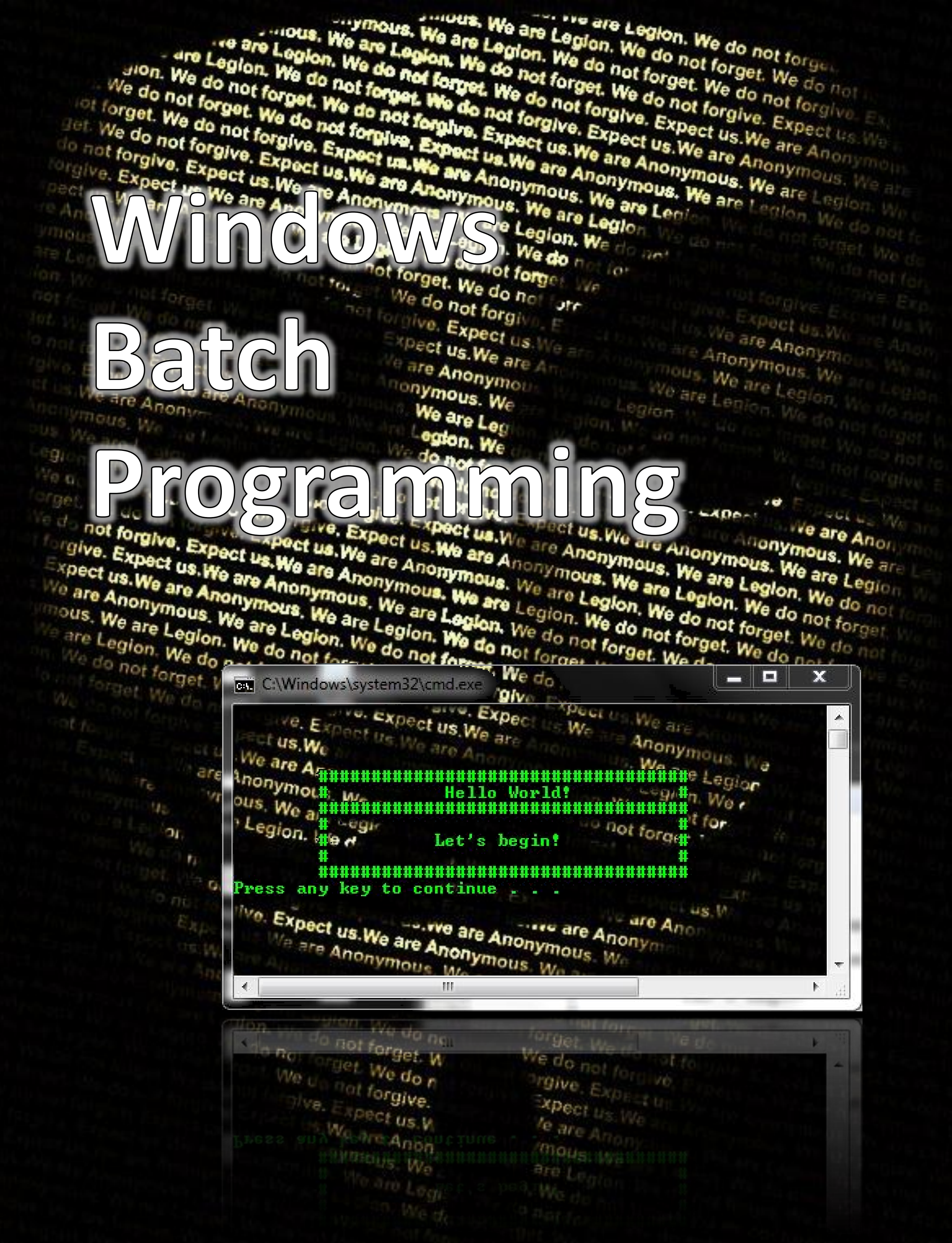
Windows Batch Programming



```
C:\Windows\system32\cmd.exe
#######################################
#           Hello World!              #
#######################################
#                                     #
#          Let's begin!               #
#                                     #
#######################################
Press any key to continue . . .
```

Author: Steve Jansen

Layout by: Dishendra

Index

Getting Started

Getting Started with Windows Batch Scripting

Windows batch scripting is incredibly accessible – it works on just about any modern Windows machine. You can create and modify batch scripts on just about any modern Windows machine. The tools come out of the box: the Windows command prompt and a text editor like Notepad.exe. It's definitely far from the best shell scripting langauge, but, it gets the job done. It's my "duct tape" for Windows.

Launching the Command Prompt

Windows gurus launch the command prompt using the keyboard shortcut `Windows Logo Key`+`R` (i.e., "Run") > Type `cmd.exe` then `Enter`. This is way faster than navigating the Windows Start Menu to find the Command Prompt.

Editing Batch Files

The universal text editor for batch files is Notepad (`Windows Logo Key` + `R` > Type `notepad` then `Enter`). Since batch files are just ASCII text, you can probably use just about any text editor or word processor. Very few editors do anything special for Batch files like syntax highlighting or keyword support, so notepad is good enough fine and will likely be installed on just about every Windows system you encounter.

Viewing Batch Files

I would stick with Notepad for viewing the contents of a batch file. In Windows Explorer (aka, "My Computer"), you should be able to view a batch file in Notepad by right clicking the file and seleting `Edit` from the context menu. If you need to view the contents within a command prompt window itself, you can use a DOS command like `TYPE myscript.cmd` or `MORE myscript.cmd` or `EDIT myscript.cmd`

Batch File Names and File Extensions

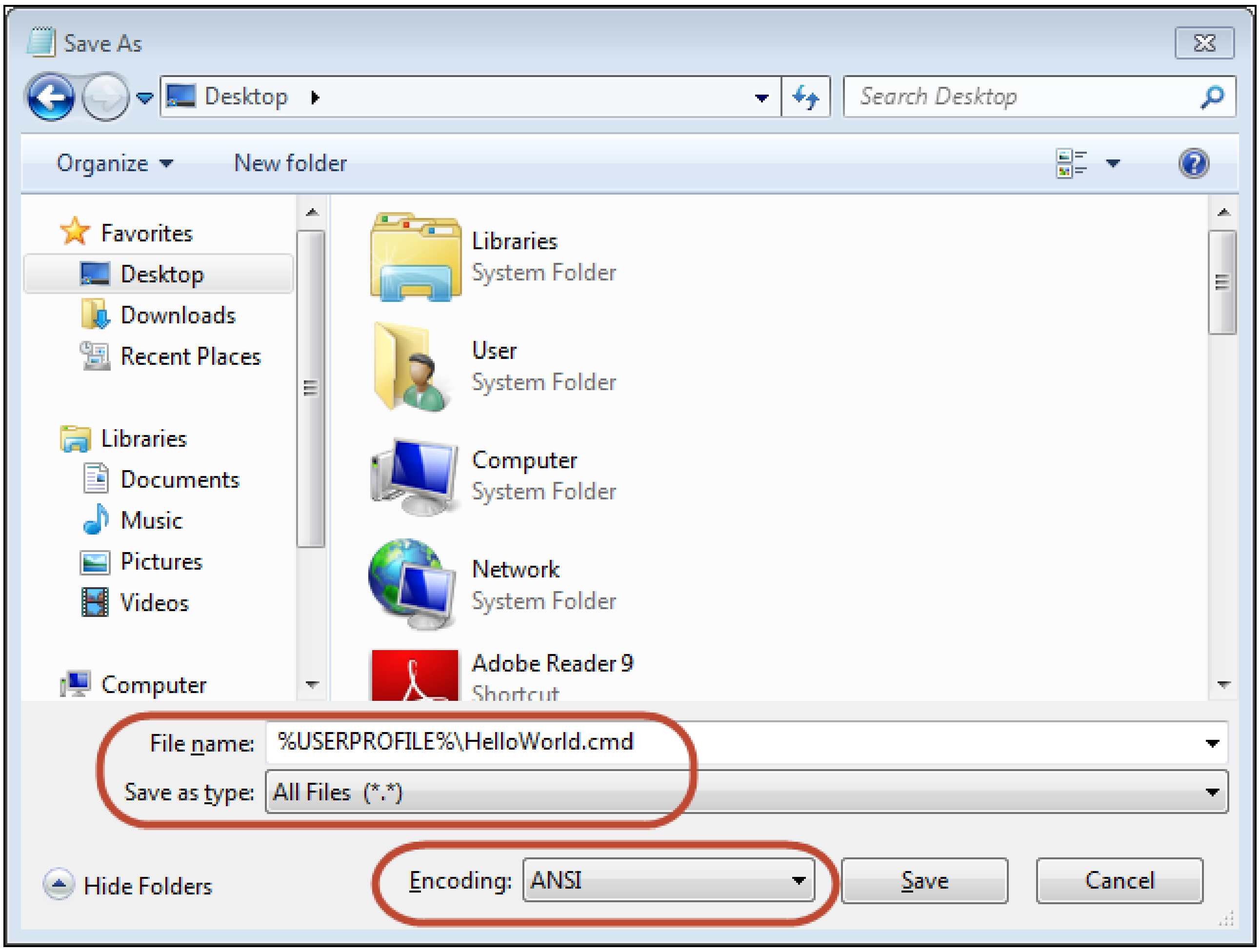
Assuming you are using Windows XP or newer, I recommend saving your batch files with the file extension `.cmd`. Some seriously outdated Windows versions used `.bat`, though I recommend sticking with the more modern `.cmd` to avoid some rare side effects with .bat files.

With the `.cmd` file extension, you can use just about filename you like. I recommend avoiding spaces in filenames, as spaces only create headaches in shell scripting. Pascal casing your filenames is an easy way to avoid spaces (e.g., `HelloWorld.cmd` instead of `Hello World.cmd`). You can also use punctuation characters like `.` or `-` or `_` (e.g. `Hello.World.cmd`, `Hello-World.cmd`, `Hello_World.cmd`).

Another thing with names to consider is avoiding names that use the same name of any built-in commands, system binaries, or popular programs. For example, I would avoid naming a script `ping.cmd` since there is a widely used system binary named `ping.exe`. Things might get very confusing if you try to run `ping` and inadvertently call `ping.cmd` when you really wanted `ping.cmd`. (Stay tuned for how this could happen.) I might called the script `RemoteHeartbeat.cmd` or something similar to add some context to the script's name and also avoid any naming collisions with any other executable files. Of course, there could be a very unique circumstance in which you want to modify the default behavior of `ping` in which this naming suggestion would not apply.

Saving Batch Files in Windows

Notepad by default tries to save all files as plain jane text files. To get Notepad to save a file with a `.cmd` extension, you will need to change the "Save as type" to "All Files (.)". See the screenshot below for an example of saving a script named "HelloWorld.cmd" in Notepad.


File name: %USERPROFILE%\HelloWorld.cmd

Save as type: All Files (*.*)

Encoding: ANSI

SIDEBAR: I've used a shortcut in this screenshot that you will learn more about later. I've saved the file to my "user profile folder" by naming the file `%USERPROFILE%\HelloWorld.cmd`. The `%USERPROFILE%` keyword is the Windows environmental variable for the full path to your user profile folder. On newer Windows systems, your user profile folder will typically be `C:\Users\<your username>`. This shortcut saves a little bit of time because a new command prompt will generally default the "working directory" to your user profile folder. This lets you run `HelloWorld.cmd` in a new command prompt without changing directories beforehand or needing to specify the path to the script.

Running your Batch File

The easy way to run your batch file in Windows is to just double click the batch file in Windows Explorer (aka "My Computer"). Unfortunately, the command prompt will not give you much of a chance to see the output and any errors. The command prompt window for the script will disappear as soon as the script exits. (We will learn how to handle this problem in Part 10 – Advanced Tricks).

When editing a new script, you will likely need to run the batch file in an existing command window. For newbies, I think the easiest foolproof way to run your script is to drag and drop the script into a command prompt window. The command prompt will enter the full path to your script on the command line, and will quote any paths containing spaces.

Some other tips to running batch files:

- You can recall previous commands using the up arrow and down arrow keys to navigate the command line history.
- I usually run the script as `%COMPSPEC% /C /D "C:\Users\User\SomeScriptPath.cmd" Arg1 Arg2 Arg3` This runs your script in a new command prompt child process. The `/C` option instructs the child process to quit when your script quits. The `/D` disables any auto-run scripts (this is optional, but, I use auto-run scripts). The reason I do this is to keep the command prompt window from automatically closing should my script, or a called script, call the `EXIT` command. The `EXIT` command automatically closes the command prompt window unless the `EXIT` is called from a child command prompt process. This is annoying because you lose any messages printed by your script.

Comments

The official way to add a comment to a batch file is with the `REM` (Remark) keyword:

```
 REM This is a comment!
```

The power user method is to use `::`, which is a hack to uses the the label operator `:` twice, which is almost always ignored.

Most power authors find the `::` to be less distracting than `REM`. Be warned though there are a few places where `::` will cause errors.

```
  :: This is a comment too!! (usually!)
```

For example, a `FOR` loop will error out with `::` style comments. Simply fall back to using `REM` if you think you have a situation like this.

Silencing Display of Commands in Batch Files

The first non-comment line of a batch file is usually a command to turn off printing (ECHO’ing) of each batch file line.

```
 @ECHO OFF
```

The `@` is a special operator to suppress printing of the command line. Once we set ECHO’ing to off, we won’t need the `@` operator again in our script commands.

You restore printing of commands in your script with:

```
ECHO ON
```

Upon exit of your script, the command prompt will automatically restore ECHO to it’s previous state.

Debugging Your Scripts

Batch files invole a lot of trial and error coding. Sadly, I don’t know of any true debugger for Windows batch scripts. Worse yet, I don’t know of a way to put the command processor into a verbose state to help troubleshoot the script (this is the common technique for Unix/Linux scripts.) Printing custom ad-hoc debugging messages is about your only option using the `ECHO` command. Advanced script writers can do some trickery to selectively print debugging messages, though, I prefer to remove the debugging/instrumentation code once my script is functioning as desired.

Variables

Variable Declaration

DOS does not require declaration of variables. The value of undeclared/uninitialized variables is an empty string, or `""`. Most people like this, as it reduces the amount of code to write. Personally, I'd like the option to require a variable is declared before it's used, as this catches silly bugs like typos in variable names.

Variable Assignment

The `SET` command assigns a value to a variable.

```
SET foo=bar
```

NOTE: Do not use whitespace between the name and value; `SET foo = bar` will *not* work but `SET foo=bar` will work.

The `/A` switch supports arthimetic operations during assigments. This is a useful tool if you need to validated that user input is a numerical value.

```
SET /A four=2+2
4
```

A common convention is to use lowercase names for your script's variables. System-wide variables, known as environmental variables, use uppercase names. These environmental describe where to find certain things in your system, such as `%TEMP%` which is path for temporary files. DOS is case insensitive, so this convention isn't enforced but it's a good idea to make your script's easier to read and troubleshoot.

WARNING: `SET` will always overwrite (clobber) any existing variables. It's a good idea to verify you aren't overwriting a system-wide variable when writing a script. A quick `ECHO %foo%` will confirm that the variable `foo` isn't an existing variable. For example, it might be tempting to name a variable "temp", but, that would change the meaning of the widely used "%TEMP%" environmental varible. DOS includes some "dynamic" environmental variables that behave more like commands. These dynamic varibles include `%DATE%`, `%RANDOM%`, and `%CD%`. It would be a bad idea to overwrite these dynamic variables.

Reading the Value of a Variable

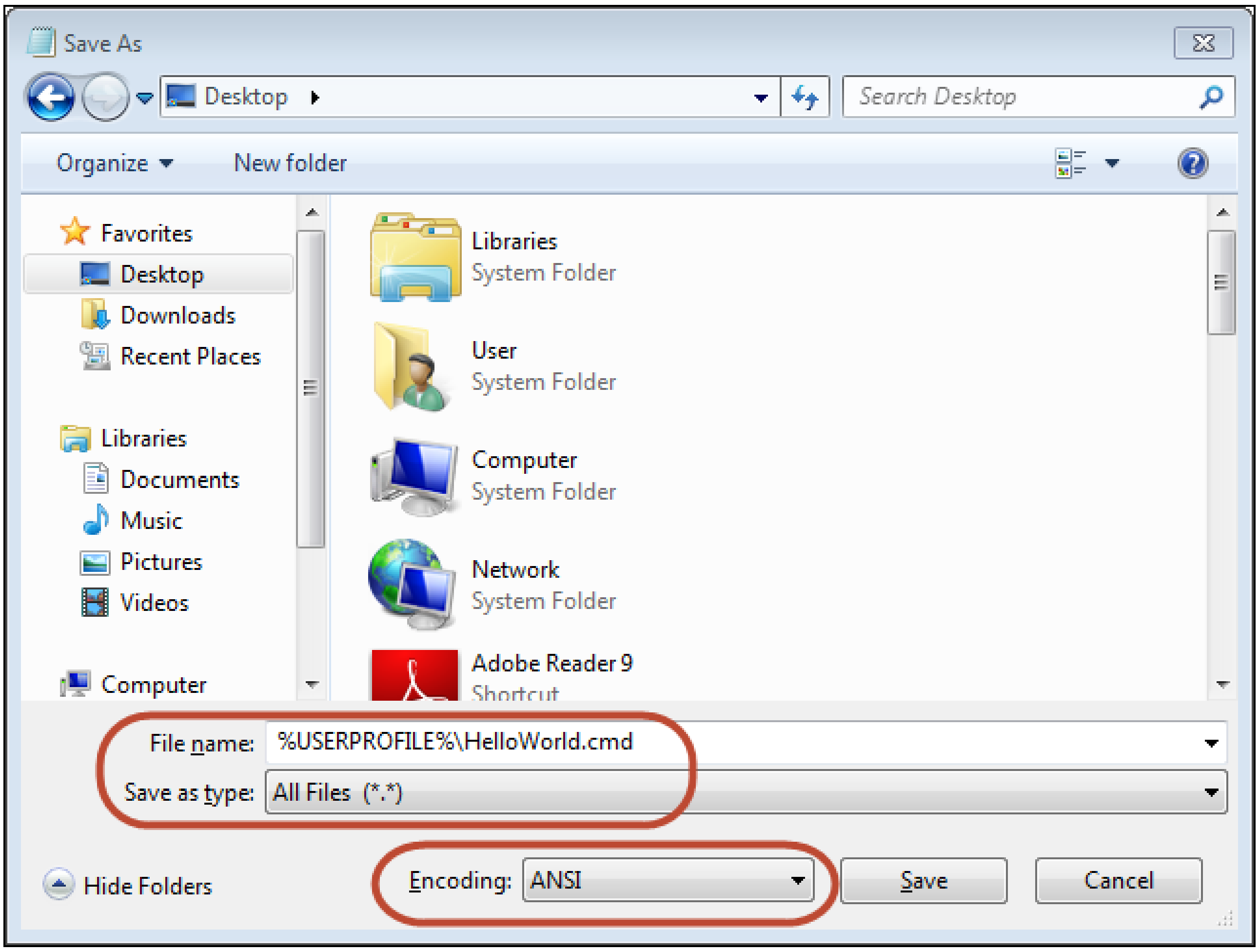
In most situations you can read the value of a variable by prefixing and postfixing the variable name with the `%` operator. The example below prints the current value of the variable `foo` to the console output.

```
C:\> SET foo=bar
C:\> ECHO %foo%
bar
```

There are some special situations in which variables do not use this `%` syntax. We'll discuss these special cases later in this series.

Listing Existing Variables

The `SET` command with no arguments will list all variables for the current command prompt session. Most of these varaiables will be system-wide environmental variables, like `%PATH%` or `%TEMP%`.



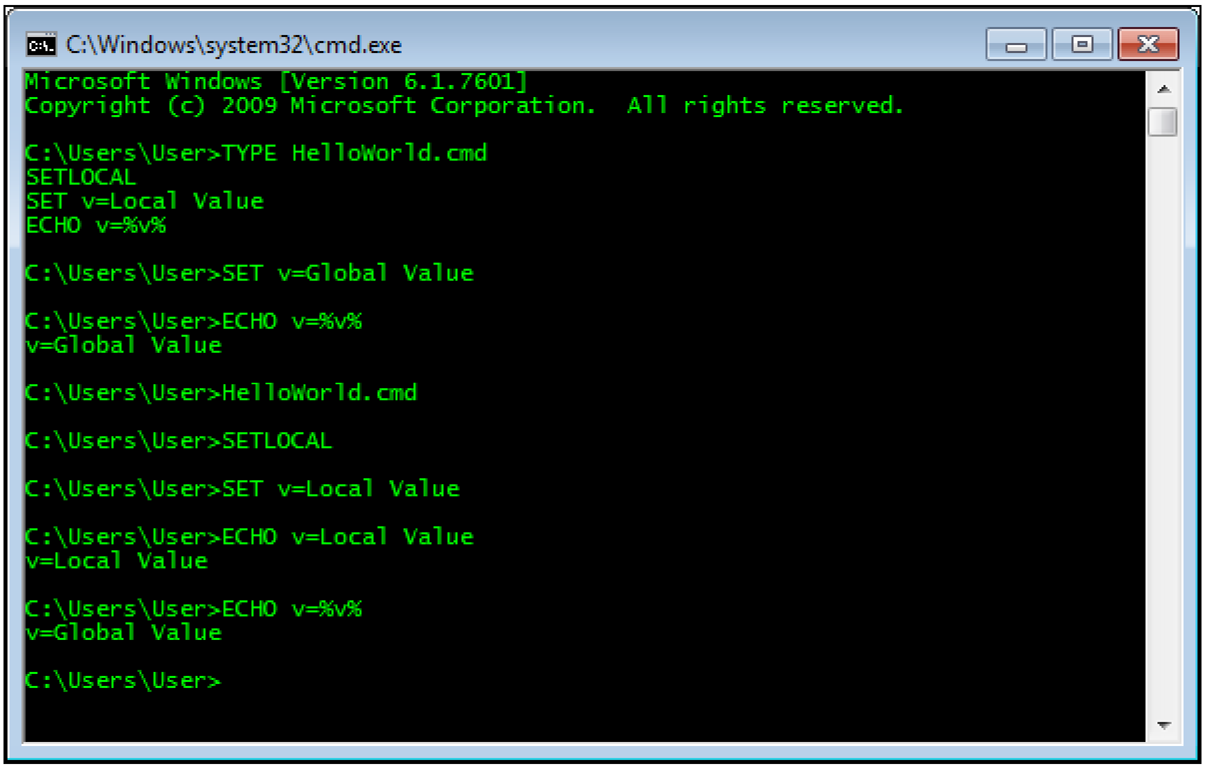
File name: %USERPROFILE%\HelloWorld.cmd
Save as type: All Files (*.*)
Encoding: ANSI

NOTE: Calling `SET` will list all regular (static) variables for the current session. This listing excludes the dynamic environmental variables like `%DATE%` or `%CD%`. You can list these dynamic variables by viewing the end of the help text for SET, invoked by calling `SET /?`

Variable Scope (Global vs Local)

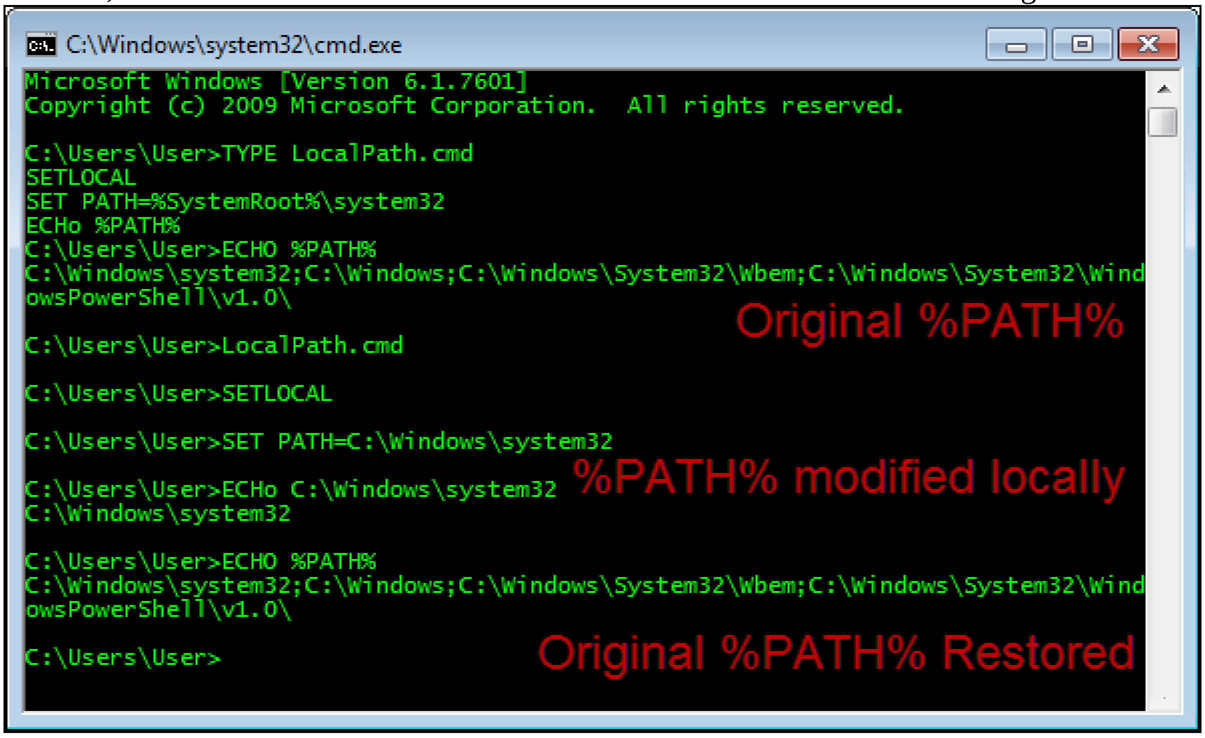
By default, variables are global to your entire command prompt session. Call the `SETLOCAL` command to make variables local to the scope of your script. After calling `SETLOCAL`, any variable assignments revert upon calling `ENDLOCAL`, calling `EXIT`, or when execution reaches the end of file (EOF) in your script.

This example demonstrates changing an existing variable named `foo` within a script named `HelloWorld.cmd`. The shell restores the original value of `%foo%` when `HelloWorld.cmd` exits.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\User>TYPE HelloWorld.cmd
SETLOCAL
SET v=Local Value
ECHO v=%v%

C:\Users\User>SET v=Global Value

C:\Users\User>ECHO v=%v%
v=Global Value

C:\Users\User>HelloWorld.cmd

C:\Users\User>SETLOCAL

C:\Users\User>SET v=Local Value

C:\Users\User>ECHO v=Local Value
v=Local Value

C:\Users\User>ECHO v=%v%
v=Global Value

C:\Users\User>
```

A real life example might be a script that modifies the system-wide `%PATH%` environmental variable, which is the list of directories to search for a command when executing a command.



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\User>TYPE LocalPath.cmd
SETLOCAL
SET PATH=%SystemRoot%\system32
ECHo %PATH%
C:\Users\User>ECHO %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Wind
owsPowerShell\v1.0\

C:\Users\User>LocalPath.cmd

C:\Users\User>SETLOCAL

C:\Users\User>SET PATH=C:\Windows\system32

C:\Users\User>ECHo C:\Windows\system32
C:\Windows\system32

C:\Users\User>ECHO %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Wind
owsPowerShell\v1.0\

C:\Users\User>
```

Special Variables

There are a few special situations where variables work a bit differently. The arguments passed on the command line to your script are also variables, but, don't use the `%var%` syntax. Rather, you read each argument using a single `%` with a digit 0-9, representing the ordinal position of the argument. You'll see this same style used later with a hack to create functions/subroutines in batch scripts.

There is also a variable syntax using `!`, like `!var!`. This is a special type of situation called delayed expansion. You'll learn more about delayed expansion in when we discuss conditionals (if/then) and looping.

Command Line Arguments to Your Script

You can read the command line arguments passed to your script using a special syntax. The syntax is a single `%` character followed by the ordinal position of the argument from `0` – `9`. The zero ordinal argument is the name of the batch file itself. So the variable `%0` in our script `HelloWorld.cmd` will be "HelloWorld.cmd".

The command line argument variables are * `%0`: the name of the script/program as called on the command line; always a non-empty value * `%1`: the first command line argument; empty if no arguments were provided * `%2`: the second command line argument; empty if a second argument wasn't provided * …: * `%9`: the ninth command line argument

NOTE: DOS does support more than 9 command line arguments, however, you cannot directly read the 10th argument of higher. This is because the special variable syntax doesn't recognize `%10` or higher. In fact, the shell reads `%10` as postfix the `%0` command line argument with the string "0". Use the `SHIFT` command to pop the first argument from the list of arguments, which "shifts" all arguments one place to the left. For example, the the second argument shifts from position `%2` to `%1`, which then exposes the 10th argument as `%9`. You will learn how to process a large number of arguments in a loop later in this series.

Tricks with Command Line Arguments

Command Line Arguments also support some really useful optional syntax to run quasi-macros on command line arguments that are file paths. These macros are called variable substitution support and can resolve the path, timestamp, or size of file that is a command line argument. The documentation for this super useful feature is a bit hard to find – run 'FOR /?' and page to the end of the output.

- `%~1` removes quotes from the first command line argument, which is super useful when working with arguments to file paths. You will need to quote any file paths, but, quoting a file path twice will cause a file not found error.

```
SET myvar=%~1
```

- `%~f1` is the full path to the folder of the first command line argument

- `%~fs1` is the same as above but the extra `s` option yields the DOS 8.3 short name path to the first command line argument (e.g., `C:\PROGRA~1` is usually the 8.3 short name variant of `C:\Program Files`). This can be helpful when using third party scripts or programs that don't handle spaces in file paths.
- `%~dp1` is the full path to the parent folder of the first command line argument. I use this trick in nearly every batch file I write to determine where the script file itself lives. The syntax `SET parent=%~dp0` will put the path of the folder for the script file in the variable `%parent%`.
- `%~nx1` is just the file name and file extension of the first command line argument. I also use this trick frequently to determine the name of the script at runtime. If I need to print messages to the user, I like to prefix the message with the script's name, like `ECHO %~n0: some message` instead of `ECHO some message` . The prefixing helps the end user by knowing the output is from the script and not another program being called by the script. It may sound silly until you spend hours trying to track down an obtuse error message generated by a script. This is a nice piece of polish I picked up from the Unix/Linux world.

Some Final Polish

I always include these commands at the top of my batch scripts:

```
SETLOCAL ENABLEEXTENSIONS
SET me=%~n0
SET parent=%~dp0
```

The `SETLOCAL` command ensures that I don't clobber any existing variables after my script exits. The `ENABLEEXTENSIONS` argument turns on a very helpful feature called command processor extensions. Trust me, you want command processor extensions. I also store the name of the script (without the file extension) in a variable named `%me%`; I use this variable as the prefix to any printed messages (e.g. `ECHO %me%: some message`). I also store the parent path to the script in a variable named `%parent%`. I use this variable to make fully qualified filepaths to any other files in the same directory as our script.

Return Codes

Today we'll cover return codes as the right way to communicate the outcome of your script's execution to the world. Sadly, even skilled Windows programmers overlook the importance of return codes.

Return Code Conventions

By convention, command line execution should return zero when execution succeeds and non-zero when execution fails. Warning messages typically don't effect the return code. What matters is did the script work or not?

Checking Return Codes In Your Script Commands

The environmental variable `%ERRORLEVEL%` contains the return code of the last executed program or script. A very helpful feature is the built-in DOS commands like `ECHO`, `IF`, and `SET` will preserve the existing value of `%ERRORLEVEL%`.

The conventional technique to check for a non-zero return code using the `NEQ` (Not-Equal-To) operator of the `IF` command:

```
IF %ERRORLEVEL% NEQ 0 (
  REM do something here to address the error
)
```

Another common technique is:

```
IF ERRORLEVEL 1 (
  REM do something here to address the error
)
```

The `ERRORLEVEL 1` statement is true when the return code is any number equal to or greater than 1. However, I don't use this technique because programs can return negative numbers as well as positive numbers. Most programs rarely document every possible return code, so I'd rather explicity check for non-zero with the `NEQ 0` style than assuming return codes will be 1 or greater on error.

You may also want to check for specific error codes. For example, you can test that an executable program or script is in your PATH by simply calling the program and checking for return code 9009.

```
SomeFile.exe
IF %ERRORLEVEL% EQU 9009 (
  ECHO error - SomeFile.exe not found in your PATH
)
```

It's hard to know this stuff upfront – I generally just use trial and error to figure out the best way to check the return code of the program or script I'm calling. Remember, this is duct tape programming. It isn't always pretty, but, it gets the job done.

Conditional Execution Using the Return Code

There’s a super cool shorthand you can use to execute a second command based on the success or failure of a command. The first program/script must conform to the convention of returning 0 on success and non-0 on failure for this to work.

To execute a follow-on command after sucess, we use the `&&` operator:

```
SomeCommand.exe && ECHO SomeCommand.exe succeeded!
```

To execute a follow-on command after failure, we use the `||` operator:

```
SomeCommand.exe || ECHO SomeCommand.exe failed with return code
%ERRORLEVEL%
```

I use this technique heavily to halt a script when any error is encountered. By default, the command processor will continue executing when an error is raised. You have to code for halting on error.

A very simple way to halt on error is to use the `EXIT` command with the `/B` switch (to exit the current batch script context, and not the command prompt process). We also pass a specific non-zero return code from the failed command to inform the caller of our script about the failure.

```
SomeCommand.exe || EXIT /B 1
```

A simliar technique uses the implicit GOTO label called `:EOF` (End-Of-File). Jumping to EOF in this way will exit your current script with the return code of 1.

```
SomeCommand.exe || GOTO :EOF
```

Tips and Tricks for Return Codes

I recommend sticking to zero for success and return codes that are positive values for DOS batch files. The positive values are a good idea because other callers may use the `IF ERRORLEVEL 1` syntax to check your script.

I also recommend documenting your possible return codes with easy to read `SET` statements at the top of your script file, like this:

```
SET /A ERROR_HELP_SCREEN=1
SET /A ERROR_FILE_NOT_FOUND=2
```

Note that I break my own convention here and use uppercase variable names – I do this to denote that the variable is constant and should not be modified elsewhere. Too bad DOS doesn’t support constant values like Unix/Linux shells.

Some Final Polish

One small piece of polish I like is using return codes that are a power of 2.

```
SET /A ERROR_HELP_SCREEN=1
SET /A ERROR_FILE_NOT_FOUND=2
SET /A ERROR_FILE_READ_ONLY=4
SET /A ERROR_UNKNOWN=8
```

This gives me the flexibility to bitwise OR multiple error numbers together if I want to record numerous problems in one error code. This is rare for scripts intended for interactive use, but, it can be super helpful when writing scripts you support but you don't have access to the target systems.

```
@ECHO OFF
SETLOCAL ENABLEEXTENSIONS

SET /A errno=0
SET /A ERROR_HELP_SCREEN=1
SET /A ERROR_SOMECOMMAND_NOT_FOUND=2
SET /A ERROR_OTHERCOMMAND_FAILED=4

SomeCommand.exe
IF %ERRORLEVEL% NEQ 0 SET /A errno^|=%ERROR_SOMECOMMAND_NOT_FOUND%

OtherCommand.exe
IF %ERRORLEVEL% NEQ 0 (
    SET /A errno^|=%ERROR_OTHERCOMMAND_FAILED%
)

EXIT /B %errno%
```

If both SomeCommand.exe and OtherCommand.exe fail, the return cde will be the bitwise combination of 0x1 and 0x2, or decimal 3. This return code tells me that both errors were raised. Even better, I can repeatedly call the bitwise OR with the same error code and still interpret which errors were raised.

Stdin, Stdout, Stderr

DOS, like Unix/Linux, uses the three universal "files" for keyboard input, printing text on the screen, and the printing errors on the screen. The "Standard In" file, known as stdin, contains the input to the program/script. The "Standard Out" file, known as stdout, is used to write output for display on the screen. Finally, the "Standard Err" file, known as stderr, contains any error messages for display on the screen.

File Numbers

Each of these three standard files, otherwise known as the standard streams, are referernced using the numbers 0, 1, and 2. Stdin is file 0, stdout is file 1, and stderr is file 2.

Redirection

A very common task in batch files is sending the output of a program to a log file. The `>` operator sends, or redirects, stdout or stderr to another file. For example, you can write a listing of the current directory to a text file:

```
DIR > temp.txt
```

The `>` operator will overwrite the contents of temp.txt with stdout from the DIR command. The `>>` operator is a slight variant that appends the output to a target file, rather than overwriting the target file.

A common technique is to use `>` to create/overwrite a log file, then use `>>` subsequently to append to the log file.

```
SomeCommand.exe   > temp.txt
OtherCommand.exe >> temp.txt
```

By default, the `>` and `>>` operators redirect stdout. You can redirect stderr by using the file number `2` in front of the operator:

```
DIR SomeFile.txt  2>> error.txt
```

You can even combine the stdout and stderr streams using the file number and the `&` prefix:

```
DIR SomeFile.txt 2>&1
```

This is useful if you want to write both stdout and stderr to a single log file.

```
DIR SomeFile.txt > output.txt 2>&1
```

To use the contents of a file as the input to a program, instead of typing the input from the keyboard, use the `<` operator.

```
SORT < SomeFile.txt
```

Suppressing Program Output

The pseudofile `NUL` is used to discard any output from a program. Here is an example of emulating the Unix command `sleep` by calling ping against the loopback address. We redirect stdout to the `NUL` device to avoid printing the output on the command prompt screen.

```
PING 127.0.0.1 > NUL
```

Redirecting Program Output As Input to Another Program

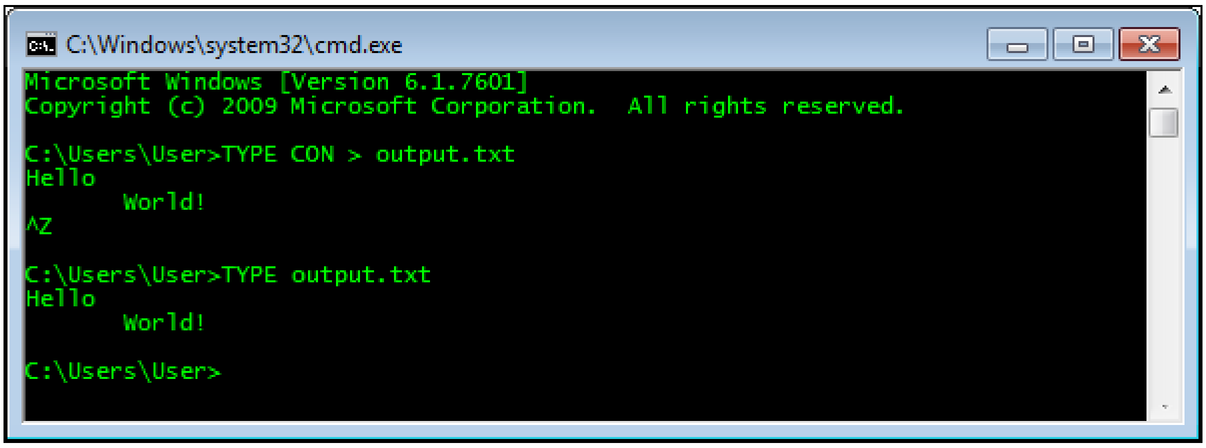
Let's say you want to chain together the output of one program as input to another. This is known as "piping" output to another program, and not suprisingly we use the pipe character `|` to get the job done. We'll sort the output of the DIR commmand.

```
DIR /B | SORT
```

A Cool Party Trick

You can quickly create a new text file, say maybe a batch script, from just the command line by redirecting the command prompt's own stdin, called `CON`, to a text file. When you are done typing, hit `CTRL+Z`, which sends the end-of-file (EOF) character.

```
TYPE CON > output.txt
```



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\User>TYPE CON > output.txt
Hello
        World!
^Z

C:\Users\User>TYPE output.txt
Hello
        World!

C:\Users\User>
```

There are a number of other special files on DOS that you can redirect, however, most are a bit dated like like LPT1 for parallel portt printers or COM1 for serial devices like modems.

If/Then Conditionals

Computers are all about 1’s and 0’s, right? So, we need a way to handle when some condition is 1, or else do something different when it’s 0.

The good news is DOS has pretty decent support for if/then/else conditions.

Checking that a File or Folder Exists

```
IF EXIST "temp.txt" ECHO found
```

Or the converse:

```
IF NOT EXIST "temp.txt" ECHO not found
```

Both the true condition and the false condition:

```
IF EXIST "temp.txt" (
    ECHO found
) ELSE (
    ECHO not found
)
```

NOTE: It’s a good idea to always quote both operands (sides) of any IF check. This avoids nasty bugs when a variable doesn’t exist, which causes the the operand to effectively disappear and cause a syntax error.

Checking If A Variable Is Not Set

```
IF "%var%"=="" (SET var=default value)
```

Or

```
IF NOT DEFINED var (SET var=default value)
```

Checking If a Variable Matches a Text String

```
SET var=Hello, World!

IF "%var%"=="Hello, World!" (
    ECHO found
)
```

Or with a case insensitive comparison

```
IF /I "%var%"=="hello, world!" (
    ECHO found
)
```

Artimetic Comparisons

```
SET /A var=1

IF /I "%var%" EQU "1" ECHO equality with 1

IF /I "%var%" NEQ "0" ECHO inequality with 0

IF /I "%var%" GEQ "1" ECHO greater than or equal to 1

IF /I "%var%" LEQ "1" ECHO less than or equal to 1
```

Checking a Return Code

```
IF /I "%ERRORLEVEL%" NEQ "0" (
    ECHO execution failed
)
```

Loops

Looping through items in a collection is a frequent task for scripts. It could be looping through files in a directory, or reading a text file one line at a time.

Old School with GOTO

The old-school way of looping on early versions of DOS was to use labels and GOTO statements. This isn't used much anymore, though it's useful for looping through command line arguments.

```
:args
SET arg=%~1
ECHO %arg%
SHIFT
GOTO :args
```

New School with FOR

The modern way to loop through files or text uses the `FOR` command. In my opinion, `FOR` is the single most powerful command in DOS, and one of the least used.

GOTCHA: The `FOR` command uses a special variable syntax of `%` followed by a single letter, like `%I`. This syntax is slightly different when `FOR` is used in a batch file, as it needs an extra percent symbol, or `%%I`. This is a very common source of errors when writing scripts. Should your for loop exit with invalid syntax, be sure to check that you have the `%%` style variables.

Looping Through Files

```
FOR %I IN (%USERPROFILE%\*) DO @ECHO %I
```

Looping Through Directories

```
FOR /D %I IN (%USERPROFILE%\*) DO @ECHO %I
```

Recursively loop through files in all subfolders of the %TEMP% folder

```
FOR /R "%TEMP%" %I IN (*) DO @ECHO %I
```

Recursively loop through all subfolders in the %TEMP% folder

```
FOR /R "%TEMP%" /D %I IN (*) DO @ECHO %I
```

Functions

Functions are de facto way to reuse code in just about any procedural coding language. While DOS lacks a bona fide function keyword, you can fake it till you make it thanks to labels and the `CALL` keyword.

There are a few gotchas to pay attention to:

1. Your quasi functions need to be defined as labels at the bottom of your script.
2. The main logic of your script must have a `EXIT /B [errorcode]` statement. This keeps your main logic from falling through into your functions.

Defining a function

In this example, we'll implement a poor man's version of the *nix tee utility to write a message to both a file and the stdout stream. We'll use a variable global to the entire script, `%log%` in the function.

```
@ECHO OFF
SETLOCAL

:: script global variables
SET me=%~n0
SET log=%TEMP%\%me%.txt

:: The "main" logic of the script
IF EXIST "%log%" DELETE /Q %log% >NUL

:: do something cool, then log it
CALL :tee "%me%: Hello, world!"

:: force execution to quit at the end of the "main" logic
EXIT /B %ERRORLEVEL%

:: a function to write to a log file and write to stdout
:tee
ECHO %* >> "%log%"
ECHO %*
EXIT /B 0
```

Calling a function

We use the `CALL` keyword to invoke the quasi function labelled `:tee`. We can pass command line arguments just like we're calling another batch file.

We have to remember to `EXIT /B` keyword at the end our function. Sadly, there is no way to return anything other than an exit code.

Return values

The return value of `CALL` will always be the exit code of the function. Like any other invokation of an executable, the caller reads `%ERRORLEVEL%` to get the exit code.

You have to get creative to pass anything other than integer return codes. The function can `ECHO` to stdout, letting the caller decide to handle the output by pipeling the output as the input to another executable, redirecting to a file, or parsing via the `FOR` command.

The caller could also pass data by modifying a global variable, however, I try to avoid this approach.

Parsing Input

Robust parsing of command line input separates a good script from a great script. I'll share some tips on how I parse input.

The Easy Way to read Command Line Arguments

By far the easiest way to parse command line arguments is to read required arguments by ordinal position.

Here we get the full path to a local file passed as the first argument. We write an error message to stderr if the file does not exist and exit our script:

```
SET filepath=%~f1

IF NOT EXIST "%filepath%" (
    ECHO %~n0: file not found - %filepath% >&2
    EXIT /B 1
)
```

Optional parameters

I assign a default value for parameters as such

```
SET filepath=%dp0\default.txt

:: the first parameter is an optional filepath
IF EXIST "%~f1" SET filepath=%~f1
```

Switches

Named Parameters

Variable Number of Arguments

Reading user input

```
:confirm
SET /P "Continue [y/n]>" %confirm%
FINDSTR /I "^(y|n|yes|no)$" > NUL || GOTO: confirm
```

Logging

I use basic logging facilities in my scripts to support troubleshooting both during execution and after execution. I use basic logging as a way to instrument what my scripts are doing at runtime and why. I remember watching a network operations center trying to troubleshoot a legacy batch process where the sysadmins literrally had to try to read the lines of a console window as they trickled by. This technique worked fine for years when the batch machines used dial-up modems for connectivity to remote resources. However, the advent of brooadband meant the batch script executed faster than anyone could read the output. A simple log file would have made troubleshooting work much easier for these sysadmins.

Log function

I really like the basic `tee` implementation I wrote in Part 7 – Functions.

```
@ECHO OFF
SETLOCAL ENABLEEXTENSIONS

:: script global variables
SET me=%~n0
SET log=%TEMP%\%me%.txt

:: The "main" logic of the script
IF EXIST "%log%" DELETE /Q %log% >NUL

:: do something cool, then log it
CALL :tee "%me%: Hello, world!"

:: force execution to quit at the end of the "main" logic
EXIT /B %ERRORLEVEL%

:: a function to write to a log file and write to stdout
:tee
ECHO %* >> "%log%"
ECHO %*
EXIT /B 0
```

This `tee` quasi function enable me to write output to the console as well as a log file. Here I am reusing the same log file path, which is saved in the users `%TEMP%` folder as the name of the batch file with a .txt file extension.

If you need to retain logs for each execution, you could simply parse the %DATE% and %TIME% variables (with the help of command line extensions) to generate a unique filename (or at least unique within 1-second resolution).

```
REM create a log file named [script].YYYYMMDDHHMMSS.txt
SET
log=%TEMP%\%me%.%DATE:~10,4%_%DATE:~4,2%_%DATE:~7,2%%TIME:~0,2%_%TIME:~3,2%
_%TIME:~6,2%.txt
```

Taking a queue from the *nix world, I also like to include a prefix custom output from my own script as `script: some message`. This technique drastically helps to sort who is complaining in the case of an error.

Displaying startup parameters

I also like to display the various runtime conditions for non-interactive scripts, like something that will be run on a build server and redirected to a the build log.

Sadly, I don't know of any DOS tricks (yet) to discrimintate non-interactive sessions from interactive sessions. C# and .Net has the `System.Environment.UserInteractive` property to detect if this situation; *nix has some tricks with tty file descriptors. You could probably hack up a solution by inspecting a custom environmental variable like `%MYSCRIPT_DEBUG%` that defaults to being false.

Advanced Tricks

Boilplate info

I like to include a header on all of scripts that documents the who/what/when/why/how. You can use the `::` comment trick to make this header info more readable:

```
:: Name:     MyScript.cmd
:: Purpose:  Configures the FooBar engine to run from a source control tree
path
:: Author:   stevejansen_github@icloud.com
:: Revision: March 2013 - initial version
::           April 2013 - added support for FooBar v2 switches

@ECHO OFF
SETLOCAL ENABLEEXTENSIONS ENABLEDELAYEDEXPANSION

:: variables
SET me=%~n0


:END
ENDLOCAL
ECHO ON
@EXIT /B 0
```

Conditional commands based on success/failure

The conditional operators `||` and `&&` provide a convenient shorthand method to run a 2nd command based on the succes or failure of a 1st command.

The `&&` syntax is the AND opeartor to invoke a 2nd command when the first command returns a zero (success) exit code.

```
DIR myfile.txt >NUL 2>&1 && TYPE myfile.txt
```

The `||` syntax is an OR operator to invoke a 2nd command when the first command returns a non-zero (failure) exit code.

```
DIR myfile.txt >NUL 2>&1 || CALL :WARNING file not found - myfile.txt
```

We can even combined the techniques. Notice how we use the `()` grouping construct with `&&` to run 2 commands together should the 1st fail.

```
DIR myfile.txt >NUL 2>&1 || (ECHO %me%: WARNING - file not found -
myfile.txt >2 && EXIT /B 1)
```

Getting the full path to the parent directory of the script

```
:: variables
PUSHD "%~dp0" >NUL && SET root=%CD% && POPD >NUL
```

Making a script sleep for N seconds

You can use `PING.EXE` to fake a real *nix style `sleep` command.

```
:: sleep for 2 seconds
PING.EXE -N 2 127.0.0.1 > NUL
```

Supporting “double-click” execution (aka invoking from Windows Explorer)

Test if `%CMDCMDLINE%` is equal to `%COMSPEC%` If they are equal, we can assume that we are running in an interactive session. If not equal, we can inject a PAUSE into the end of the script to show the output. You may also want to change to a valid working directory.

```
@ECHO OFF
SET interactive=0

ECHO %CMDCMDLINE% | FINDSTR /L %COMSPEC% >NUL 2>&1
IF %ERRORLEVEL% == 0 SET interactive=1

ECHO do work

IF "%interactive%"=="0" PAUSE
EXIT /B 0
```