JUDAGO'S SCRIPTS / CONSOLE STUFF

SO LONG.....

BATCH VARIABLES

This page isn't finished and won't be, this site is now abandoned. The environment variables section is incomplete and the other sections may contain errors and/or omissions.

As with any language, variables are integral to windows batch scripting. The aim of this page is to cover the basics as well as many other issues and useful ways to utilise variables in batch. All examples are tested on Windows XP.

Although basic, the variables available are generally enough to accomplish most simple tasks one would undertake in batch. As no memory management needs to be considered by the scripts themselves it is also generally very easy to get started.

The behaviour of variables can be rather strange in contrast to many other languages. All variables can be executed directly as a command, or part there of, and some even execute metacharacters on expansion. So some care must be taken to avoid problems occurring. The behaviour can also be put to good use, sometimes even enabling features that wouldn't otherwise be available.

Variable "types" in batch are divided up by their context rather than content. The purpose of these variables defines their behaviour, some are directly modifiable while others sre only indirectly populated. All variables are strings with automatic sizing and resizing(if modifiable).

The "types" of variables I will refer to in this document are:

FOR VARIABLES

For variables are the variables used by for loops iterating over data. These variables allow the data being iterated over to be passed to commands with each iteration. All for loops in batch require a variable to be defined in syntactical order, even if it is never used. For more information on for loops see the batch for loop page (http://judago.webs.com/batchforloops.htm).

For variables are only available within the "do" part of a for loop, elsewhere they are unavailable. Modifiers are available, without said modifiers these variables typically look like this in scripts:

```
%%a
```

On the command line only one percentage sign is used:

```
%a
```

The naming conventions are fairly simple. For variables can only be a single character and the character used can be *almost* any printable ASCII character. The variables are case sensitive, so %%a is considered different to %%A. "For /f" can also implicitly define variables by using the following characters in ASCII so it's generally easier to use characters that can easily be counted up, so a-z and A-Z are commonly used.

Although valid I would suggest to avoid using numbers as for variables because the look similar to argument variables (%1 vs %%1).

Modifiers

Modifiers are available to for variables, most of these relate to file and directory paths.

- %%~ - Remove Surrounding quotes(true for all modifiers).
- %%~a - Attributes of the file or folder.
- %%~d - Drive letter of the variable.
- %%~f - Fully qualified path name.
- %%~n - File or folder name without the extension or path.
- %%~p - Path only; no file name.
- %%~s - Short file name alias; 8.3 format.
- %%~t - File date and time(local format)
- %%~x - File extension only.
- %%~z - File size, a directory will always return 0.
- %%~$variable: - Commonly used with the path variable, but can be any variable with a delimited list of folder names. It finds and returns the first matching file name contained in the list of folders.

Modifiers can be compounded to form desired output, an example of "filename" combined with "extension":

```
%%~nx
```

The list above shows the modifiers without the for variable character appended, to use them simply add the character number to the end of the modifier. It is also important to note that **all** strip surrounding double quotes, if being used on file or path names you should double quote the variable to avoid issues with spaces or other problem characters.

Parsing Problems

For variables escape *almost* all metacharacters so unwanted execution isn't an issue. They can still be executed intentionally by using the variable *as* a command.

Delayed expansion however creates a major issue, exclamation marks expand variables *inside* for variables. Delayed expansion can be inherited from a calling script or enabled with the following command or a variation thereof.

```
setlocal enabledelayedexpansion
```

With this in place for variables that *should* contain literal exclamation marks contain no exclamation marks, the contents of expanded variables or interpret a leading caret to escape the exclamation mark(removing the caret).

Here is an example of the behavior, to show the implications to "for /f" I have used a text file:

```
@echo off
SetLocal DisableDelayedExpansion
>temp.txt echo !time!^^!
SetLocal EnableDelayedExpansion
for /f "delims=" %%a in (temp.txt) do echo For  : %%a
set /p =Type : < nul
type temp.txt
pause
```

The only general solution is to disable delayed expansion and face the pitfalls of standard expansion. To disable delayed expansion the below command, or a variant, is used.

```
setlocal disabledelayedexpansion
```

ARGUMENT VARIABLES

Arguments variables are the variables that represent arguments passed to a script or subroutine. These variables allow batch scripts to behave like command line utilities without prompting the user for input after the script has been invoked.

The input to these scripts should be specified at invocation in a fashion similar to the example below.

```
yourscript.bat param1 param2 param3 ....

call :label param1 param2 param3 ....
```

The command line is split up on the following characters to populate the variables unless they are either double quoted or escaped with a caret("^"):

- Space - " "
- Tab - "	"
- Equals sign - "="
- Comma - ","
- Semi-colon - ";"

The variables used have a very simple syntax, they are marked with a single percentage sign at the front followed with a number 0 - 9 or an asterisk. Only one of these variables is *always* populated at the start of execution. If not specified these variables expand empty, that is to say the variable marker is removed only. Modifiers are also supported and will be discussed later.

• **%0** - The text used to invoke the script. Usually the name of the script or the label if called as a subroutine. It is important to note this will not always be "script.bat" for scripts, it is just the text the user called the script with, see the modifiers section below to get consistent results from this variable. This variable will always be populated.
• **%1 - %9** - Individual arguments passed to the script.
• **%*** - All of the arguments passed to the script, including double quotes and delimiters.

The below table shows how the variables are broken up.

BATCH ARGUMENT POPULATION										
some.bat	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	...
					%*					
%0										
	%1	%2	%3	%4	%5	%6	%7	%8	%9		

As you may have noticed only the first nine arguments are available to the positional variables(%1 - %9) initially. It is possible to access these arguments using the positional arguments in addition to the full list(%*). The "shift" command can push down the arguments by deleting one at a time.

Using Shift

The shift command deletes and pushes down positional argument variables facilitating their use when more than nine arguments are specified or when the number of arguments is unknown. After shift is used the argument deleted is lost, so a script should capture the value into an environment variable if the argument is to be referenced later. However shift has not effect on the "all arguments" variable of "%*"

Shift can be used as many times as is necessary or desired. it will obviously have no effect after there is no data left to delete. Only one value is deleted per call.

Using the shift command is very simple, it can be used in it's default form, where all arguments are pushed down, including %0, or an optional parameter can be use to specify which value is deleted. Unfortunately only values from 0-8 can be passed to shift, so %9 can't be used as a "push" variable to capture any extra

values, although %8 can.

I would recommend against the default shift if using %0 for self referencing because after the first default shift the original value of %0 is lost entirely. Using a parameter can avoid this problem by instead deleting the original value of "%1".

All that is needed to do a default shift is the following command:

```
shift
```

The same table as above after a single default shift.

```
-------------------------------------------------------------------
|          BATCH ARGUMENT POPULATION AFTER A SINGLE "SHIFT"        |
-------------------------------------------------------------------
| some.bat | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | ... |
-------------------------------------------------------------------
|          |                          %*                          |
|          ----------------------------------------------------------
|          | %0 | %1 | %2 | %3 | %4 | %5 | %6 | %7 | %8 | %9  |     |
-------------------------------------------------------------------
```

Using the parameter to shift is also very easy, all that needs to be passed to shift is a forward slash and the number of the variable you want it to clear and repopulate with higher numbered values. Only values 0 - 8 will be accepted by shift.

Here is an example of shifting the first argument and the result it has on the original table.

```
shift /1
```

```
-------------------------------------------------------------------
|        BATCH ARGUMENT POPULATION AFTER A SINGLE "SHIFT /1"       |
-------------------------------------------------------------------
| some.bat | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | ... |
-------------------------------------------------------------------
|          |                          %*                          |
|    %0    |----------------------------------------------------------
|          |    | %1 | %2 | %3 | %4 | %5 | %5 | %7 | %8 | %9  |     |
-------------------------------------------------------------------
```

And again, but this time using:

```
shift /5
```

```
-------------------------------------------------------------------
|        BATCH ARGUMENT POPULATION AFTER A SINGLE "SHIFT /5"       |
-------------------------------------------------------------------
| some.bat | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | ... |
-------------------------------------------------------------------
|          |                          %*                          |
|    %0    |----------------------------------------------------------
|          | %1 | %2 | %3 | %4 |    | %5 | %6 | %7 | %8 | %9  |     |
```

Modifiers

Modifiers are available to argument variables, most of these relate to file and directory paths. %* can't be used with modifiers due to it's special nature, but %0 - %9 can use these modifiers.

• %~ - Remove Surrounding quotes(true for all modifiers).
• %~a - Attributes of the file or folder.
• %~d - Drive letter of the variable.
• %~f - Fully qualified path name.
• %~n - File or folder name without the extension or path.
• %~p - Path only; no file name.
• %~s - Short file name alias; 8.3 format.
• %~t - File date and time(local format)
• %~x - File extension only.
• %~z - File size, a directory will always return 0.
• %~$variable: - Commonly used with the path variable, but can be any variable with a delimited list of folder names. It finds and returns the first matching file name contained in the list of folders.

The list above shows the modifiers without the number appended, to use them simply add the variable number to the end of the modifier. It is also important to note that **all** strip surrounding double quotes, if being used on file or path names you should double quote the variable to avoid issues with spaces or other problem characters.

One of the most common uses for these modifiers is to gain consistent information from the %0 variable. such as the examples below. I have used "echo" for these examples, but the can be used with any command.

• The script name in the form of "script.bat"
```
echo "%~nx0"
```

• The name of the directory that the script is contained in.
```
echo "%~dp0"
```

• The drive the script is contained on
```
echo "%~d0"
```

Of course there are many more uses for these modifiers, I have just shown a few of the most common that can be very useful.

Parsing problems

There are two main issues surrounding argument variables, code execution and expansion of delayed environment variables. Code execution is by far the most prominent issue because there is no way to prevent it with conventional methods.

The only way I know of to avoid both issues is to use the "rem" method of capturing the arguments into

environment variables.

Code Execution

On expansion of argument variables most metacharacters are interpreted and executed unless they are surrounded by un-escaped double quotes. By "surrounded" I don't just mean between double quotes but inside a section that starts with a double quote and ends with a double quote:

• "escaped"*not_escaped*"escaped"

Argument variables *can* contain double quotes, which proves to be the biggest issue surrounding arguments in batch. There is not way to remove or prevent them.

The characters that are primarily interpreted are below. Double and single forms are accepted. All other metacharacters are interpreted as normal when used in conjunction with ampersands or a single pipes used to form a new command.

• '&' - Ampersand.
• '|' - Pipe.
• '>' - Greater than symbol.
• '<' - Less than symbol.
• '^' - Carets.
• '"' - Double quotes.

Because all of these characters are interpreted and executable no command except "rem" can deal with the possible outcomes.

Executing from an argument is very easy, all that needs to be done is to match any double quotes used inside the script to ensure that the string to be executed falls outside of a double quoted block. Escaping the command line makes this possible.

Take these three commands as examples:

```
echo %1
echo "%1"
echo "%~1"
```

One of these two strings entered as an argument will defeat them:

```
^"^&dir
^&dir
```

Even modifiers aren't immune from these attacks, take these two examples:

```
echo %~f1
echo "%~f1"
```

These strings can beat the modifier:

```
^^^"^&dir^&
^&dir
```

Delayed expansion

If environment variable delayed expansion is enabled by the script("setlocal enabledelayedexpansion) or is inherited from a calling process any exclamation marks contained within the argument are interpreted and either expanded as variables or removed.

If unmatched the marks are simply removed, if they are matched the variable is expanded from the expanded argument variable. All text that is expanded as a variable is lost, either replaced with the variable value or nothing if the variable is undefined.

Take this example:

```
call some.bat yes!time!haha!
```

Some.bat:

```
SetLocal EnableDelayedExpansion
echo %1
```

Output:

```
yes19:35:20.04haha
```

There are two ways of avoiding this issue, the first is to disable delayed expansion, the second is to use the rem method below.

Testing for empty arguments

It is quite common to test if a argument is empty, often in conjunction with shift to check when there is no data left. A typical example is below:

```
if "%~1"=="" whatever
```

If the user enters only two double quotes as the argument then the example above would consider the variable empty.

The "rem" Method

A trick can be used to get arguments into an environment variable using rem but it is a little long winded. I don't want to get into this into too much depth, but the below script gets the %* variable into the variable args. Note that the variable will contain "rem #<the strings>#".

Any argument variable can be captured this way

Check out this Dostips thread (http://www.dostips.com/forum/viewtopic.php?f=3&t=2836) for more information.

```
@echo off
(
    prompt $
    SetLocal DisableExtensions
    echo on
    for %%a in (1) do rem #%*#
    @echo off
    endlocal
    prompt
) > temp.txt
(
    set /p args=
    set /p args=
) < temp.txt


set args

pause
```

ENVIRONMENT VARIABLES

unfinished

Environment variables are the general purpose user defined variables in batch, they can be defined modified and cleared within the script.

Depending on the use, or lack thereof, of the "setlocal" command environment variables can be local to the script or even modify the inherited environment, allowing variables to be passed *back*.

Variable Naming Conventions

The naming conventions for environment variables in batch are fairly loose, almost any text can be variable name, though a few characters should be avoided:

"=" Equals signs are interpreted by the set command so can't be used in variable names.
":" Colons are used for substring and replacement expansion so shouldn't be used in variable names
"~" Tildes mark substring expansion so should also be avoided in variable names.

Some characters need to be escaped with a caret('^') to be usable, including the caret itself. Even double quotes('"' - spaces for clarity) seems to be at least partially usable, but I would recommend avoiding them in normal use.

Generally I would recommend only using A-Z, a-z, 0-9 and the following characters "#$[]{}_-".

Although variables that start with numbers are semi usable, they should also be avoided because standard expansion fails due to the percentage sign followed by a number being interpreted as a argument variable (http://judago.webs.com/batchvariables.htm#836110466). "Set /a" also refuses to set to or evaluate variables that start with number unless they are expanded.

Barring the characters and situations listed above almost any combination of characters can for valid variable names in batch. It's very important to note that spaces are valid so using spaces around the equals sign in set commands actually includes the space(s) in the variable name.

SetLocal

The setlocal command and all of it's variations effect the context that variables defined and modified with the "set" command are available in. The simplest form is issuing the "setlocal" command without any arguments:

```
setlocal
```

After being executed all variables defined or modified before the issuing of a matching "endlocal" command or the termination of the script itself are reverted to their state before "setlocal" was executed. This make all changes temporary and essentially cleans up after the script, memory is freed from variables that are no longer in use and the environment of any calling script is safe from modification. These effects also prohibit using variable to intentionally pass back data to a calling script.

"Setlocal" can "stack", so that one local environment can be created within another. There are limits on the number of stacks allowed, but for general purposes it doesn't need to be considered. If using "setlocal" in an unusual or recursive way then mechanisms should be put into place to avoid repeated calling of "setlocal" without matching "endlocal" commands.

The Set Command

The set command is used to alter environment variables, it can define, modify or clear variables. When used the previous contents of the variable, if any, will be lost with the exception of "setlocal" stacks. The same syntax is used for all operations:

```
set variablename=contents
```

Another variation is available starting from Windows XP:

```
set "variablename=contents"
```

"Variablename" can be any text that forms a valid name for a variable, do not leave any spaces before the equals sign because they will become part of the variable name.

The first equals sign present signals to the set command that the rest of the line is the contents of the variable. Any subsequent equals signs are simply interpreted as part of the contents. If no text is present on the remainder of the line the variable is cleared, spaces and tabs are considered text and will be entered as the new contents of the variable.

Special characters should either be escaped with a caret unless they are contained within a double quoted block of text.

Standard Expansion

What I refer to as "standard expansion" is the oldest and most common form of variable expansion in batch. Variable names are surrounded by percentage signs("%") and executed within or as a command line.

The below is a simple example of expanding a standard expansion environment variable. In this case it will display the contents to screen.

```
echo(%variablename%
```

This form of expansion does have a few pitfalls, the most problematic is often the interpretation of metacharacters. Although it can have it's uses in some circumstances, content obtained via "set /p" (http://judago.webs.com/setpvalidation.htm) or from file can allow the user to manipulate the inputs to execute code via the script itself. Take this example:

```
set test=some text^&pause
echo %test%
```

One might expect the string "some text&pause" to be displayed, instead the string "some text" is displayed and the pause command executed. Since this is being caused by metacharacters being interpreted it is possible to add an extra escape character to avoid the execution, though this approach is next to useless in cases of user input:

```
set test=some text^^^&pause
echo %test%
```

The other major issue with standard expansion is that when used in structures such as parenthesis "( )" marked code blocks or in conjunction with for loops they don't update to reflect any changes made therein. This is because the variable is expanded before the command is, or commands in the case of code blocks are, executed. The command processor no longer updates the variable because after being parsed it doesn't even know it's supposed to be a variable any more. Delayed expansion(explained later in this document) solves this issue, though it does cause it's own issues.

The updating issue can be demonstrated easily:

```
set test=hello
(
    set test=goodbye
    echo %test%
)
```

```
echo %test%
```

Logically one might think that the output should be "goodbye", followed by another "goodbye". The strange effect the expansion rules create results in the output of "hello" then "goodbye". This quirk can even be used to swap the contents of two variables without the need for a third for temporary storage:

```
set first=goodbye
set second=hello
(
    set first=%second%
    set second=%first%
)
echo %first%
echo %second%
pause
```

Without the strange behaviour this small script should output "hello" and "hello", but without updating the variables contents are successfully swapped.

Predefined Variables

There are usually some general predefined variables available to batch scripts these come in two different form, dynamic and static environment variables. Static variables are held in the windows registry and are usually a mix of user and system variables.

Cmd will allow a script to change these variables locally within a script or cmd.exe process, but will restore them to every new process. Unintentionally using these variable names can lead to unexpected problems so it is best to avoid them in normal use

Dynamic

Dynamic variables appear to be undefined to the set command and expand to reflect the current state of the information requested. They are expanded in the same fasion as user defined variable, surrounded by percentage signs "%" or exclamation marks "!" depending on context. Different windows versions have more or less dynamic variables available, these are the ones available in Windows XP:

```
CD             - The current directory.
DATE           - The current date in local format.
TIME           - The time when expanded, usually military format
RANDOM         - A pseudo-random decimal number between 0 and 32767
ERRORLEVEL     - The exit code returned by the last program or command.
CMDEXTVERSION - The cmd extension version number.
CMDCMDLINE     - The text that invoked the original cmd process.
```

The undocumented internal dynamic variables I know of are as follows. Notice that they start with "=" so set can't alter them.

```
=EXITCODE      - The same as ERRORLEVEL in hex format.
=EXITCODEASCII - If errorlevel is between 32 and 126 then this will expand to
equivalent ASCII character.
=X:            - The current directory of drive "x", can be any drive.
```

If the set command is used to define dynamic variables they become ordinary user defined variables until cleared.

Static

Static variables are stored in the registry and loaded by cmd.exe at start up. They can either be part of the user or system environment.

Some are for the convenience of the user and other programs, while others provide information such as directories to find binary programs in to the host cmd.exe process. I'm not going to list them all, but I will list a few that shouldn't be used as user variables.

```
ComSpec      - The location of the command processor(usually "cmd.exe").
Path         - A list of directories to search for programs.
PATHEXT      - The file extension to be used with "path".
ProgramFiles - The program files directory.
SystemDrive  - The drive that windows runs from.
SystemRoot   - The windows directory.
windir       - Depreciated; Same as "SystemRoot".
TEMP         - Temporary file directory.
TMP          - Same as TEMP.
```

There are many more available but I listed these because reusing their names by accident can cause many problems. To see all variable in use on your system open up a command prompt and type in "set".

Substitutive Expansion

Variable expansion offers a couple of useful extra tools: Substrings and Text Replacement. Both are only available if Cmd Extensions are enabled.

As is to be expected substitutive expansion only works on defined variables. Undefined variables result in cmd.exe trying to output the variable name with the text denoting the special expansion method. For this reason it is important to check the variable is defined before use in uncertain circumstances.

Substrings

Substrings use a fairly simple syntax and support both absolute and relative positions.

Substrings are marked by using a colon(":") followed by a tilde("~") and one or two numbers, if two numbers are used they must be comma(",") separated.

Text replacement

Setlocal DisableExtensions

SetLocal EnableDelayedExpansion

Delayed Expansion
