MS-DOS BATCH SCRIPT GUIDE

SCRIPT LANGUAGE, SYNTAX AND EXAMPLES

COVERS SCRIPT WRITING IN MS-DOS 5, MS-DOS 6.22, WINDOWS 95,
WINDOWS 98 AND (BASE-CONCEPTS FOR) WINDOWS NT 4.0

VERSION 1.1

TABLE OF CONTENTS

VARIABLES IN SCRIPT

Intrinsic Variables

Every batch file has a special environmental variable assigned upon startup. This variable is accessed internally through %0, which returns the full path and name of itself.

This variable is useful because DOS batch files are not aware of where they are. With this variable, you can have your batch file dynamically determine it's own location, and pass the path off to another script or look for other files within the same directory or sub-directories by using a special syntax: `%0\..\`

Using this notation, your script file can call another script file in the same or deeper location, without the environment needing to have the initial location in the PATH.

SYNTAX:	EXAMPLE:
`%0`	Returns: `C:\MYPATH\MYFILE.BAT`
`%0\..\NextFile.bat`	Yields: `C:\MYPATH\NEXTFILE.BAT`

Command Line Variables

Command line arguments are the strings following the program name in the command that invokes the program. In batch files they are read- only and are referred to by an escaped digit ('%' is the escape character in batch files). %0 through %9 are readily available, and any beyond %9 can be read by

SHIFTing them down into the single digit range. For each SHIFT, %0 disappears, and each argument takes on the value of the one to its right on the command line. Once shifted out of the %0 - %9 range, an argument cannot be recovered. %0 always exists (initially) and is the name of the batch file as given in the command. If given in short form, it is just the name; if given in long form, the path that was used precedes the name.

Command line arguments can be used to pass any string that doesn't contain a delimiter into the batch file. When a delimiter is encountered, the argument is split into two arguments, and all information about the nature of the delimiter is lost (you have no way of knowing if it was a space, a a tab, a comma, a semicolon, or an equals sign) - quotes don't help: the opening quote mark is part of the earliest argument and the closing quote mark is part of the last one in the string:

SYNTAX:	EXAMPLE:
Command line or within other program: FILENAME.EXT VAR1 VAR2 VAR3 Access from within FILENAME: %0 %1 %2 %3 %4	C:\TOUPPER.BAT killer

Environmental Variables

Environment variables are strings associated with a variable's name - in fact, a string associated with a string. Nearly all characters (including delimiters, even spaces) except the '=' sign are permitted in both the name and value. These variables reside in the current environment area of memory, and changes to the environment are accessible to child programs, but not to parent programs. One string is set equal to another by means of the SET command, and the value is read by placing the name between a pair of '%' characters.

One common problem with these variables is due to the inclusion of unwanted spaces. This is a byproduct of the ability to include spaces anywhere in the strings on both sides of the '=' sign - the name ends at the '=' sign and the value begins immediately after it. The usual remedy for this sort of thing is the use of quoted strings, but quoted strings exist in only a few places in batch file syntax (notably in remarks, where they are meaningless, and in strings to be ECHOed) - everywhere else, the quote marks are simply additional characters. These are the only kinds of variables to which the user can assign a constant value or have carry over from batch file to other programs. They are frequently used inside batch files as scratch pad memory.

Names of environment variables can contain most characters except the equals sign and lower case letters, but can't be counted on to work right unless they begin with a letter character. I'm not certain of the limits to the truth of that statement, but I had to give up using numbers because they didn't work right. Any environment set with the SET command will have its name in all upper case and any reference to the name with the %name% syntax will reference only the upper case name. Normally this is more of a feature than a wart - you don't have to pay attention to the case of the name, just of the value - but Windows sets a variable that gives the Windows directory path, and it has a lower case name: "windir", which makes it inaccessible from batch files. This fragment sets "WINDIR" to the contents of "windir":

```
set | find "windir" > }{.bat
echo set windir=%%1> windir.bat
call }{
del }{.bat
del windir.bat
```

To set a local variable: `SET [BatchVariable]=[Contents]`

To use the variable, place variable identifier % around the BatchVariable name, as in this example to set an environment path, using the above mentioned BatchVariable:

```
PATH D:\;%BatchVariable%
```

To delete the variable from memory, you re-assign the variable to an implied null, using a similar syntax:

```
SET BatchVariable=
```

SYNTAX:	EXAMPLE:

`SET VARIABLE=[String]`
`[Command]  %VARIABLE%`
`SET VARIABLE=`
	`SET MYNAME=PAUL`
	`ECHO My Name is %MYNAME%`
	`SET MYNAME=`

FOR Variables

FOR variables are given a temporary value of each element specified or given in the set part of the "for item in set do" statement. Inside batch files they are always in the form of a double '%' followed by a letter:

```
for %%a in (1 2 3 4) do echo %%a

for %%a in (*.*) do dir %%a
```

These variable have no existence outside the FOR statement.

SYNTAX:	EXAMPLE:

FOR %%[var] IN (set) DO [command]	FOR %%A IN (1 2 3 4) DO ECHO %%A

ERRORLEVEL

ERRORLEVEL is the exit code of the last executable program to be invoked, whether from the batch file, or before the batch file was invoked. The ERRORLEVEL remains set to whatever it is until another executable exits. DOS clears this to zero if the executable does not return any exit code. The only thing we can do with ERRORLEVEL is to compare it with a number, and then only with an "equals or is greater than" `(if errorlevel => x )` test, because the IF statement will return true for any value equal to or greater than the test value. This latter "feature" causes no end of trouble for unaware batch programmers. There are three ways to deal with it:

Testing Equal or Greater than, which is written in Inverse Order:

```
if errorlevel 255 goto x255
if errorlevel 245 goto x254
. . .
if errorlevel 2 goto x2
if errorlevel 1 goto x1
```

Test for ERRORLEVEL is less than by using NOT:

```
if not errorlevel 1 goto xok
:xfail
. . .

:xok
```

Which in essence performs a < test; and
Isolate individual values or ranges with a double test

```
if errorlevel 1 if not errorlevel 2 goto x1
if errorlevel 4 if not errorlevel 7 goto x4
if errorlevel 1 goto xelse
:x0
. . .
```

```
:x1
. . .

:x4
. . .

:xelse
. . .

etc.
```

Note that since ERRORLEVEL is of type "byte", its range is 0 through 255. Note also that there is no '=' in the syntax (it would be incorrect if used because the test is not one of equality).

ERRORLEVEL 0 indicates that the program terminated successfully, assuming that the program actually returned an exit code. However, it must be kept in mind that "success" was defined by the author of the executable and it's meaning may not be quite what the programmer of the batch file would expect.

For example:

```
XCOPY *.ext c:\temp /a
```

returns 0 if any files matching *.ext exist in the default directory, whether it actually copies any or not (it wouldn't, unless at least one had its archive attribute bit set). In the example case, the user might expect that ERRORLEVEL 0 would indicate that something had been copied and that ERRORLEVEL 1 would indicate some kind of failure to copy files, and a careful reading of the documentation for XCOPY implies that this is the case - nevertheless, reality is that there is no ERRORLEVEL that indicates that some files matching the pattern were found but none were copied because none satisfied the auxiliary test (the /a switch). DOS is like that - the wise batch file programmer is careful to test each element of the program's action separately.

INNER-SCRIPT SYNTAX

Labels

Labels normally mark the beginning of a code block that is the target of a GOTO instruction, but they can also be used for comments. Labels begin with ':' and contain up to eight characters - anything beyond the first eight will be ignored, both in the label and in the GOTO command. Not all characters are permissible in labels: those that COMMAND considers special (delimiters and markers), redirection characters, and wildcards - specifically , , , comma, semicolon, colon, double quote, '<', '>', '|', slash, backslash, '*', '?', '@', '%', '=', and period. In addition '[' and ']' are forbidden, at least in DOS 6+. The upper characters (above 127) appear to be usable, but the low characters (below 33) appear unusable. Labels are not case sensitive. Avoid using spaces in the label name, although if the space is replaced with character 255 (usually rendered as a blank), the label can be visually the same as with a space, but is handled as though character 255 is a regular character. Syntax Example:

SYNTAX:	EXAMPLE:

`:[Label]`	`[command]`
	`[command]`
	`GOTO MYLABEL`
	`[command]`
	
	`:MYLABEL`
	`[command]`

INTRINSIC COMMANDS

@

Placing the '@' symbol at the beginning of the line suppresses the line from being display when the command interpreter executes the command. The '@' symbol does not suppress display of the line's results.

SYNTAX:	EXAMPLE:
`@[command]`	`@ECHO OFF`

<, > and >> (Redirection)

Redirection is an operator, not a command. Redirection tells COMMAND.COM to connect the input or output of programs to either files (redirection) or other programs (pipes). This works only if the program uses the standard input and output streams for its console I/O. In general, programs with fancy screens don't and those with crude command line interfaces do. Most DOS utilities do. Nearly all filters do.

It is useful to know that when reading a line that uses pipes and redirectors, the line is read right to left if pipes are used prior to the redirection, then left to right after the redirection symbol. Graphical example:

```
echo. | date | find "Current" >> foo.log
<----r---e---a---d---------- then -r-e-a-d-->
```

English read: "Find/output the line starting with the word 'current' when running the Date program, using the <enter> generated by the echo command to answer the programs question, and display the find results as an addition to a file called foo.log."

<

The < is the input redirector, and connects a file (or device) to the input stream of a program so that you can put responses to prompts in a file and run the program automatically. Input redirection is seldom used these days because few programs use the standard input stream (STDIN) and few users understand the benefits.

> and >>

Output redirection is more generally used - it is useful for putting directory listings into files and for creating and filling files from inside batch programs, using the ECHO command and redirection to change the destination of the ECHO to a file instead of the console. > causes the file to be created or truncated - multiple ECHOs to the same file will leave the file with just the last one in it while >> causes whatever is redirected to be appended to the end of the file. >> is useful for building batch files from within batch files: the file is created with a > redirection and the remaining ECHO lines use >> to add the lines to the end of the file.

```
echo. | date | find "Current" > foo.log
echo. | time | find "Current" >> foo.log
echo. >> foo.log
```

The flow is to create a log file, by redirecting the output of the date search to a file, then add to that file with the output of the time search, and finally adding a blank line to the end of the log file. The resulting file contains:

```
Current date is Wed 03-18-1998
Current time is 10:56:25.31a
```

The same effect as above can be had, though more slowly, by using input redirection for the blank line, and redirecting it to the input of the DATE and TIME commands and TYPEing it to the target file (where response.fil initially only contains a hard return):

```
date < response.fil | date | find "Current" >> foo.log
time < response.fil | date | find "Current" >> foo.log
type response.fil >> foo.log
```

Note that output redirection and pipes work only with the standard output stream (STDOUT) and the standard input stream (STDIN).

The results is a log file that contains:

```
Current date is Wed 03-18-1998
Enter new date (mm-dd-yy): Current date is Wed 03-18-1998
```

```
Current date is Wed 03-18-1998
Enter new date (mm-dd-yy): Current time is 11:20:51.43a
```

There are many gotchas and subtleties to redirection and pipes: in complex commands only the output of the first command is redirected, and that is often something like if exist or FOR that doesn't even have an output in the usual sense.

SYNTAX:	EXAMPLE:

`[command] < [file]`
`[command] > [file]`
`[command] >> [file]`
	`FORMAT A: /U /V:TEST <`
	`C:\ANSWER.TXT`
	`ECHO LOG FILE NEW >`
	`C:\NEWLOG.LOG`
	`ECHO SECOND LINE >>`
	`C:\NEWLOG.LOG`

| (Pipes)

Pipes use the | symbol to connect the output of one program or command to the input of another. Information about pipes is documented under redirectors. Please read that section for additional information.

The use of the pipe command is to send the output of one action to the input of another.

In the example (which is read right to left):

```
ECHO. | DATE
```

The date program is started, and uses the echo's blank line (in essence a hard return or <enter>) to answer the question if asks (enter new date).

SYNTAX:	EXAMPLE:
`[command] | [command]`	`ECHO. | DATE`

FOR

The FOR command has many uses. It appears that the original intent appears to be to return file names in a list or matching a pattern one at a time, and it is still very useful for that sort of work, as shown in the below example:

```
for %%a in (*.bat) do if not exist .\bak\%%a copy %%a .\bak
```

This example makes sure that there is a backup copy of each of the batch files in the default directory, but copies only the ones that don't already have backups. It doesn't check to see if they are the same, but the backups may well be intended to be the previous versions. Note that the variable is a single letter preceded by double '%' signs. One of these is stripped off during command processing, so the real name is "%a", which is what you use when using this syntax from the command line. (It is not essential that the variable name use a letter, but numbers are forbidden and the convention is to use letters that mean something or are in alphabetic order (for multiple uses of FOR in the same file). Use of multiple letters for the variable name can result in syntax errors and halt process. FOR also has other uses, for example, to process a list of items:

```
for %%a in (*.bat *.exe, *.com) do if not exist .\bak%%a copy
%%a .\bak
```

does what the first example does, except for all three of those extensions.

The list (set) need not have anything to do with files, as the following digit counter illustrates:

```
FOR %%c in (0 1 2 3 4 5 6 7 8 9) do echo Counter at %%c >>
e:\system\desktop\results.txt
```

The end result of this line is a loop counter that repeats 10 times, outputting it's text to a continuously written file that records the counter.

FOR loops can not be nested in the traditional sense. Doing so results in a FOR syntax error.

There is also the undocumented syntax that places a '/' in front of a string. This syntax has the curious property of calling the DO clause twice - once with the character following the '/', and the second time with the rest of the string. This is handy for stripping leading characters and for reformatting strings.

Unfortunately, I can't figure out exactly how it works, so there is no sample code provided. If you can figure it out, let me know. The below code is supposed to remove the first 4 characters and truncate the "MAC" address to a DOS name, but when run in a Windows command.com environment, no truncating took place. It may be that this syntax only works in a DOS 6.22 environment.

```
@echo off
%1 %2 %3 %4
goto pass0
:pass1
for %%b in (/%SHRTNM%) do call %0 goto pass2 %%b
goto end
:pass2
goto pass2%pxt%
```

```
:pass2a
set pxt=b
goto end
:pass2b
set pxt=a
set shrtnm=%3
goto end
:pass0
set pxt=a
set shrtnm=C08U886AB1266
for %%a in (1 2 3 4) do call %0 goto pass1
set pxt=
:end
echo %shrtnm%
```

SYNTAX:	EXAMPLE:

`FOR %%[var] IN ([set]) DO [command]`	`FOR %%A IN (0 1 2) DO ECHO COUNT %%A`
	
	`Refer to the various examples in text for further usage.`

IF

The IF command is primarily set for variable string comparisons, using the dual "=" sign. The strings can be command line arguments, environment variables, or string literals.

```
@echo off
if "%1" == "ed" goto ed
if "%1" == "c" goto c
echo ERROR: the %1 argument cannot be identified.
goto end
:ed
  ... code to set up for, launch, and clean up after the text
editor
goto end
:c
  ... code to set up for, launch, and clean up after the c
compiler and its IDE
 goto end
:end
```

Test for NUL can be accomplished by using the IF command as follows:

```
if "%temp%" == "" goto novar
```

The "" marks allow comparison of null strings since they become part of the string - if %temp% has not been set to something, the above becomes ""=="" after normalizing and substituting by the command processor.

A good idea is to convert the strings to all caps prior to comparison, which can be done by passing it thought the PATH command, as string characters are tested for the ASCII value, thus making 'y' and 'Y' two different values, and requiring two different IF statements.

Comparing single characters does not always require the use of quotes. The following is legal syntax, and will run correctly, but may generate unexpected results if the comparison value had quotes around it:

```
IF %TestVar%==Y goto pass
```

Testing for numerical values does not require quotes:

```
IF %TestNumVar%==3 goto three
```

SYNTAX:	EXAMPLE:
`IF [variable] == [comparison] [command]`	`IF %0==2 GOTO TWO`

NUL

NUL is the equivalent to a zero-length string. In evaluation commands, NUL is represented by "" (see IF statement).

NUL can also be used for output redirection. If you wish to run a command in a script, which outputs information to the console, and you don't want the user to see this information, redirection to NUL will allow the command to execute normally, while masking it's output.

```
c:\command\3c905.com >NUL
```

For this example, the NIC card DOS driver is started, and all output from that process is redirected from the console to NUL, resulting in nothing being saved to the HDD or displayed on the screen.

SYNTAX:	EXAMPLE:
`[command] >NUL` `{within evaluation statements} ""`	`C:\COMMAND\3C905.COM >NUL` `IF "%1"=="" THEN GOTO END`

Pause

Allows you to pause the flow from within the batch file, sending a message to the output device `Press Any Key To Continue…`

Once the user presses any key, batch file execution continues.

SYNTAX:	EXAMPLE:
PAUSE	PAUSE

REM (Remark)

The REM (remark) statement, followed by your line of text, is used to place comments in the script file that are ignored by the command interpreter.

The use of two colons. If a line is started using the "`::`" command, the line is parsed as a label, but the parsing is aborted on the second character, so it takes the command interpreter far less time to process it than does a traditional remark, which must be parsed for redirection as well as commands.

One point of note for the REM statement, is that if the REM statement is redirected to a file (say TEMP.TMP) it creates a zero length file named TEMP.TMP, which can then be used in an IF EXIST test, where it will test as existing, but doesn't actually consume disk space, only a directory entry. The curious thing about that use of REM is that you can still include a comment string in the remark without having it go into the target file.

SYNTAX:	EXAMPLE:
`REM [line of text]`	`REM This line is a comment and ignored`

COMMAND INTERPRETER COMMANDS

CALL

CALL is much like CALLs in high level languages. It causes the name of the CALLing batch file and the location in the file to be saved (by COMMAND.COM) so that execution can be resumed at the next line in the file, when the CALLed batch file terminates. Since each CALL consumes some memory until it is cleared by the return of the CALLed program, there is a finite limit to the number of CALLs that can be nested.

CALL lets you invoke a batch file from a batch file without loosing your place in the first one. The two batch files may be different files, or the same file. When they are the same, the effect is that of recursion. CALL causes COMMAND.COM to save the name and file pointer for the current file before invoking the second one and to reenter the original file at the location indicated by the file pointer on its termination. CALL can also be used to invoke executables, though COMMAND.COM doesn't seem to do anything different when you do. However, there is a certain flexibility in using CALL to invoke executables: you can later replace the .EXE file with a same named .BAT file to change the action. It is also safer to use CALL when you don't know for sure that the program you have been told just the name of is in fact a .COM or .EXE, rather than . .BAT file. CALL also is useful in FOR loops both in batch files and from the command line to prevent the loop from terminating prematurely when the action is a batch file. Example:

```
FOR %a in (foo.bat bar.bat baz.bat) do %a
```

will execute only FOO, but

```
FOR %a in (foo.bat bar.bat baz.bat) do CALL %a
```

will execute all three.

SYNTAX:	EXAMPLE:
`CALL [EXE, BAT, CMD, COM]`	`CALL C:\EXAMPLE.BAT`

CD

CD changes the default directory on either the default or given drive. It is important to keep in mind that when changing the directory on a non-default drive the default drive itself is not changed. If you used the code:

```
c:
cd D:\drctory
```

leaves you in the C: drive, but changes the default directory on the D: drive to \drctory. This is sometimes useful, but it should be kept in mind that under Windows, the default directory on any drive is subject to being changed by another program while the batch file is running - it is a very good idea to assume nothing about the default directory. Not even the default directory on the default drive for the window in which the batch file is running can be assumed to be stable.

SYNTAX:	EXAMPLE:
`CD [DIRECTORY]`	`CD \INSTALL\PAST`

DIR

DIR is the command to list the contents of a directory or sub-directory. It has many useful switches, and can be useful in determining an environment within a batch file.

DIR is used as follows:

DIR [drive:][path][filename] [/P] [/W] [/A[[:]attributes]] [/O[[:]sortorder]] [/S] [/B] [/L] [/V]

[drive:][path][filenam e]	Specifies drive, directory, and/or files to list. (Could be enhanced file specification or multiple filespecs.)
/P	Pauses after each screenful of information
/W	Uses wide list format
/A	Displays files with specified attribues D = Directory

	R = Read-Only Files
	H = Hidden Files
	A = Files ready for Archiving
	S = System File
	- = Prefix meaning NOT
/O	List by files in sorted order
	N = By name (alphabetic)
	S = By size (smallest first)
	E = By extension (alphabetic)
	D = By date & time (earliest first)
	G = Group directories first
	A = By Last Access Date (earliest first)
	- = Prefix to reverse order
/S	Displas files in specified directory and all subdirectiries
/B	Uses bare format (no heading information or summary)
/L	Uses lowercase
/V	Verbose mode

Switches may be preset in the DIRCMD environment variable. Override preset switches by prefixing any switch with - (hyphen)--for example, /-W.

ECHO

ECHO is really two commands: one to control the ECHO status (whether the command lines in the file will be displayed, or just the messages they generate) using the ON and OFF arguments, and one to display text on the console. Considering the case of the first usage, if a batch file is always invoked or called by another, only the first one need have the @ECHO OFF command, which will remain in force as long as COMMAND.COM remains in batch mode. Note that this does not affect secondary command processors.

ECHO, without any arguments, simply displays the echo status: either ON or OFF.

ECHO <text> seems to have been provided as a means of issuing messages to the user, but there are other uses. The not well documented syntax ECHO. (note the '.' at the end of the command) ECHOs a blank line. This is quite useful, when piped to certain commands, for supplying the <Enter> required to terminate the command:

```
@ECHO OFF
ECHO. | DATE
ECHO. | TIME
```

ECHO is also used in a similar way with 'y' or 'n' to provide an automatic response to a command requiring a Y/N response. The most common use of this is, perhaps, to automatically delete the garbage in the TEMP directory during AUTOEXEC.BAT:

```
echo y | del c:\temp\*.*
```

SYNTAX:	EXAMPLE:
ECHO [ON\OFF] ECHO. ECHO <text> ECHO	@ECHO OFF ECHO. ECHO Line of Text to Output Device ECHO

GOTO

GOTO directs the flow of the batch file either on run-time decisions, or hard-coded flow. In a batch file, when command.com interprets the GOTO command, and the destination label is below the calling GOTO command, command.com can quickly jump to the label below the call, and execute the commands. However, if the label is above the calling GOTO, command.com must read to the end of the batch file, then start from the beginning, and read down until it encounters the label.

In `GOTO` statements, label names may well be wholly or partially variable. This can often simplify program flow control. For example, if you need to temporarily direct the flow from the main program to a generic, multi-use function, and then have flow return to the division point, you can implement the use of variable in the label name. Example:

```
@echo off

:f1
set return=1
goto function

:r1
echo The function returned to R1

f2
set return=2
goto function

:r2
echo The function returned to R2

:f3
```

```
set return=3
goto function

:r3
echo The function returned to R3
goto end

:function
echo The function was called from F%return%
goto r%return%

:end
```

Note that either a space or a colon can be used between the GOTO and label, which can be useful when writing FOR statements like:

```
for %%a in (echo goto:end) do %%a Message
```

Which will echo MESSAGE and then GOTO label END.

SYNTAX:	EXAMPLE:

`GOTO [label]`	`[command]`
	`IF %VARIABLE%==Y GOTO YES`
	`[command]`
	
	`:YES`

ADDITIONAL MS-DOS UTILITIES

FIND.EXE

Find searches for a text string in a file or files, and returns the line of text where the search parameters are found.

FIND [/V] [/C] [/N] [/I] "string" [[drive:][path]filename[...]]

/V	Displays all lines NOT containing the specified string
/C	Displays only the count of lines containing the string
?n	Displays line numbers with the displayed lines
?I	Ignores the case of characters when searching for the string
“string”	Specifies the text string to find
[drive:] [path] filename	Specifies a file or files to search

If a pathname is not specified, FIND searches the text typed at the prompt or piped from another command.

MOVE.EXE

Moves files and renames files and directories.

To move one or more files: MOVE [/Y | /-Y] [drive:][path]filename1[,...] destination

To rename a directory: MOVE [/Y | /-Y] [drive:][path]dirname1 dirname2

[drive:] [path] filename1	Specifies the location and name of the file or files you want to move

Destination	Specifies the new location of the file.. Destination can consist of a drive letter and colon, a directory name, or a combination. If you are moving only one file, you can also include a filename if you want to rename the file when you move it.
[drive:] [path] dirname	Specifies the directory you want to rename
dirname2	Specifies the new name of the dirctory
/Y	Suppresses prompting to confirm creation of a directory or overwriting of the destination
/-Y	Causes prompting to confirm creation of a directory or overwriting of the destination

The switch /Y may be present in the COPYCMD environment variable. This may be overridden with /-Y on the command line.

XCOPY.EXE

Copies files and directory trees.

XCOPY source [destination] [/A | /M] [/D[:date]] [/P] [/S [/E]] [/W] [/C] [/I] [/Q] [/F] [/L] [/H] [/R] [/T] [/U] [/K] [/N]

Source	Specifies the file(s) to copy.
Destination	Specifies the location and/or name of new files.
/A	Copies files with the archive attribute set, doesn't change the attribute.
/M	Copies files with the archive attribute set, turns off the archive attribute.
/D:date	Copies files changed on or after the specified date. If no date is given, copies only those files whose source time is newer than the destination time.
/P	Prompts you before creating each destination file.
/S	Copies directories and subdirectories except empty ones.
/E	Copies directories and subdirectories, including empty ones. Same as /S /E. May be used to modify /T.
/W	Prompts you to press a key before copying.
/C	Continues copying even if errors occur.
/I	If destination does not exist and copying more than one file, assumes that destination must be a directory.
/Q	Does not display file names while copying.
/F	Displays full source and destination file names while copying.
/L	Displays files that would be copied.

/H	Copies hidden and system files also.
/R	Overwrites read-only files.
/T	Creates directory structure, but does not copy files. Does not include empty directories or subdirectories. /T /E includes empty directories and subdirectories.
/U	Updates the files that already exist in destination.
/K	Copies attributes. Normal Xcopy will reset read-only attributes.
/Y	Overwrites existing files without prompting.
/-Y	Prompts you before overwriting existing files.
/N	Copy using the generated short names.

FUNCTIONS AND OBJECTS

One problem with batch file writing is that you need to write the same code over and over to accomplish the same task. You can avoid re-writing you code by placing your repetitive code in separate files, in library files, or as returnable labels from within the same file.

"VOID" Functions

When a task must be accomplished that does not require any input from either the user, or the program itself, and the program and the user does not need any information back from the task when it completes, the task can be considered a VOID function.

Lets take the case of mapping or unmapping drives. If you frequently need to verify that your environment is clear of unwanted mappings, or if you need to clear mappings you or the system have created in order to map differently,, you have a VOID task. For this task, you don't need to tell your script lines anything special for them to unmap given drives, and you don't need to know anything special from the script lines when they are finished. The following script unmaps drives on a Windows 95 / NT system:

UNMAP.BAT

```
:UNMAP
NET USE D: /D
NET USE E: /D
NET USE F: /D

:END
```

To use this function, you can use the CALL function from within a parent batch file to utilize these lines:

EXAMPLE.BAT

```
@ECHO OFF
ECHO <Some line of text to the user or whatever>
SET RETURN=1
CALL UNMAP.BAT
NET USE D: \\SERVER\SHARE
. . .
:END
```

"Value" Functions

Sometimes it is necessary to send data to, and receive altered data from functions within your batch file. Batch files only really allow variable to be passed from either the command line or from within system variables.

Take the example of converting strings to uppercase. Batch files see 'y' and 'Y' as two different values, even though to the user, they are the same. The script writer can write the extra code check for the existence of upper or lower case values, but that is two lines of code for each input. Writing a function to change the variable into uppercase would be very handy – and easy to do. So we create a function that takes in a string (as defined as characters up to a separator character) and return the uppercase equivalent:

For this example, we are hard-coding the variable's value in the batch file.

TEST.BAT

```
@echo off
set string=lowerstring
echo %string%
call toupper.bat %string%
set string=%toupper%
set toupper=
echo Passed and back: %string%
:end
```

TOUPPER.BAT

```
set oldpath=%path%
path %1
set toupper=%path%
set path=%oldpath%
set oldpath=
```

```
goto end
:end
```

The result is this:

```
C:\
lowerstring
Passed and back: LOWERSTRING
```

The PATH statement in DOS converts everything to uppercase. The TOUPPER batch file preserves your computers current PATH statement, uses the PATH statement to convert the passed string to uppercase, resets the PATH, then sets a system variable to the uppercase version of the passed string. The calling batch file uses the system variable.

RECURSION

Batch file recursion is when a batch file (or batch file function) continuously calls itself with data until the required data is achieved. Take the example:

RECURS.BAT

```
@ECHO OFF
IF "%1"=="" GOTO SETUP
goto %1

:a
Echo Value is currently %1 - resetting value to B
set value=B
call recurs.bat %value%
goto end

:b
Echo Value is currently %1 - resetting value to C
set value=C
call recurs.bat %value%
goto end

:c
echo Final Amount is: %value%
goto end

:setup
echo There is no current Value, setting value to A
set value=A
call recurs.bat %value%

:end
```

This batch file calls itself with altered data until the desired data is received. The output of the program is:

```
There is no current Value, setting value to A
Value is currently A - resetting value to B
Value is currently B - resetting value to C
Final Amount is: C
```

NOTES

Calling Files in Sub-folders

You can call script files that are on the same level or within subfolders without knowing where your file actually is by simply indicating the subdirectory prior to the script. However, I have noticed that if a file is called this way, the resulting intrinsic %0 variable might not contain the whole path, but rather simply the subdirectory name and then file name.

SYNTAX:	EXAMPLE:
`SUBFOLDER\NEXTFILE.BAT`	