.
DIR     Displays a list of files and subdirectories in a directory.
FIND    Searches for a text string in a file or files.
HELP    Provides Help information for Windows commands.
MD      Creates a directory.
MORE    Displays output one screen at a time.
MOVE    Moves files from one directory to another directory.
RD      Removes a directory.
REN     Renames a file or files.
SORT    Sorts input.
TYPE    Displays the contents of a text file.
```

To play with these commands, use Windows Explorer to copy a set of various kinds of files -- some text files, some graphics files, whatever -- to the "Test" directory. Run CMD.EXE and then enter "DIR" at the prompt; the result is a file listing of the form:

```
 Volume in drive C is HP
 Volume Serial Number is A234-B94A

 Directory of C:\\Users\\Bugs\\Documents\\Workspace\\Test

06/25/2010  12:47 PM    <DIR>          .
06/25/2010  12:47 PM    <DIR>          ..
06/20/2010  06:53 PM             9,431 aax.txt
02/15/2010  01:29 PM         2,743,479 analog.pdf
02/25/2010  08:18 PM            19,893 avpogo.txt
06/23/2010  12:43 PM             1,253 cmd.exe - Shortcut.lnk
05/30/2010  08:19 AM           142,470 test1.txt
05/04/2010  01:01 PM             2,700 test2.txt
               6 File(s)      2,919,226 bytes
               2 Dir(s)  272,897,269,760 bytes free
```

Of course, the listing will look slightly different in other PCs. Note that the "." directory listing just means "current" directory, that is "Test"; the ".." directory means the "parent" directory, that is "Workspace". If we wanted to save this listing in a file, named say "dirlist.txt", we could use the ">" character to perform "output redirection" as follows:

```
DIR > dirlist.txt
```

If we type in this command at the prompt and press ENTER, the listing is dumped to the file -- nothing shows up in the CMD.EXE window, it just displays the prompt again. Output redirection is a general sort of trick, it can be used with any command executed in CMD.EXE that generates an output.

If we wanted to look at the contents of text file, say "test1.txt", we could list it to the window with:

```
TYPE test1.txt
```

However, for a text file of any real length, the text just streams up the display to the end of the file, which isn't very useful. We can, however, using the "|" to create a "pipe" to MORE:

```
TYPE test1.txt | MORE
```

A "pipe" is a connection between the output of one command and the input of another -- redirecting output with ">" doesn't work with a command, it's only good for dumping to files and the like. Incidentally, we could also "perform input redirection" with "<" to send the file to TYPE as follows:

```
TYPE | MORE < test1.txt
```

This is not very sensible since TYPE can accept a filename as a parameter, but there are cases where input redirection does make sense, and so it's good to know what it is. If we want to clear the window and get rid of the stuff we listed, we can do that with:

```
CLS
```

Now let's play with creating and moving between directories. We can use "MD" or "MKDIR" to create two directories:

```
MD TestDir1
MD TestDir2
```

Just as an exercise, we can copy all the text files from the "Test" directory to "TestDir1" using:

```
COPY *.txt TestDir1
```

The "*" is a "wildcard character" that will match any filename, or at least in this case any filename that ends in ".txt". If we wanted to copy all the files, we'd use "*.*"; if we just wanted to copy text files that started with "h", we'd use "h*.txt". That done, we can change to "TestDir1" using:

```
CD TestDir1
```

Perform a "DIR" and then the files we copied to "TestDir1" will be listed. We can then change directories to "TestDir2" using:

```
CD ..\\TestDir2
```

Remember that ".." is the parent directory; if we just tried:

```
CD TestDir2
```

-- CMD.EXE would assume that "TestDir2" is a sub-directory of "TestDir1" and, since it isn't, would give an error message. Anyway, that done, we can change back to "TestDir1" with:

```
CD ..\\TestDir1
```

-- and then delete all the files in that directory with:

```
DEL *.*
```

This is a command that should be used with caution since it can potentially do a great deal of damage. Incidentally, to confirm the directory, just enter "CD" by itself, and it will give the current directory name. In any case, we can return to the "Test" directory with:

```
CD ..
```

-- and then kill off "TestDir1" using "RD" or "RMDIR":

```
RD TestDir1
```

Some commands have "switches", or flags of the form "/<someletter>", to alter their behavior. For example, "SORT" normally sorts text in alphabetic order -- "A" to "Z" -- but by using it with "SORT /R", it will sort in reverse alphabetic order -- "Z" to "A".

* Anyway, that gives a quick tour of what CMD.EXE can do and how it is used. It's not extremely useful knowledge in itself, except possibly for someone wanting to learn about how people used PCs back in the 1980s, but it gives useful background for creating batch files. Anyone interesting in learning about the command set can get a listing by entering "HELP" at the CMD.EXE prompt, with details of each command and its switches provided by entering "HELP <command>" -- for example, "HELP DIR" gives a detailed description of "DIR". Incidentally, those who don't like the colors or fonts of the CMD.EXE can alter them by performing a "right click" on the title bar of the CMD.EXE window, and then selecting "properties".

It should be noted that some of the commands in the set, like "CLS", are built into CMD.EXE, while others, like "FIND", are actually separate programs. Any Windows program can be invoked from CMD.EXE just by entering it at the prompt; for example, to run a text file in Windows Notepad, just enter:

```
notepad mytextfile.txt
```

-- and the Notepad window will pop up. In fact, Windows includes a number of programs for system and network administration that can be invoked from CMD.EXE, though discussion of these advanced tools is beyond the scope of this document.

BACK_TO_TOP

[2] BATCH FILE BASICS

* A Windows batch file is no more than a text file containing a list of commands for CMD.EXE and with a ".BAT" extension. A batch file can be written up in a text editor like Windows Notepad. Note that the file does have to be plain text; using a word processor that creates files containing formatting information is not likely to work very well and should be avoided. For example, if we write a list of commands in a file named:

```
TESTBATCH.BAT
```

-- and save it, then if we click on it with a mouse, a CMD.EXE shell will be brought up automatically and run it, sequencing through the list of commands. A batch file is effectively a Windows program in itself; it can be placed in a directory in the Windows "search path" -- that is, the list of directories in which Windows looks for programs to execute -- and then can be invoked from any directory in the system. C:\Windows is a good place to store batch files, though more advanced user can set them up in their own directory -- more on this below.

Batch files can also be set up in the Windows "Open With" menus available under Windows Explorer,

allowing them to be used with different types of files:

- "Right click" with the mouse on the appropriate file icon.
- Select "Open With" from the menu.
- A panel appears to allow selection of existing programs associated with the file type. There's "browse" button there that can be used to explore the file system and locate the desired batch file, for example in C:\Windows.
- Once the batch file has been added to the set of associations, then it can invoked by bringing up the right-click menu, selecting "Open With", and then selecting the batch file name from the list that pops up.

If a copy of the batch file or a shortcut to the batch file is left in a directory, then it can be invoked by simply performing a "click and drag" of the desired files onto the batch file icon. This is not usually particularly convenient, however; it may be better to invoke a CMD.EXE shell.

* In any case, if we create a text file named KILLALL.BAT that contains the command:

```
DEL *.txt
```

-- then when we click on the batch file, all the text files in the directory will be deleted. Of course we can add more commands to the batch file as we like.

BACK_TO_TOP

[3] INFORMATIVE BATCH FILES: ECHO, PAUSE, REM

* Normally when a batch file is run, each statement in the file is displayed in the CMD.EXE window as the file is executed. This is no great trouble for little batch files and can be useful for debugging, but it is a nuisance for bigger batch files. We can turn off this "echoing" of commands by simply including the command "ECHO OFF" at the beginning of the batch file. Unfortunately for the perfectionist, the command "ECHO OFF" is still echoed itself, but luckily it echoing can also be disabled for individual commands by preceding them with an "@" character:

```
@ECHO OFF
```

"ECHO ON" returns the batch file to normal echoing. Simply invoking "ECHO" without an argument gives the current echoing state:

```
ECHO is on
```

-- or:

```
ECHO is off
```

"ECHO" can also be used to display text from the batch file, being invoked as:

```
ECHO <string>
```

For example, if we create a batch file "TEST.BAT" containing:

```
@ECHO OFF
ECHO This is a test!
```

-- then when we execute "TEST.BAT", we get the string:

```
This is a test!
```

-- at the prompt. And nothing else; note the use of "@ECHO OFF" to disable batch file command echoing. Echoing text permits a batch file to let the user know what it's doing, generally a recommended practice.

However, there are a few catches. For example, suppose we want to "ECHO" a blank line to the output. We can't just execute "ECHO" without an argument, because that will produce:

```
ECHO is off
```

To "ECHO" what appears to be a blank line, all we have to do is "ECHO" a string consisting of a backspace or a period or some other harmless character:

```
ECHO .
```

Another trip-up is to try to echo a string of the form:

```
ECHO  Correct syntax is:  LS [/D | /F] [<file_specifier_argument>]
```

This will cause an error message, because in that string there are the characters "|", "<", and ">", which are the DOS characters for a pipe, input redirection, and output redirection respectively. These characters will completely baffle the "ECHO" statement, causing it to generate a "FILE NOT FOUND" error. It is possible to "escape" any such "special characters" by preceding them with a "caret" or "^" -- for example, "^|" or "^>" or "^<". It is also possible to enclose the string in double quotes ("") to give some protection:

```
ECHO "Correct syntax is:  LS [/D | /F] [<file_specifier_argument>]"
```

However, any time special characters become involved, things get tricky. They get *very* tricky when the string itself is supposed to include double quotes.

* One of the issues with using "ECHO" to follow what a batch file is doing is the fact that when a batch file is executed, a window pops up, the text is displayed, and then the window promptly disappears before the text can be read. To leave the window on the display, just put a "PAUSE" statement as the last command in the batch file. "PAUSE" stops batch file execution, with a prompt displayed to ask the user to press any key to continue; press any key, and the window then disappears. For debugging, as many "PAUSE" statements can be placed in a batch file as desired, allowing a user to step through the sections of a batch file.

* It is also possible to include comments using the "REM" (remark) command. If we start a comment line with "REM", CMD.EXE will ignore it -- "::" can be used in place of "REM" if desired. Remarks are useful for documenting batch files or for adding spacing to make their contents neater, both good ideas for batch files of any complexity. It's also useful to include a revision code or at least a "last modified" date, to prevent different versions of the same batch file from being confused.

BACK_TO_TOP

[4] BATCH FILE VARIABLES

* As defined so far, batch files simply execute a list of commands. If all we can do is execute the exact same actions each time we run the batch file, they're of limited use. For this reason, batch files support "command line arguments" to allow us to invoke file names or other text along with the batch file. The batch file can read these arguments and modify its operation accordingly. The arguments are

stored in "argument variables". These are batch-file keywords of the form "%1", "%2", and so on, up to "%9", and each contains the corresponding argument on the command line. For example, consider a batch file named "TEST.BAT" that contains:

```
DEL %1
DEL %2
DEL %3
```

If we invoke TEST.BAT as:

```
TEST FILE1 FILE2 FILE3
```

-- then the batch file would perform the actions:

```
DEL FILE1
DEL FILE2
DEL FILE3
```

If we added a "FILE4" to the list, that argument would be ignored. If we didn't specify "FILE3", we'd get an error message because "DEL %3" would evaluate to "DEL", which would complain that it needed an argument. Notice that there is also a "%0" argument variable, which gives the name of the batch file itself. In the case of the example above, "%0" would be "TEST.BAT".

* Along with the argument variables, user-defined variables can also be created in batch files with the "SET" command. For example, to create a user-defined variable named "UVAR" and load it with the string "hi_there", we invoke:

```
SET UVAR=hi_there
```

Of course, we can use any other reasonable name -- except names that match those of CMD.EXE commands or other "reserved words". Using a reserved word as a variable name can lead to error messages and subtler difficulties, which can be hard to track down if we're not aware that we're duplicating a reserved word. Anyway, now suppose we wanted to display the contents of UVAR. If we simply invoked:

```
ECHO UVAR
```

-- that wouldn't work, of course it would simply display the string "UVAR". We have to enclose the variable name in percent signs ("%") to allow a batch file to recognize it as a variable:

```
ECHO %UVAR%
```

This properly displays the contents of UVAR, that is "hi_there". We can clear the value of the environment variable just by setting it to nothing:

```
SET UVAR=
```

New string values can be easily tacked onto existing string values:

```
SET UVAR=hi_there
ECHO %UVAR%
SET UVAR=%UVAR%_everybody
ECHO %UVAR%
```

This batch file displays the strings "hi_there" and "hi_there_everybody".

* A batch file can also access "environment variables" available from the PC's Windows installation. Environment variables can be either "system variables", used directly by Windows, or "user

variables", defined by the user. For example, the "OS" system variable defines the name of the operating system being used. To inspect or alter environment variables:

- Right-click on the to-level "Computer" icon in Windows Explorer and select "Properties" from the menu.
- Click on "Advanced Systems Settings" in the Control Panel window.
- Click on "Environment Variables" in the window that pops up. This gives a panel with a list of the user variables at top and the system variables at bottom

It is not wise to tinker with system variables and usually there's no reason to do it. One exception is the "Path" system variable, which lists the directories that Windows searches to find programs to run. As mentioned above, we need to place batch files in a directory where Windows can find them to run, for example "C:\Windows", which is in the "Path". If this seems clumsy, we can set up a special directory for batch files, say "C:\Windows\Batch", and then include that name in the "Path", using a ";" as a separator character.

As far as user variables go, they can be defined for any purpose a user likes. For example, if we have a work directory of the form:

```
C:\\Users\\Bugs\\Documents\\Workspace\\Test
```

-- and we write batch files that make use of it, it is *not* wise to actually use the full directory name in the batch files -- since if the system changes and the directory name is affected, then we'll have to change all the batch files accordingly. It is better to set up the directory name as a user variable, say "WKDIR", and then use the variable in the batch files instead:

```
CD %WKDIR%
```

Incidentally, "relative" directory names generally work fine and are no problem:

```
CD ..\\TestDir2
```

Notice that if we "SET" environment variables in a batch file, we can check their values simply by invoking the "SET" command without any arguments. The variables accessible to the batch file and their values will be listed.

Note also that "SET" can be used to read user input into a variable. Back in MS-DOS days, getting user input beyond "press any key to continue" was tricky, but under the modern CMD.EXE, a "/P" switch can be used to tell "SET" to get user input. For example, this little batch file asks the user to input a color:

```
@ECHO OFF
SET /P CVAR=What is your favorite color?
CLS
ECHO Favorite color is:  %CVAR%
PAUSE
```

Another nice feature added to modern batch files is the ability to "SET" numeric variables using the "/A" switch. Traditionally, a variable just stored a string of text. Perform:

```
SET NVAR=100
```

-- and then execute:

```
SET NVAR = %NVAR% + 23
```

```
ECHO %NVAR%
```

-- we get, duh:

```
100 + 23
```

Change this to:

```
SET /A NVAR=100
SET /A NVAR = %NVAR% + 23
ECHO %NVAR%
```

-- and we get:

```
123
```

All the standard arithmetic operations -- "+ - * /" -- work, as do parenthesis -- "(2/5)+32". Logical operations are supported as well, but in the interests of brevity they are not discussed here. There's also "assignment operators" of the form "+= -= *= /=" that permit replacing, say:

```
SET /A NVAR = %NVAR% + 23
```

-- with:

```
SET /A NVAR += 23
```

Works the same either way. Numeric variables are something of a patchup fix, they don't really make batch files a very good tool for number-crunching, being more adequate to perform counts, tallies, or simple conversions. They're worth having; back in the old MS-DOS days, the lack of any ability to handle numbers in batch files could be troublesome.

* Incidentally, the discussion of argument variables defined nine such variables, "%1" through "%9", which contain the first nine command-line arguments. But what if we want to get at more than nine command-line arguments?

The SHIFT command allows us to do this. When the batch file executes SHIFT, the value in %2 is moved to %1; the value of %3 is moved to %2; the value of %4 is moved to %3; and so on. We can use looping constructs, discussed below, to access as many command-line arguments as we like.

* Since we are not likely to know exactly what a variable contains, we can get into trouble if we use ECHO to, say, write a password stored in a variable into a file as follows:

```
ECHO %PASSWD% > MYFILE.TXT
```

That password may not actually work. Why not? Because it ends up with a space character on the end, and that's because we left a space before the ">" operator in the batch file. Getting rid of the space is very easy:

```
ECHO %PASSWD%> MYFILE.TXT
```

It's a little klunky-looking, but it does work.

BACK_TO_TOP

[5] DECISION-MAKING: IF, GOTO

* Batch files can make simple decisions using the "IF" statement, with the decisions based on a set of

simple criteria:

- Whether a variable matches or does not match a certain string.
- Whether a file exists or not.
- Whether a variable exists or not.
- Whether an "error code" returned by a program called by a batch file is above or below a certain value.

The most important test is for checking to see if an argument variable matches a certain string, or, by using a secondary "NOT" keyword, to see if it *doesn't* match a certain string. If the conditional test succeeds, the IF statement will execute a command. This command is very often a "GOTO" statement, though it can be any other legal batch file command.

So what is the "GOTO" command? It is a statement that can be used, on its own or as part of an "IF" statement, to tell CMD.EXE to skip to a "line label" (a string preceded by a ":") and start execution there. For example, consider this example "TEST.BAT" batch file:

```
@ECHO OFF
IF "%1"=="DONOTHING" GOTO Skipit
IF "%1"=="" ECHO No arguments!
IF "%1"=="COYOTE" ECHO Argument is COYOTE!
IF NOT "%1"=="COYOTE" ECHO Argument isn't COYOTE!
:Skipit
ECHO Done!
PAUSE
```

A few test runs yield the results:

```
_____________________________________________

run               result
_____________________________________________

TEST              No arguments!
                  Done!

TEST DONOTHING    Done!

TEST hello        Argument isn't COYOTE!
                  Done!

TEST coyote       Argument isn't COYOTE!
                  Done!

TEST COYOTE       Argument is COYOTE!
                  Done!

_____________________________________________
```

Note that the test is "case-sensitive" -- that is, the argument "COYOTE" successfully matches the comparison string "COYOTE", but the argument "coyote" does not. For a "case-insensitive" comparison, use the "/I" switch with the "IF" test:

```
IF /I "%1"=="COYOTE" ECHO Argument is COYOTE!
```

Note also the use of double-quotes to define parameters:

```
IF "%1"=="DONOTHING" GOTO Skipit
```

This is because the IF statement *must* have an argument on the left side of the comparison, or we'll get an error message. The statement:

```
IF %1==DONOTHING GOTO Skipit
```

-- might *seem* legal, but it ceases to be so if there isn't any argument for "%1", making the statement appear as:

```
IF ==DONOTHING GOTO Skipit
```

If we embed both the argument variable and test string in double-quotes, we'll always get a legal comparison. Actually, we could use almost *any*characters beside double-quotes, for example:

```
IF [%1]==[DONOTHING] GOTO Skipit
```

-- but double-quotes are a little less confusing. Incidentally, there's a "special" line label, ":EOF", that isn't actually a line label, it's just a trick to tell the batch file to "stop". That is, if the following condition is met, the batch file stops immediately:

```
IF "%1"=="STOPIT" ECHO Stopped! & GOTO :EOF
```

The "&" is another trick, used to group commands together; in this case it groups the "ECHO" and the "GOTO", allowing the batch file to announce that it has "Stopped!" and then quit.

* The "IF" statement can also be used with an "ELSE" clause as follows:

```
IF "%1"=="COYOTE" (ECHO Yes) ELSE (ECHO No)
```

Notice the use of parenthesis to "disambiguate" the "ELSE" clause. If we didn't use the parenthesis:

```
IF "%1"=="COYOTE" ECHO Yes ELSE ECHO No
```

-- on a match, this statement would output:

```
Yes ELSE ECHO No
```

-- and the "ELSE" clause would never work. We can chain IF statements together, which results in an "AND" of the various test conditions -- that is, the batch file does nothing unless *all* the test conditions are met:

```
IF "%1"=="ONE" IF "%2"=="TWO" GOTO Match
```

However, it is generally more practical to use multiple "IF" statements to perform such tests. For example, here's how we would perform a test on an AND of conditions:

```
IF NOT "%1"=="TEST1" GOTO Fail
IF NOT "%2"=="TEST2" GOTO Fail
IF NOT "%3"=="TEST3" GOTO Fail
GOTO Success
```

The "IF" statements form a "gauntlet" through which command processing must pass if it is to perform the statements desired, or in this case, jump to the commands labeled with "Success". If any of the parameters don't match, command processing goes to "Fail". We can perform an "OR" of conditions by reversing this logic:

```
IF "%1"=="A" GOTO Success
IF "%1"=="B" GOTO Success
```

```
IF "%1"=="C" GOTO Success
GOTO Fail
```

There's actually a more elegant way of performing such sets of tests, as discussed below. However, it should be noted that complicated decision-making in batch files is a difficult and error-prone process, and once we start placing a lot of "IF" conditions in our batch files we swiftly find yourself finding that we are using a poor tool for the job. There are better tools for complex jobs, mentioned later.

* The other forms of the IF statement are not used as often. We can perform a test for the existence of a file using the "EXIST" keyword:

```
IF EXIST %1 GOTO Goodfile
```

We can use the "NOT" keyword to reverse the sense of the comparison:

```
IF NOT EXIST %1 GOTO Goodfile
```

This command can be used for a level of error-checking. However, it cannot perform a check on a directory name; the argument has to be a file.

We can check to see if a variable has been defined:

```
SET TVAR=Whatever
IF DEFINED TVAR ECHO "Variable defined!"
```

Of course we can use "NOT DEFINED" to check to see if the variable hasn't been defined.

We can also perform error testing using the "IF ERRORLEVEL" form, which tests to see if the last program run by a batch file returns an "error number" equal to or greater than the specified error level. For example:

```
IF ERRORLEVEL 32 GOTO Error
```

The "NOT" prefix can be used to reverse the sense of the comparison.

By the way, with the introduction of numeric variables, the test conditions were enhanced from merely "==" to a set of numeric comparison tests:

```
EQU        actually same as "=="
NEQ        actually same as "NOT ... =="
LSS        less than
LEQ        less than or equal to
GTR        greater than
GEQ        greater than or equal to
```

For example, the following batch file performs a series of tests on a command-line argument:

```
SET /A NVAR=%1
ECHO number is: %NVAR%
IF %NVAR% EQU 0 ECHO equal to zero
IF %NVAR% NEQ 0 ECHO not equal to zero
IF %NVAR% LSS 0 ECHO less than to zero
IF %NVAR% LEQ 0 ECHO less than or equal to zero
IF %NVAR% GTR 0 ECHO greater than to zero
IF %NVAR% GEQ 0 ECHO greater than or equal to zero
```

They can actually be used for string variable comparisons as well, but in that case they work on the basis of alphabetical order.

BACK_TO_TOP

[6] LOOPING: FOR

* The "FOR" construct offers looping capabilities for batch files. In specific, it allows sequencing through a list of parameters. For example, the "TYPE" command doesn't understand wildcards, and if we hand it a wildcard argument, we'll get an error message. But we can appropriately handle all file arguments using the following commands:

```
@ECHO OFF
:Test
IF "%1"=="" GOTO Done
FOR %%F IN (%1) DO TYPE %%F
SHIFT
GOTO Test
:Done
```

The "FOR" statement assigns the appropriate expanded values of %1 sequentially to the variable %%F -- notice the use of doubled percent signs -- and this %%F variable is used as an argument for "TYPE". That is, if we had the files:

```
FILE_1
FILE_2
FILE_3
```

-- then if %1 had the value "FILE_?", %%F would be sequentially given the values of all three file names, and the three files would be "TYPEd" in sequence.

We can give "FOR" a list. For example, suppose we want to test to see if a particular command-line argument had one of a number of values (an "OR" condition). We could use the following FOR loop to test:

```
FOR %%F IN (A B C D E a b c d e) DO IF "%1"=="%%F" GOTO Match
GOTO Nomatch
```

This is much cleaner than performing multiple "IF" tests.

The modern version of "FOR" can do a lot more than it could in MS-DOS days. For example, we can tell "FOR" to count up over a range of values by a specified increment using the "/L" switch. For example:

```
FOR /L %%X IN (0,1,50) DO ECHO %%X
```

-- counts from 0 to 50, incrementing by 1. Change the range specification to "(0,2,50)" and it counts incrementing by 2. Another useful trick is the "/R" switch, which not only cycles the FOR through the current directory, but through all subordinate directories as well. For example, to copy all text files in the current directory to a temporary directory named "C:\tmp", use:

```
FOR /R %%F IN (*.txt) DO COPY %%F C:\\tmp
```

The modern version of "FOR" also includes a "/F" switch that allows it to scan through the text in a list of files, find text matching a string, and sort out various fields of text in a file. It is complicated to explain, and is not discussed further here.

BACK_TO_TOP

[7] EXECUTING OTHER BATCH FILES: CALL

* We can have a batch file execute another batch file. Consider the following example:

```
@ECHO OFF
ECHO Now running other batch file!
TEST.BAT
ECHO Done running other batch file!
```

This performs the first "ECHO" statement, and then executes the "TEST.BAT" batch file (to do whatever it does). However, the second "ECHO" statement is *never* executed -- why not?

The reason is that when we tell CMD.EXE to execute the statements in "TEST.BAT", it does so to the end of the "TEST.BAT" batch file, and at the end, CMD.EXE stops. It's reached the end of the batch file, and it has no way of knowing that it's supposed to return to the "main" batch file and go on executing statements.

That's where the "CALL" statement comes in. If we execute the other batch file using "CALL":

```
@ECHO OFF
ECHO Now running other batch file!
CALL TEST.BAT
ECHO Done running other batch file!
```

-- CMD.EXE is then smart enough to come back to the "main" batch file and continue executing commands there.

"CALL" is very important relative to the "FOR" command. Since "FOR" can only execute a single batch file command -- or maybe two or three linked by "&" -- if we want to execute multiple commands we'll need to put them in a separate batch file and use "CALL" to run that file:

```
FOR %%F IN (%1) DO CALL BATCH2.BAT
```

This same trick could also be used in an "IF" statement as an alternative to "GOTO", though proliferating batch files isn't necessarily a clean solution.

* Incidentally, one of the treacherous things about performing a "CALL" from a "parent" batch file on a "child" batch file is that the parent and child share the same "context". What that means is that if the child batch file, say, changes directories, it will also change directories for the parent batch file. This is not necessarily true for some other types of shell programs, so it can be confusing. If we want the parent to stay in the directory it was originally running in, one way is to store the directory name and then change back to it after running the child batch file:

```
SET MYDIR=%CD%
CD C:\Tmp\SomeDir
CALL BATCH2.BAT
CD %MYDIR%
```

However, the same lack of context can trip up this trick as well, since the parent and child batch files also share variable names -- if the child batch file has a variable named "MYDIR" and changes it, then the original directory name will be lost. Actually, the batch file command set includes specific provisions for dealing with this situation, in the form of the "PUSHD" and "POPD" commands. "PUSHD" changes the directory but also automatically records the current directory, while "POPD" restores the current directory:

```
PUSHD C:\Tmp\SomeDir
CALL BATCH2.BAT
```

```
POPD
```

"PUSHD" can be used multiple times to change directories, leaving behind a trail of old directories on a "stack", with "POPD" then executed repeatedly to trace back through the directory changes in reverse order.

BACK_TO_TOP

[8] BATCH FILES IN PRACTICE

* Now that we have a set of tools for building batch files, let's consider what they can do and what they can't. First, let's see what they *can't* do.

There is a temptation when first using batch files to try to use them for general-purpose programming tasks. They are wholly unsuited to such jobs, they are really only intended for file handling and to sequence the execution of other programs. Even at that, above a certain level of complexity batch files are hard to design, debug, and maintain. There are also a number of maddening quirks, some of which have already been mentioned. For another, suppose we have a batch file named "TEST.BAT" that generates a stream of output, and we want to pipe it through "MORE":

```
TEST.BAT | MORE
```

That won't work. When CMD.EXE is executing the batch file's commands, it knows nothing about either pipes or I/O redirection outside the file, though they work inside the batch file. We can get this to work by specifying a second CMD.EXE along with the "/C" option on the command line to execute the batch file:

```
CMD.EXE /C TEST | MORE
```

This "farms out" execution of the batch file commands to the second CMD.EXE, so the topmost CMD.EXE, the one accepting our command input, can simply accept it as part of the chain of commands.

* Batch files are suited to the execution of *system* functions, and even at that they're clumsy if the task requires much intelligence. For example one useful trick with a Windows batch file is a "mini-backup" facility, building a batch file that performs a "COPY" to, say, a pocket flash drive plugged into a PC; the batch file can be set up in the "right click" menu to be invoked on a file so it can be saved.

Batch files are also handy for automating laborious one-shot system management tasks. Suppose we want to copy one file, a template, to a long list of other files. Doing a copy and rename from Windows Explorer would be very time-consuming. Anybody handy with a text editor, however, could much more easily write a batch file to do the job, performing copy-pastes to build up a list and then setting up the target file names.

While batch files can be tortured into doing other tasks, they aren't really a good tool for them, and a general-purpose programming language would be a much better tool. The effort is better invested in learning the Python programming language -- available for free, with plenty of documentation available, capable of doing anything a batch file can and far more besides.

BACK_TO_TOP

[9] COMMENTS, SOURCES, & REVISION HISTORY

* This document was originally written during the 1990s for DOS-based programming, but when I set up my website a decade later it seemed too archaic to bother with. In 2010 I got to using batch files on Windows 7 and found them to be a pretty effective tool, within their limitations. In any case, I dusted off the old text, cleaned it up, updated it, and posted it to the website.

* Revision history:

```
v1.0.0 / 01 aug 10
v1.0.1 / 01 jul 12 / Added "\\R" flag for FOR loops.
                     Comments about unwanted extra spaces from ECHO.
v1.0.2 / 01 jun 14 / Review & polish.
```

BACK_TO_TOP