Windows NT Shell **Script**ing - Chapter 5: A **Script**ing Toolkit

New Riders, 13 Feb 2001

4.25 (8 votes) Rate:

An introduction to the native Windows NT **script**ing language, including shell command syntax

Title	Windows NT Shell **Script**ing
Authors	Tim Hill
Publisher	New Riders
Published	Apr 1998
ISBN	1578700477
Price	US 32.00
Pages	380

Chapter 5 - A **Script**ing Toolkit

- Building **script**s - Learn the tools and techniques needed to construct robust **script**s.
- Standard **script** skeleton - This section provides a complete **script** skeleton that can be used as the starting point for any new **script** project.
- Standard library skeleton - Complete libraries of **script** procedures can be constructed using the library skeleton described in this section.
- An example library - This sample library provides many useful procedures that are used by the other sample **script**s in this book.

Building **Script**s

This chapter provides guidelines for using tools to build **script**s, and also provides two complete **script** skeletons:templates that can be used as starting points for customized **script**s. In addition, the **_MTPLIB.BAT script** library source code is presented and described in detail. This library contains many useful procedures that can be accessed directly from custom **script**s. The sample **script**s of Chapter 6 and 7 also make extensive use of the **_MTPLIB.BAT script** library.

Because **script**s are text files, the only real tool needed when developing a **script** is a good text file editor. However, certain special **script** requirements place two specific constraints on this editor:

- First, some **script**s require trailing spaces at the end of a line (for example, the **PJCOUNT.BAT script** of Chapter 6). Therefore, check that the editor does not automatically strip trailing white space from lines.
- Second, a FOR command bug sometimes requires literal tab characters in the **script** (for example, the **_MTPLIB.BAT script** of this chapter). Therefore, make sure that any editor you use does not convert tab characters into spaces (or vice versa).

The Windows Notepad editor correctly handles both trailing spaces and literal tab characters.

The only other (and most important) tool needed to develop a **script** is good programming discipline. **Script**s are often seen as "quick and dirty" solutions to one-off problems. Typically, however, the quick and dirty **script** takes on a life of its own and is modified and enhanced until it has far outgrown its humble origins. More robust and manageable **script**s result when each **script**, however small, is treated as a simple programming project. The skeleton **script**s in this chapter are good starting points for any **script** and encourage a structured approach to **script** creation.

The command shell does not offer any built-in **script** debug facilities, other than the capability to enable and disable **script** command echo (using the **ECHO** command). The following techniques may prove helpful when debugging **script**s:

- Use the standard preamble shown in the skeleton **script**s in this chapter, and disable/enable **script** tracing using the **ECHO** variable.
- Use the **TRACE** variable shown in the skeleton **script**s in this chapter.
- Add **ECHO** commands when developing a **script** to show intermediate variable values, control flow, and exit codes (via **%ERRORLEVEL%**). These can be removed when development is complete.
- Add **PAUSE** commands before a critical part of a **script**, so that the **script** can be stopped if something appears to be wrong. These can be removed when development is complete.
- Add **ECHO** commands to preview complex or "dangerous" commands (such as a command which deletes lots of files) before they are executed. Add a **PAUSE** command after the **ECHO** command but before the actual execution of the command.

Most of these techniques are highlighted in the sample **script** presented in this chapter.

Standard **Script** Skeleton

Figure 5.1 shows the **SKELETON.BAT script**. This **script** does not actually do anything, but instead provides a complete template for **script** development.

Follow these steps to create a new **script** based on the **SKELETON.BAT** template:

1. Copy **SKELETON.BAT** to a new **script** file.
2. Add **ECHO** commands to the **HELP** procedure to display brief on-line help information.
3. Add the **script** program logic to the **MAIN** procedure. If necessary, call any external library **INIT** procedures following the call to the **_MTPLIB.BAT INIT** procedure.
4. Create additional procedures used by **MAIN** following the end of the **MAIN** procedure and before the **DOSEXIT** label.

Although **SKELETON.BAT** does not do anything, it does bring together many of the structural suggestions mentioned in Part I of this book into a complete, ready-to-use **script**. After some initial setup, the **script** calls a procedure named **MAIN** at line 15, and passes to this procedure all of the command line arguments. The **MAIN** procedure should contain the program logic of the **script**. When the **MAIN** procedure exits, the entire **script** exits. Template code surrounding the **MAIN** procedure handles all of the logic needed to make the **script** a "good citizen"-a local scope for variables is created, and the current state is saved.

Before the **MAIN** procedure is called, the template logic checks to see if the first argument is either **/?** or **/HELP** (lines 13 and 14). In this case, the **HELP** procedure is called instead of **MAIN**. Typically, this procedure displays a short help message describing the use of the **script**.

The first two lines of **SKELETON.BAT** (shown in Figure 5.1) provide the command echo management discussed in Part I. When the **script** is executed, if the variable **ECHO** has the value **ON**, then **script** command echo is enabled. If the variable **ECHO** has the value **OFF** or is not defined, then **script** command echo is disabled.

The third line checks the operating system type. If the **script** is run on an OS other than Windows NT, the **script** jumps immediately to the **DOSEXIT** label. This label, located at the very end of the **script**, simply displays a warning message and then ends **script** execution by "falling off" the end of the file. Thus, if a **script** based on this skeleton is run on an OS other than Windows NT, it simply displays:

Hide Copy Code

```
This script requires Windows NT
```

One implication of this code is that the first three lines of the **script** and all the lines following the **DOSEXIT** label must be syntax-compatible with MS-DOS, Windows 3.1, Windows 95, and OS/2. This is why, for example, there is no colon character preceding the **DOSEXIT** label on line 3. Once past that line, however, all of the syntax enhancements provided by Windows NT can be safely used.

Figure 5.1. The **SKELETON.BAT script**

Following the initial setup lines is the main **script** body. In outline, this is constructed as follows:

Hide Copy Code

```
1. setlocal & pushd & set RET=

2. .

3. .

4. popd & endlocal & set RET=%RET%

5. goto :EOF
```

The first line creates a local scope for variables as well as the current drive and directory (via the **SETLOCAL** and **PUSHD** commands). After the **script** body, a corresponding set of commands, **ENDLOCAL** and **POPD**, close the local scope. This makes the **script** "well behaved" and preserves the current drive and directory, as well as all environment variables. It also means that code within the **script** is free to alter any variable, as any changes made are automatically restored when the **script** completes.

The **RET** variable is also initialized by the **script** body code, and the variable tunneling technique described in Chapter 3 is used to pass the **RET** value back from the **script** (**SET RET=%RET%** on the same line as **ENDLOCAL**). Thus, after the **script** executes, the only change in the environment will be the **RET** value. This means that the **SKELETON.BAT script** can be used to develop complete **script** procedures that can be called from other **script**s and return results via the **RET** variable.

The main **script** body between the local scope "brackets" described previously is as follows:

Hide Copy Code

```
1. set SCRIPTNAME=%~n0

2. set SCRIPTPATH=%~f0

3. if "%DEBUG%"=="1" (set TRACE=echo) else (set TRACE=rem)

4. call _mtplib :INIT %SCRIPTPATH%

5. if /i {%1}=={/help} (call :HELP %2) & (goto :HELPEXIT)

6. if /i {%1}=={/?} (call :HELP %2) & (goto :HELPEXIT)

7. call :MAIN %*

8. :HELPEXIT
```

The first two lines set two standard variables: **SCRIPTNAME** and **SCRIPTPATH**. The **SCRIPTNAME** variable contains the name of the **script** (**SKELETON**, in this case). Uses for the **script** name include constructing data file names (see the **ANIMAL.BAT script** of Chapter 7), and choosing a Registry key name (see the **REPL.BAT script** of Chapter 7). The second variable, **SCRIPTPATH**, contains the full path to the **script** (even if the full path was not entered on the command line). These two variables

are set because the program logic in the **MAIN** procedure does not have direct access to the **script** name. (The %0 parameter within the **MAIN** procedure has the value **:MAIN**, regardless of the **script** name.)

The third line in the main **script** body provides trace facilities. The variable **TRACE** is either set to echo or rem depending upon the value of the **DEBUG** variable. If the **DEBUG** variable has the value 1 before the **script** is executed, then **TRACE** is defined as echo. If the **DEBUG** variables has another value or is not defined, then **TRACE** is defined as rem.

The purpose of the **TRACE** variable is to provide automatic trace functionality within a **script**. Consider this **script** statement:

Hide Copy Code

```
%TRACE% Computing total size...
```

If the **DEBUG** variable is not1, the **TRACE** variable is rem, and the statement expands to:

Hide Copy Code

```
rem Computing total size...
```

Obviously, the **REM** command does nothing. If, however, the **DEBUG** variable is 1, the **TRACE** variable is echo, and the statement expands to:

Hide Copy Code

```
echo Computing total size...
```

This displays the text in the console window. Thus, if **DEBUG** is 1, all **%TRACE%** prefixed commands display trace information. If **DEBUG** is not 1, no output is displayed. This allows **script** trace commands to be embedded throughout a **script** and enabled or disabled just by changing the **DEBUG** variable before running the **script**. All the sample **script**s use this technique to display the names of called procedures. Each procedure starts with this line:

Hide Copy Code

```
if defined TRACE %TRACE% [proc %0 %*]
```

This displays the name of the procedure (%0) and all the procedure arguments. To ensure robustness, the line first checks that the variable **TRACE** is defined (using the **IFDEFINED** command) before executing the trace statement.

Following the **TRACE** setup command are one or more library initialization procedure calls. Only one **script** library, **_MTPLIB.BAT**, is initialized by the sample skeleton **script**. If the **script** uses other libraries, the **INIT** procedure for each library should be called here. The use of **script** libraries and the **INIT** procedure are discussed in Chapter 4.

Finally, when all libraries have been initialized, the main **script** logic is invoked. Depending upon the first argument, either the **HELP** or the **MAIN** procedure is called. The **MAIN** procedure is passed all **script** arguments. The **HELP** procedure is passed only the second argument, allowing a help context to be requested by passing an additional argument following the **/?** or /**HELP** switch.

Standard Library Skeleton

Figure 5.2 shows the **_LIBSKEL.BAT** library **script**. This library has no function other than to provide a complete template for **script** library development.

Follow these steps to create a new library based on the **_LIBSKEL.BAT** template:

1. Copy **_LIBSKEL.BAT** to a new **script** file.
2. Change the library name and version number in the first **ECHO** command after the preamble.
3. Add any necessary library setup logic to the **INIT** procedure.
4. Add all desired library procedures following the **INIT** procedure.
5. Place the **script** library in a directory that is part of the system (not the user) **PATH**. This ensures that the library is available to all **script**s, even if the **script** is run by the **AT** command.
6. Document the library! **Script** logic is not easy to follow, and can be quite obscure even to the **script** author six months after it was written.

Like **SKELETON.BAT**, the **_LIBSKEL.BAT** library **script** does not do anything, but it does implement the suggestions for **script** libraries described in Chapter 4. The **script** contains a procedure dispatcher (lines 13 to 16), which automatically vectors to the correct procedure in the library. It also contains an empty **INIT** procedure, where library initialization code can be added. By convention, any **script** that makes use of the procedures in a **script** library should call the **INIT** procedure in that library first. This allows the library to initialize any resources it needs.

Creating procedures for **script** libraries is identical to creating procedures for regular **script**s. However, as any other **script** can call the procedures in a **script** library, some care must be taken to ensure that library procedures are well behaved. Some things to watch out for include:

- Avoid changing state information, such as the current drive or directory (unless that is a desired function of the library procedure). If it is necessary to alter these, use **PUSHD** and **POPD** to preserve the caller state.
- Avoid changing global variables. Create a local scope using **SETLOCAL** and **ENDLOCAL** if extensive use is made of variables. If only one or two variables are needed, use names prefixed by the name of the library (such as **_MYLIB_T1**), so that the names do not collide with those used in the caller **script**.
- Pass return results back to the calling procedure in the **RET** variable. Pass additional results in additional **RET** variables (such as **RETX**, **RETV**). Document the return results carefully. It is good practice to initialize the **RET** variable(s) to a default return value immediately upon entry to a procedure.
- Pass arguments to the procedures as parameters. Avoid passing arguments in variables unless there is a special need (for example, if the argument contains a large amount of text). If arguments are passed in variables, document this carefully. Also document any standard variables used by the procedure (such as **TEMP** or **PATH**).
- Define label names carefully. Labels are global within a **script** file, so using a label such as **:LOOP** isn't particularly friendly, as it will quite likely collide with another label of the same name elsewhere in the library. Within a procedure, prefix all label names with the procedure name. (This advice applies equally to procedures in a regular **script**.)

The **_LIBSKEL.BAT** **script** begins with the same preamble lines as the **SKELETON.BAT** **script**, and ends with the same **DOSEXIT** label and code. The first line following the preamble checks to see if any arguments are present. If not, the **script** displays the library name and version information and then exits. If one or more arguments are present, the **script** falls through into the dispatch code.

As explained in Chapter 4, a library procedure is called using an indirect **CALL** command, which follows the **CALL** command with the library name, procedure name, and then procedure arguments. For example:

Hide Copy Code

```
call _libskel :MYPROC arg1 arg2
```

Here, the procedure **MYPROC** in the **_LIBSKEL.BAT** library is called with arguments arg1 and arg2. The CALL command calls the **_LIBSKEL.BAT** library, passing **:MYPROC**, arg1 and arg2 as three arguments. Eventually, the dispatch code in **_LIBSKEL.BAT** is reached. This code is:

Hide Copy Code

```
1. set _PROC=%1
2. shift /1
3. goto %_PROC%
```

The first line of the dispatch code saves the first argument (the procedure name, in this case **:MYPROC**) in the **_PROC** variable. The second line then shifts the arguments, discarding the procedure name and moving all other arguments down one place in the argument list. In the example above, this moves **arg1** to **%1** and **arg2** to **%2**. This is where these arguments are expected by the procedure. Finally, the third line jumps to the label specified by the **_PROC** variable (that is, the procedure name specified in the original **CALL** command). The result of this processing is that the procedure specified in the original **CALL** command is called and passed the arguments specified.

The **_LIBSKEL.BAT script** provides one sample procedure, **:CHECKX86**, which sets the **RET** variable to 0 or 1 depending upon whether the **script** is run on an Intel x86 platform or not. This sample procedure also shows the use of the **TRACE** variable described previously.

Figure 5.2. The **_LIBSKEL.BAT script**.

An Example Library

The **_MTPLIB.BAT script** library shown in Figure 5.3 is a complete sample library based upon the **_LIBSKEL.BAT script** library described in the previous section. **_MTPLIB.BAT** contains a number of useful procedures that are used by many of the sample **script**s in Chapters 6 and 7, and can also be used by other **script**s as desired. To use this library, place it in a directory that is on the system **PATH**. The **SKELETON.BAT script** already contains code to call the **INIT** procedure of this library.

The **_MTPLIB.BAT** library contains procedures to assist in the following tasks:

- Deleting multiple variables (by variable prefix).
- Parsing a command line for switches and positional arguments.
- Saving and restoring variables to/from the Registry.
- Generating pseudo-random numbers.
- Resolving recursive (nested and indirect) variable references.
- Reading a line of user input to a variable.
- Synchronizing **script**s using lock files.
- Generating unique temporary file names.

Tip

The **REGGETM** and **REGGETU** procedures in the **_MTPLIB.BAT** library must be entered carefully. The output of the **REG** command is tab-delimited, and (unfortunately) a bug in the **FOR** command means that the **delims=** value for the **FOR** command delimiters must be explicitly set to a tab character. To do this, enter the **delims=** text, and immediately follow the **=** sign with a literal Tab key. The Notepad editor correctly enters literal tab characters into the **script** file.

Should **_MTPLIB.BAT** ever be edited by an editor that converts tabs to spaces, the functionality of the **REGGETM** and **REGGETU** procedures will be damaged.

Figure 5.3. The _MTPLIB.BAT **script** library

Each procedure available in the **_MTPLIB.BAT** library is described in the following sections. The procedures are described in the same order that they appear in the **script**.

VARDEL

Deletes a set of variables by prefix.

Syntax

Hide Copy Code

```
CALL _MTPLIB :VARDEL prefix
```

Arguments

prefix Prefix of variables to be deleted.

Description

The **VARDEL** procedure deletes one or more environment variables sharing a common prefix. For example, specifying a prefix of **JOB_** deletes all variables that begin with **JOB_**. One common use of **VARDEL** is deleting all members of an array by specifying the array name as the **prefix**.

Implementation

VARDEL executes a **SET** command (line 35) with the specified prefix without an equal sign following it. This command displays all variables that match this prefix (in the form **name=value**). The output of this **SET** command is captured by a **FOR** command and parsed. Each line is parsed for the first token, using = as the delimiter, which yields the variable name. The **FOR** command then executes the command **SET name=** for each parsed name, thus deleting each variable which matches the prefix.

The command error output of the first **SET** command is redirected to the **NULL** device (in other words, discarded). This takes care of the situation in which no variables match the specified prefix (in this case, the **SET** command displays an error message, which is captured and discarded by this redirection). Notice the use of the escape character (**^**) before the redirection symbol in the **SET** command. This ensures that the redirection is processed when the **SET** command is executed, not when the **FOR** command is executed.

PARSECMDLINE

Parses a command line(s).

Syntax

Hide Copy Code

```
CALL _MTPLIB :PARSECMDLINE [append]
```

Arguments

CMDLINE Contains the text of the command line to parse.

append 0 (the default) to perform a new parse, 1 to append to existing parse.

Returns

RET Total number of arguments parsed.

CMDARGCOUNT Count of arguments in **CMDARG_n** array.

CMDARG_n Command arguments (**CMDARG_1** contains first argument).

CMDSWCOUNT Count of switches in **CMDSW_n** array.

CMDSW_n Command switches (**CMDSW_1** contains the first switch).

Description

The **PARSECMDLINE** procedure parses a command line, separating the arguments into switches and positional arguments. In most Windows NT commands, a switch is a command argument preceded by a **/** or **-** character. Switches can generally appear anywhere on a command line, intermixed with regular (positional) arguments. Switches can also contain optional values, which follow the switch and are separated by a colon. For example,

Hide Copy Code

```
/SRC:\\transit\files
```

is the **/SRC** switch with the value **\\transit\files**.

The **PARSECMDLINE** procedure parses a command line and separates the arguments into two distinct arrays of arguments: switches and positional arguments. Once the command line has been parsed, the parsed arguments can be accessed directly in the arrays or indirectly via the **GETARG**, **GETSWITCH**, and **FINDSWITCH** procedures.

The command line to parse is passed in the **CMDLINE** variable, not as an argument. Typically, this variable is initialized with the **%*** parameter (which contains the entire command line). However, it can also be initialized from other variables or from the contents of a data file. This allows **PARSECMDLINE** to parse command text from many different sources.

PARSECMDLINE takes one optional argument. If this argument is 0 (the default), a new parse operation is performed, and any previous parse results are discarded. If this argument is 1, the new command line information is appended to any existing parse results. This allows multiple command lines to be merged together and processed as a single line.

Positional (that is, non-switch) arguments parsed by **PARSECMDLINE** are placed in the **CMDARG_n** array. The first argument is placed in **CMDARG_1**. A count of all positional arguments is stored in **CMDARGCOUNT**. Double quotes are stripped from arguments before they are stored in the array. (See Chapter 3 for a discussion of double quotes.) The arguments are stored in the array in the same order as they occurred in the command line.

Switch arguments parsed by **PARSECMDLINE** are placed in the **CMDSW_n** array. The first switch is placed in **CMDSW_1**. A count of all switches is stored in **CMDSWCOUNT**. Double quotes are stripped from switches before they are stored in the array. The switches are stored in the array in the same order as they occurred in the command line.

Implementation

After performing some variable initialization, **PARSECMDLINE** calls **PARSECMDLINE1**, passing it the contents of the **CMDLINE** variable as an argument. The normal command shell parsing mechanism thus performs most of the work of splitting the contents of the **CMDLINE** variable into individual arguments.

PARSECMDLINE1 then runs a loop to process each parameter (lines 61 to 75). After parameter **%1** is processed, the **SHIFT** command (line 66) shifts the parameters down, and the loop repeats until parameter **%1** is empty (line 62), and hence, all parameters have been processed.

Parameter processing begins by removing double quotes via string substitution (lines 63 and 64). The first character of the parameter is then compared to the switch characters **/** and **-** (lines 67 and 68). If a match occurs, the parameter is added to the switch array (lines 72 to 75). Otherwise, the parameter is added to the positional argument array (lines 69 to 71).

GETARG

Obtains a positional argument from a parsed command line, by index.

Syntax

Hide Copy Code

```
CALL _MTPLIB :GETARG index
```

Arguments

index Index of argument to fetch. The first argument is numbered 1.

Returns

RET Argument text, or empty if index is greater than CMDARGCOUNT.

Description

The **GETARG** procedure recovers a single positional argument from a command line that has been parsed by the **PARSECMDLINE** procedure. The single argument specifies the index of the argument to return, which must be between 1 and **GETARGCOUNT**. The argument is returned in the **RET** variable. The returned argument will not contain double quotes.

Implementation

After performing some basic error checking, the core of the **GETARG** procedure is a **SET** command (line 92):

Hide Copy Code

```
set RET=%%CMDARG_%1%%
```

If the index specified is 3, for example, the **RET** variable contains **%CMDARG_3%**. This is the name of the variable containing the desired value. **GETARG** then calls the **RESOLVE** procedure (described later in this chapter) to convert the name of the variable into its value, which is the desired argument.

GETSWITCH

Obtains a switch argument from a parsed command line, by index.

Syntax

Hide Copy Code

```
CALL _MTPLIB :GETSWITCH index
```

Arguments

index Index of switch to fetch. The first switch is numbered 1.

Returns

RET Switch text (name), or empty if index is greater than **CMDSWCOUNT**.

RETV Switch value (text following colon) or empty if no value.

Description

The **GETSWITCH** procedure recovers a single switch and its value (if any) from a command line that has been parsed by the **PARSECMDLINE** procedure. The single argument, which must be between 1 and **GETSWCOUNT**, specifies the index of the switch to return.

The argument name is returned in the **RET** variable (including the **/** or **-** character) and the argument value (if any) in the **RETV** variable. The returned switch and value will not contain double quotes.

Implementation

GETSWITCH is similar in implementation to **GETARG**, except that it contains an additional step after the switch text has been recovered into the **RET** variable. This step uses a **FOR** command (line 113) to parse the text into the switch name and value. The name includes all text up to the first colon; the value includes all text after the first colon. Notice that the tokens value in the **FOR** command is tokens=1* and nottokens=1,2. This allows the switch value to contain any text, including additional colon characters.

FINDSWITCH

Finds a switch argument from a parsed command line by name.

Syntax

Hide Copy Code

```
CALL _MTPLIB :FINDSWITCH name [start-index]
```

Arguments

name Name of switch to locate (with leading / or - character).

start-index Starting index for search (the default is 1).

Returns

RET Index of switch, or 0 if not found.

RETV Switch value (text following colon), or empty if no value or not found.

Description

The **FINDSWITCH** procedure searches for a switch value by name in a command line that has been parsed by the **PARSECMDLINE** procedure. The first argument specifies the name of the switch to find (including the leading **/** or **-** character). The name is not case-sensitive. The search starts at the first switch unless start-index is present, in which case the search starts at the index specified (start-index must be less than or equal to **CMDSWCOUNT**). Using start-index allows multiple switches of the same name to be sequentially processed.

The switch index is returned in the **RET** variable. The value 0 is returned if the switch cannot be located. The value of the switch (if any) is returned in the **RETV** variable.

Implementation

FINDSWITCH is implemented as a simple loop (lines 129 to 134) that calls **GETSWITCH** for each index until a matching switch name is found or the end of the switch array is reached. The IF command (line 132), which compares the switch names, uses the **/I** switch to perform a case-insensitive comparison.

REGSETM and REGSETU

Sets Registry values from variables.

Syntax

Hide Copy Code

```
1. CALL _MTPLIB :REGSETM contextprefix

2. CALL _MTPLIB :REGSETU contextprefix
```

Arguments

context Registry context (location). Typically the **script** name.

prefix Prefix of variables to be saved in Registry.

Description

The **REGSETM** and **REGSETU** procedures save one or more variables in the Registry. These procedures thus provide a way for a **script** to maintain persistent state information, even across system restarts. The **REGSETM** procedure stores the variables in the **HKEY_LOCAL_MACHINE** portion of the Registry, while the **REGSETU** procedure stores the variables in the **HKEY_CURRENT_USER** portion of the Registry.

In order to distinguish one **script**'s state from another, each **script** must specify a context for the variables. This same context is used when restoring the variables via the **REGETM** and **REGGETU** procedures. Typically, the **script** uses the **script** name as the context value. If the **script** is based on the **SKELETON.BAT** template, the **script** name is available in the **SCRIPTNAME** variable.

The variables to save are specified by the prefix argument. Like the **VARDEL** procedure, the **REGSETU/REGSETM** procedures save all variables that have a prefix that matches prefix.

Each variable is stored as a **REG_SZ** registry value. The values are placed in the key **HKEY_LOCAL_MACHINE\Software\MTPScriptContexts\context** for **REGSETM** and **HKEY_CURRENT_USER\Software\MTPScriptContexts\context** for **REGSETU**.

Implementation

Both **REGSETM** and **REGSETU** are similar to the **VARDEL** procedure. They use a **SET** command in a **FOR** command to extract a list of all variables which match the specified prefix. The procedures then call the procedure **REGSET1** for each matching variable, passing it the root key name (**HKLM** or **HKCU**), the name of the variable, and the variable value (in double quotes, as it can contain spaces). **REGSET1** then executes a **REG** command (lines 157 and 158) to store the passed value into the appropriate registry key. Both a **REG ADD** and a **REG UPDATE** command are executed, as the procedure has no way of knowing if the variable already exists in the Registry.

The **REG** command has one undocumented peculiarity. In the value string passed, the backslash character is treated as an escape character to allow special values to be passed. Therefore, before the variable values are stored in the Registry, each backslash character is converted to a double backslash.

REGGETM and **REGGETU**

Gets variables from the Registry.

Syntax

Hide Copy Code

```
1. CALL _MTPLIB :REGGETM context [variable]

2. CALL _MTPLIB :REGGETU context [variable]
```

Arguments

context Registry context (location). Typically the **script** name.

variable Name of variable to restore (optional; default restores entire context).

Returns

RET Value of last (or only) variable loaded.

De**script**ion

The **REGGETM** and **REGGETU** procedures reverse the actions of the **REGSETM** and **REGSETU** procedures, and restore one or more variables from the Registry. The **REGGETM** procedure restores the variables from the **HKEY_LOCAL_MACHINE** portion of the Registry, while the **REGGETU** procedure restores the variables from the **HKEY_CURRENT_USER** portion of the Registry.

The context specifies the context from which to restore the variables. It should be the same context name that was used to store the variables. Typically, the **script** uses the **script** name as the context value. If the **script** is based on the **SKELETON.BAT** template, the **script** name is available in the **SCRIPTNAME** variable.

To restore an individual variable, specify it using the variable argument. To restore all the variables from a context, omit the variable argument. When restoring a complete context, only variables that are found in the Registry are altered. Therefore, before calling **REGGETM** or **REGGETU**, preset all variables with default values.

Implementation

These procedures use a **REG QUERY** command (lines 177 and 181) to recover the contents of the variables. The output of this command is somewhat verbose, but by filtering the output by a **FIND** command, only those lines that actually define a variable are isolated. The entire **REG/FIND** command is captured by a **FOR** command and parsed to extract the variable name and value, which is then used in a **SET** command to restore the variable.

The procedures must also undo the doubled backslash processing performed by the **REGSETM** and **REGSETU** procedures. Double quotes are also removed from the variable values.

REGDELM and REGDELU

Delete saved Registry variables or entire context.

Syntax

Hide Copy Code

```
CALL _MTPLIB :REGDELM context [variable]

CALL _MTPLIB :REGDELU context [variable]
```

Arguments

context Registry context (location). Typically the **script** name.

variable Name of variable to delete (optional; the default deletes the entire context).

Description

The **REGDELM** and **REGDELU** procedures delete one or more variables from the Registry. The **REGDELM** procedure deletes the variables from the **HKEY_LOCAL_MACHINE** portion of the Registry, while the **REGDELU** procedure deletes the variables from the **HKEY_CURRENT_USER** portion of the Registry.

The context specifies the context from which to delete the variables. It should be the same context name that was used to store the variables. Typically, the **script** uses the **script** name as the context value. If the **script** is based on the **SKELETON.BAT** template, the **script** name is available in the **SCRIPTNAME** variable.

To delete an individual variable, specify it using the variable argument. To delete all the variables from a context, omit the variable argument.

Implementation

The core of these procedures is a **REG DELETE** command (lines 202 and 209). Since this command prompts for confirmation before proceeding with the delete operation, a temporary file is created containing the needed response, and the console input of the **REG** command is redirected to this file. Early versions of the **REG** command supported the **/F** switch to skip this confirmation, but newer versions do not, so these procedures avoid the use of this switch.

SRAND

Seeds the random number generator.

Syntax

CALL _MTPLIB :SRAND value

Arguments

value New seed value.

Description

The **SRAND** procedure can be used to re-seed the random number generator used by the **RAND** procedure. This has the effect of changing the sequence of pseudo-random numbers generated by **RAND**.

Implementation

The **RAND** procedure keeps the current seed in the **_MTPLIB_NEXTRAND** variable. Therefore, all this procedure does is assign a new value to this variable.

RAND

Generates a pseudo-random number.

Syntax

Hide Copy Code

```
CALL _MTPLIB :RAND
```

Returns

RET Next pseudo-random number.

Description

The RAND procedure returns the next pseudo-random number in the pseudo-random sequence used by the generator. Each call to RAND returns a new value in the RET variable. These values are between 0 and 32767, inclusive.

Implementation

The algorithm used is based upon that recommended by ANSI for the standard C language library. The algorithm is fast and simple, and generates numbers with uniform distribution. It should not, however, be used for serious statistical analysis.

The RAND procedure stores the current seed in the **_MTPLIB_NEXTRAND** variable, which is updated each time **RAND** is called. If **_MTPLIB_NEXTRAND** is not defined, the procedure defines it with an initial value of 1 (line 233). This initialization could be performed by the **INIT** procedure of the **_MTPLIB** library, but placing the initialization within the procedure improves the localization and makes the procedure self-contained.

RESOLVE

Resolves recursive and nested variable substitution.

Syntax

Hide Copy Code

```
CALL _MTPLIB :RESOLVE
```

Arguments

RET Text containing variables to be resolved.

Returns

RET Text with all variable references resolved.

Description

The **RESOLVE** procedure implements the technique described in Chapter 3 to handle recursive and nested variable substitutions. The command shell scans all **script** commands for variable references to expand. However, if the value of a variable that is expanded contains additional variables to expand, these are not expanded. The **RESOLVE** procedure overcomes this limitation by providing a fully recursive variable expansion facility.

Before calling **RESOLVE**, the **RET** variable should contain the text to be processed. This text can contain any number of variable references (each surrounded by percent signs, as usual). In addition, the value of any of these variables can also contain variable references to resolve and so on to any nesting depth. After **RESOLVE** has been called, the **RET** variable contains the original text, with all variable substitutions fully resolved.

RESOLVE is particularly useful when variable information must be accessed indirectly (for example, when arrays of data are accessed), or a variable name must be composed from fragments and then resolved. It is used extensively in the **_MTPLIB** library and throughout the sample **script**s.

Implementation

The key to the **RESOLVE** procedure is to execute the command ECHO %RET% and capture the output of this command back into the **RET** variable (line 251). In the process of executing the **ECHO** command, any variable substitutions within the text of the **RET** variable are resolved. The new text (with the variable substitutions made) is then stored back into the **RET** variable, thus resolving one layer of variable indirection.

The **FOR** command is used to capture the output of the **ECHO** command. No actual parsing is done, as the entire contents of the **ECHO** command output are assigned back to the **RET** variable. To allow any depth of nested substitution to occur, the **FOR/ECHO** commands are executed in a loop (lines 248 to 252). The loop terminates when the contents of the **RET** variable before and after the **FOR/ECHO** commands are unchanged. This indicates that no more variable substitution is required.

GETINPUTLINE

Reads a line of input.

Syntax

Hide Copy Code

```
CALL _MTPLIB :GETINPUTLINE
```

Returns

RET Line of text read (can be empty).

Description

The **GETINPUTLINE** procedure reads a single line of text from the keyboard and returns it in the RET variable. This allows a **script** to prompt for and receive interactive input. To terminate the input line and continue **script** execution, press **Ctrl+Z** followed by **Enter** (or the other way around).

Troubleshooting Tip

There is a bug in the **CHOICE** command that ships with the resource kit. After a **CHOICE** command executes, all subsequent **COPY** commands that read from the console do not echo typed characters. This therefore disables keyboard echo for the **GETINPUTLINE** procedure. To avoid this problem, execute the **CHOICE** command in a sub-shell (using **CMD /C**).

Implementation

GETINPUTLINE uses the **COPY** command (line 265) to capture keyboard input. This means that **Ctrl+Z** (in addition to **Enter**) must be typed to terminate the input line. The line is copied to a temporary file, and this file is then parsed by the **FOR** command to capture the file contents into the **RET** variable.

The **FOR** command (line 266) actually parses the results of a **TYPE** command, rather than parsing the temporary file directly. This is because the **FOR** command cannot correctly process a file name containing spaces (it mistakenly assumes that the spaces separate multiple file names). Instead, the **TYPE** command (which can correctly handle file names with spaces) is used, and its output is parsed.

GETSYNCFILE

Obtains the name of a synchronization file.

Syntax

Hide Copy Code

```
CALL _MTPLIB :GETSYNCFILE
```

Returns

RET Name of the file to use for synchronization.

Description

The **GETSYNCFILE** procedure returns the name of a file that can be used by the **SETSYNCFILE**, **WAITSYNCFILE**, and **DELSYNFILE** procedures for **script** synchronization purposes. The file name is returned in the **RET** variable.

Typically, **GETSYNCFILE** is called by the "master" **script**, which then passes the file name to the "slave" **script** either as a command line argument or in a variable.

Using files for **script** synchronization is described in Chapter 4.

Implementation

GETSYNCFILE is simply a wrapper procedure for the **GETTEMPNAME** procedure.

SETSYNCFILE

Sets a synchronization file.

Syntax

Hide Copy Code

```
CALL _MTPLIB :SETSYNCFILE filename
```

Arguments

filename Name of the file to set. Typically obtained from **GETSYNCFILE**.

Description

The **SETSYNCFILE** procedure "sets" the synchronization file. This causes any **script**s which are waiting for the file to be set (via the **WAITSYNCFILE** procedure) to continue execution. A synchronization file is "set" by being created. A slave **script** typically sets a synchronization file to indicate to a master **script**(s) that it has completed processing.

The file to set is specified by filename, which is typically a name returned by the **GETSYNCFILE** procedure.

Using files for **script** synchronization is discussed in Chapter 4.

Implementation

A file is "set" by being created. Therefore, the procedure simply uses the **ECHO** command (line 290) to output a single period character to the file.

DELSYNCFILE

Deletes a synchronization file.

Syntax

Hide Copy Code

```
CALL _MTPLIB :DELSYNCFILE filename
```

Arguments

filename Name of the file to delete. Typically obtained from **GETSYNCFILE**.

Description

The **DELSYNCFILE** procedure deletes a synchronization file after it has been used. The file to delete is specified by the filename argument. Typically, this file name is obtained using the **GETSYNCFILE** procedure.

DELSYNCFILE should be called to delete the synchronization file after it has been used. Otherwise, the Windows temporary directory will gradually fill with old synchronization files. After **DELSYNCFILE** has been called, it is possible to use the same file again for another synchronization event.

Using files for **script** synchronization is discussed in Chapter 4.

Implementation

DELSYNCFILE is simply a wrapper for the DEL command.

WAITSYNCFILE

Waits for a synchronization file.

Syntax

Hide Copy Code

```
CALL _MTPLIB :WAITSYNCFILE filename [timeout]
```

Arguments

filename Name of the file to wait for. Typically obtained from **GETSYNCFILE**.

timeout Timeout period, in seconds (the default is 60).

Returns

RET Timeout remaining, or 0 if a timeout occurred.

Description

The **WAITSYNCFILE** procedure waits for a synchronization file to set. The file upon which to wait is specified by the filename argument. The procedure waits for up to timeout seconds (the default is 60 seconds) before failing with a timeout. Typically, a master **script** waits for a slave **script** to complete by using the **WAITSYNCFILE** procedure.

Using files for **script** synchronization is discussed in Chapter 4.

Implementation

WAITSYNCFILE contains a simple loop (lines 317 to 320) that checks for the existence of the specified synchronization file. The loop (and procedure) exits as soon as the file exists. While the file does not exist, the loop continues. Each pass through the loop contains a **SLEEP 1** command, which suspends execution for 1 second. Thus, the file is polled every second until it exists. Forcing the **script** to sleep for one second also yields the CPU while the **script** is waiting, and the loop thus consumes almost no CPU time while waiting.

Since the loop executes once per second (approximately), the timeout code simply counts down until the timeout reaches zero, at which time the loop executes regardless of the state of the synchronization file. Notice the use of the double IF command (line 320) to create an AND condition. This can be interpreted as "if the timeout has not expired and the synchronization file does not exist, continue looping."

GETTEMPNAME

Creates a unique temporary file name.

Syntax

Hide Copy Code

```
CALL _MTPLIB :GETTEMPNAME
```

Returns

RET Temporary file name (can contain spaces).

Description

The **GETTEMPNAME** procedure creates a temporary file name that can be used for temporary storage of data. The file name is guaranteed not to exist and to be in a location where unrestricted read/write access is permitted.

Implementation

The temporary file name is formed from the prefix **~SH**, a pseudo-random number, and the suffix **.TMP**. The pseudo-random number uses the same algorithm as the **RAND** procedure, though it uses an independent seed value. If the TEMP variable exists, it is used as the path name for the file. Otherwise, if the **TMP** variable exists, it is used as the path name. Otherwise, the **%SYSTEMROOT%** directory is used.

Chapter 5 - A Scripting Toolkit Part II: Real-World Scripting

Hide Shrink Copy Code

```
01. @echo OFF
02. @if not "%ECHO%"=="" echo %ECHO%
03. @if not "%OS%"=="Windows_NT" goto DOSEXIT
04. rem $Workfile: skeleton.bat $ $Revision: 2 $ $Date: 12/04/97 9:51a $
05. rem $Archive: /TimH/Pubs/Books/Macmillan/Windows NT
                  Scripting/Scripts/skeleton.bat $
06.
07. rem Set local scope and call MAIN procedure
08. setlocal & pushd & set RET=
09.     set SCRIPTNAME=%~n0
10.     set SCRIPTPATH=%~f0
11.     if "%DEBUG%"=="1" (set TRACE=echo) else (set TRACE=rem)
12.     call _mtplib :INIT %SCRIPTPATH%
13.     if /i {%1}=={/help} (call :HELP %2) & (goto :HELPEXIT)
14.     if /i {%1}=={/?} (call :HELP %2) & (goto :HELPEXIT)
15.     call :MAIN %*
16.     :HELPEXIT
17. popd & endlocal & set RET=%RET%
18. goto :EOF
19.
20. rem ////////////////////////////////////////////////////////////////////
21. rem HELP procedure
22. rem Display brief on-line help message
23. rem
24. :HELP
25. if defined TRACE %TRACE% [proc %0 %*]
26.     rem Put help message here...
27.
28. goto :EOF
29.
30. rem ////////////////////////////////////////////////////////////////////
31. rem MAIN procedure
32. rem  Chapter 5: A Scripting Toolkit
33. :MAIN
34. if defined TRACE %TRACE% [proc %0 %*]
35.     rem Put main script code here...
36.
37. goto :EOF
38.
39. rem ////////////////////////////////////////////////////////////////////
40. rem Additional procedures go here...
41.
42. rem These must be the FINAL LINES in the script...
43. :DOSEXIT
44. echo This script requires Windows NT
45.
46. rem ////////////////////Chapter 5  -  A Scripting Toolkit Part II:
                                        Real-World Scripting
```

Hide Shrink Copy Code

```
01. @echo OFF
02. @if not "%ECHO%"=="" echo %ECHO%
03. @if not "%OS%"=="Windows_NT" goto DOSEXIT
04. rem $Workfile: _libskel.bat $ $Revision: 2 $ $Date: 12/04/97 9:51a $
05. rem $Archive: /TimH/Pubs/Books/Macmillan/Windows NT
                  Scripting/Scripts/_libskel.bat $
06.
07. rem If no arguments, show version information and exit
08. if "%1"=="" (
09.     (echo Script Library Skeleton [%0] $Revision: 2 $)
10.     (goto :EOF)
11. )
12.
13. rem At least one argument, so dispatch to procedure
14. set _PROC=%1
15. shift /1
16. goto %_PROC%
17.
18. rem ////////////////////////////////////////////////////////////////////
19. rem INIT procedure
20. rem Must be called in local state before other procs are used
21. rem
22. :INIT
23. if defined TRACE %TRACE% [proc %0 %*]
24.
```

```
25. goto :EOF
26.
27. rem /////////////////////////////////////Part II: Real-World Scripting
28. rem CHECKX86 procedure
29. rem Sample procedure verifies that we are running on an Intel CPU
30. rem
31. rem Returns:    RET=1 if on x86, else RET=0
32. rem
33. :CHECKX86
34. if defined TRACE %TRACE% [proc %0 %*]
35.     if /i "%PROCESSOR_ARCHITECTURE%"=="x86" (set RET=1) else (set RET=0)
36. if defined TRACE %TRACE% [proc %0 returns {%RET%}]
37. goto :EOF
38.
39. rem /////////////////////////////////////////////////////////////////////
40. rem Additional procedures go here...
41.
42. rem These must be the FINAL LINES in the script...
43. :DOSEXIT
44. echo This script requires Windows NT
45.
46. rem /////////////////////////////////////Chapter 5: A Scripting Toolkit
```

Hide Shrink ▲ Copy Code

```
001. @echo OFF
002. @if not "%ECHO%"=="" echo %ECHO%
003. @if not "%OS%"=="Windows_NT" goto DOSEXIT
004. rem $Workfile: _mtplib.bat $ $Revision: 2 $ $Date: 12/04/97 9:51a $
005. -rem $Archive:
       /TimH/Pubs/Books/Macmillan/Windows NT Scripting/Scripts/_mtplib.bat $
006.
007. rem If no arguments, show version information and exit
008. if "%1"=="" (
009.     (echo Script MTP Script Library [%0] $Revision: 2 $)
010.     (goto :EOF)
011.)
012.
013. rem At least one argument, so dispatch to procedure
014. set _PROC=%1
015. shift /1
016. goto %_PROC%
017.
018. rem /////////////////////////////////////////////////////////////////////
019. rem INIT procedure
020. rem Must be called in local state before other procs are used
021. rem
022. :INIT
023. if defined TRACE %TRACE% [proc %0 %*]
024.
025. goto :EOF
026.
027. rem /////////////////////////////////////Part II: Real-World Scripting
028. rem VARDEL procedure
029. rem Delete multiple variables by prefix
030. rem
031. rem Arguments:    %1=variable name prefix
032. rem
033. :VARDEL
034. if defined TRACE %TRACE% [proc %0 %*]
035.     for /f "tokens=1 delims==" %%I in ('set %1 2^>nul') do set %%I=
036. goto :EOF
037.
038. rem /////////////////////////////////////////////////////////////////////
039. rem PARSECMDLINE procedure
040. rem Parse a command line into switches and args
041. rem
042. rem Arguments:    CMDLINE=command text to parse
043. rem        %1=0 for new parse (def) or 1 to append to existing
044. rem
045. rem Returns:    CMDARG_n=arguments, CMDSW_n=switches
046. rem        CMDARGCOUNT=arg count, CMDSWCOUNT=switch count
047. rem        RET=total number of args processed
048. rem
049. :PARSECMDLINE
050. if defined TRACE %TRACE% [proc %0 %*]
051.     if not {%1}=={1} (
052.         (call :VARDEL CMDARG_)
053.         (call :VARDEL CMDSW_)
054.         (set /a CMDARGCOUNT=0)
055.         (set /a CMDSWCOUNT=0)
056.     )
057.     set /a RET=0
058.     call :PARSECMDLINE1 %CMDLINE%
059.     set _MTPLIB_T1=
060. goto :EOF
061. :PARSECMDLINE1
062.     if {%1}=={} goto :EOF
063.     set _MTPLIB_T1=%1
064.     set _MTPLIB_T1=%_MTPLIB_T1:"=%
065.     set /a RET+=1
066.     shift /1
067.     if "%_MTPLIB_T1:~0,1%"=="/" goto :PARSECMDLINESW
068.     if "%_MTPLIB_T1:~0,1%"=="-" goto :PARSECMDLINESW
```

```
069.     set /a CMDARGCOUNT+=1
070.     set CMDARG_%CMDARGCOUNT%=%_MTPLIB_T1%
071.     goto :PARSECMDLINE1
072.     :PARSECMDLINESW
073.     set /a CMDSWCOUNT+=1
074.     set CMDSW_%CMDSWCOUNT%=%_MTPLIB_T1%
075.     goto :PARSECMDLINE1
076. goto :EOF  Chapter 5: A Scripting Toolkit
077.
078. rem /////////////////////////////////////////////////////////////////////
079. rem GETARG procedure
080. rem Get a parsed argument by index
081. rem
082. rem Arguments:    %1=argument index (1st arg has index 1)
083. rem
084. rem Returns:    RET=argument text or empty if no argument
085. rem
086. :GETARG
087. if defined TRACE %TRACE% [proc %0 %*]
088.     set RET=
089.     if %1 GTR %CMDARGCOUNT% goto :EOF
090.     if %1 EQU 0 goto :EOF
091.     if not defined CMDARG_%1 goto :EOF
092.     set RET=%%CMDARG_%1%%
093.     call :RESOLVE
094. goto :EOF
095.
096. rem /////////////////////////////////////////////////////////////////////
097. rem GETSWITCH procedure
098. rem Get a switch argument by index
099. rem
100. rem Arguments:    %1=switch index (1st switch has index 1)
101. rem
102. rem Returns:    RET=switch text or empty if none
103. rem        RETV=switch value (after colon char) or empty
104. rem
105. :GETSWITCH
106. if defined TRACE %TRACE% [proc %0 %*]
107.     (set RET=) & (set RETV=)
108.     if %1 GTR %CMDSWCOUNT% goto :EOF
109.     if %1 EQU 0 goto :EOF
110.     if not defined CMDSW_%1 goto :EOF
111.     set RET=%%CMDSW_%1%%
112.     call :RESOLVE
113.     for /f "tokens=1* delims=:" %%I in ("%RET%") do (set RET=%%I) &
        (set RETV=%%J)
114. goto :EOF
115.
116. rem /////////////////////////////////////////////////////////////////////
117. rem FINDSWITCH procedure
118. rem Finds the index of the named switch
119. rem
120. rem Arguments:    %1=switch name
121. rem        %2=search start index (def: 1)
122. rem
123. rem Returns:    RET=index (0 if not found)
124. rem        RETV=switch value (text after colon) Part II: Real-World
                    Scripting
125. rem
126. :FINDSWITCH
127. if defined TRACE %TRACE% [proc %0 %*]
128.     if {%2}=={} (set /a _MTPLIB_T4=1) else (set /a _MTPLIB_T4=%2)
129.     :FINDSWITCHLOOP
130.         call :GETSWITCH %_MTPLIB_T4%
131.         if "%RET%"=="" (set RET=0) & (goto :FINDSWITCHEND)
132.         -if /i "%RET%"=="%1" (set RET=%_MTPLIB_T4%) & (goto :FINDSWITCHEND)
133.         set /a _MTPLIB_T4+=1
134.     goto :FINDSWITCHLOOP
135.     :FINDSWITCHEND
136.     set _MTPLIB_T4=
137. goto :EOF
138.
139. rem /////////////////////////////////////////////////////////////////////
140. rem REGSETM and REGSETU procedures
141. rem Set registry values from variables
142. rem
143. rem Arguments:    %1=reg context (usually script name)
144. rem        %2=variable to save (or prefix to save set of vars)
145. rem
146. :REGSETM
147. if defined TRACE %TRACE% [proc %0 %*]
148.     for /f "tokens=1* delims==" %%I in (‘set %2 2^>nul’)
        do call :REGSET1 HKLM %1 %%I "%%J"
149. goto :EOF
150. :REGSETU
151. if defined TRACE %TRACE% [proc %0 %*]
152.     -for /f "tokens=1* delims==" %%I in (‘set %2 2^>nul’)
         do call :REGSET1 HKCU %1 %%I "%%J"
153. goto :EOF
154. :REGSET1
155.     set _MTPLIB_T10=%4
156.     set _MTPLIB_T10=%_MTPLIB_T10:\=\\%
157.     reg add %1\Software\MTPScriptContexts\%2\%3=%_MTPLIB_T10% >nul
158.     -reg update %1\Software\MTPScriptContexts\%2\%3=%_MTPLIB_T10% >nul
```

```
159. goto :EOF
160.
161. rem /////////////////////////////////////////////////////////////////////
162. rem REGGETM and REGGETU procedures
163. rem Get registry value or values to variables
164. rem
165. rem Arguments:    %1=reg context (usually script name)
166. rem        %2=variable to restore (def: restore entire context)
167. rem
168. rem Returns:    RET=value of last variable loaded
169. rem
170. rem WARNING:    The "delims" value in the FOR commands below is a TAB
                     Chapter 5: A Scripting Toolkit
171. rem        character, followed by a space. If this file is edited by
172. rem        an editor which converts tabs to spaces, this procedure
173. rem        will break!!!!!
174. rem
175. :REGGETM
176. if defined TRACE %TRACE% [proc %0 %*]
177.     for /f "delims=     tokens=2*" %%I in
       (‘reg query HKLM\Software\MTPScriptContexts\%1\%2 ^|find "REG_SZ"’)
        do call :REGGETM1 %%I "%%J"
178. goto :EOF
179. :REGGETU
180. if defined TRACE %TRACE% [proc %0 %*]
181.     for /f "delims=     tokens=2*" %%I in
       (‘reg query HKCU\Software\MTPScriptContexts\%1\%2 ^|find "REG_SZ"’)
        do call :REGGETM1 %%I "%%J"
182. goto :EOF
183. :REGGETM1
184.     set _MTPLIB_T10=%2
185.     set _MTPLIB_T10=%_MTPLIB_T10:\\=\%
186.     set _MTPLIB_T10=%_MTPLIB_T10:"=%
187.     set %1=%_MTPLIB_T10%
188.     set RET=%_MTPLIB_T10%
189. goto :EOF
190.
191. rem /////////////////////////////////////////////////////////////////////
192. rem REGDELM and REGDELU procedures
193. rem Delete registry values
194. rem
195. rem Arguments:    %1=reg context (usually script name)
196. rem        %2=variable to delete (def: delete entire context)
197. rem
198. :REGDELM
199. if defined TRACE %TRACE% [proc %0 %*]
200.     call :GETTEMPNAME
201.     echo y >%RET%
202.     reg delete HKLM\Software\MTPScriptContexts\%1\%2 <%RET% >nul
203.     del %RET%
204. goto :EOF
205. :REGDELU
206. if defined TRACE %TRACE% [proc %0 %*]
207.     call :GETTEMPNAME
208.     echo y >%RET%
209.     reg delete HKCU\Software\MTPScriptContexts\%1\%2 <%RET% >nul
210.     del %RET%
211. goto :EOF
212.
213.
214. rem ////////////////////////////////////////Part II: Real-World Scripting
215. rem SRAND procedure
216. rem Seed the random number generator
217. rem
218. rem Arguments:    %1=new seed value
219. rem
220. :SRAND
221. if defined TRACE %TRACE% [proc %0 %*]
222.     set /a _MTPLIB_NEXTRAND=%1
223. goto :EOF
224.
225. rem /////////////////////////////////////////////////////////////////////
226. rem RAND procedure
227. rem Get next random number (0 to 32767)
228. rem
229. rem Returns:    RET=next random number
230. rem
231. :RAND
232. if defined TRACE %TRACE% [proc %0 %*]
233.     if not defined _MTPLIB_NEXTRAND set /a _MTPLIB_NEXTRAND=1
234.     set /a _MTPLIB_NEXTRAND=_MTPLIB_NEXTRAND * 214013 + 2531011
235.     set /a RET=_MTPLIB_NEXTRAND ^>^> 16 ^& 0x7FFF
236. goto :EOF
237.
238. rem /////////////////////////////////////////////////////////////////////
239. rem RESOLVE procedure
240. rem Fully resolve all indirect variable references in RET variable
241. rem
242. rem Arguments:    RET=value to resolve
243. rem
244. rem Returns:    RET=as passed in, with references resolved
245. rem
246. :RESOLVE
247. if defined TRACE %TRACE% [proc %0 %*]
```

```
    :RESOLVELOOP
        if "%RET%"=="" goto :EOF
        set RET1=%RET%
        for /f "tokens=*" %%I in (‘echo %RET%’) do set RET=%%I
    if not "%RET%"=="%RET1%" goto :RESOLVELOOP
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem GETINPUTLINE procedure
rem Get a single line of keyboard input
rem
rem Returns:    RET=Entered line
rem
:GETINPUTLINE
if defined TRACE %TRACE% [proc %0 %*]Chapter 5: A Scripting Toolkit
    call :GETTEMPNAME
    set _MTPLIB_T1=%RET%
    copy con "%_MTPLIB_T1%" >nul
    for /f "tokens=*" %%I in (‘type "%_MTPLIB_T1%"’) do set RET=%%I
    if exist "%_MTPLIB_T1%" del "%_MTPLIB_T1%"
    set _MTPLIB_T1=
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem GETSYNCFILE procedure
rem Get a sync file name (file will not exist)
rem
rem Returns:    RET=Name of sync file to use
rem
:GETSYNCFILE
if defined TRACE %TRACE% [proc %0 %*]
    call :GETTEMPNAME
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem SETSYNCFILE procedure
rem Flag sync event (creates the file)
rem
rem Arguments:    %1=sync filename to flag
rem
:SETSYNCFILE
if defined TRACE %TRACE% [proc %0 %*]
    echo . >%1
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem DELSYNCFILE procedure
rem Delete sync file
rem
rem Arguments:    %1=sync filename
rem
:DELSYNCFILE
if defined TRACE %TRACE% [proc %0 %*]
    if exist %1 del %1
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem WAITSYNCFILE
rem Wait for sync file to flag
rem
rem Arguments:    %1=sync filename
rem        %2=timeout in seconds (def: 60)
rem   Part II: Real-World Scripting
rem Returns:    RET=Timeout remaining, or 0 if timeout
rem
:WAITSYNCFILE
if defined TRACE %TRACE% [proc %0 %*]
    if {%2}=={} (set /a RET=60) else (set /a RET=%2)
    if exist %1 goto :EOF
    :WAITSYNCFILELOOP
        sleep 1
        set /a RET-=1
    if %RET% GTR 0 if not exist %1 goto :WAITSYNCFILELOOP
goto :EOF

rem /////////////////////////////////////////////////////////////////////
rem GETTEMPNAME procedure
rem Create a temporary file name
rem
rem Returns:    RET=Temporary file name
rem
:GETTEMPNAME
if defined TRACE %TRACE% [proc %0 %*]
    if not defined _MTPLIB_NEXTTEMP set /a _MTPLIB_NEXTTEMP=1
    if defined TEMP (
        (set RET=%TEMP%)
    ) else if defined TMP (
        (set RET=%TMP%)
    ) else (set RET=%SystemRoot%)
    :GETTEMPNAMELOOP
        set /a _MTPLIB_NEXTTEMP=_MTPLIB_NEXTTEMP * 214013 + 2531011
        set /a _MTPLIB_T1=_MTPLIB_NEXTTEMP ^>^> 16 ^& 0x7FFF
        set RET=%RET%\~SH%_MTPLIB_T1%.tmp
    if exist "%RET%" goto :GETTEMPNAMELOOP
```

```
342.     set _MTPLIB_T1=
343. goto :EOF
344.
345. rem These must be the FINAL LINES in the script...
346. :DOSEXIT
347. echo This script requires Windows NT
348.
349. rem ////////////////////Chapter 5: A Scripting ToolkitPart II:
                                      Real-World Scripting
```


License

Share

About the Author

New Riders
United States No Biography provided

You may also be interested in...

Comments and Discussions

Fine thing
MLoibl 14-Feb-01 4:00

Not that I can use these things by not I think its important that a developer is familiar with batch-processing.
Congratulations.

Reply · Email · View Thread

Is this a joke?
Kastellanos Nikos 14-Feb-01 2:53

Does this think really exist?
Gee!
I wonder how many people know/use those Batch file command extensions.

Memory leaks is the price we pay

Reply · Email · View Thread

Re: Is this a joke?
Anonymous 23-Jun-03 14:57

don't know the number but I STILL have important production systems using 'em.

Reply · View Thread | Edit · Delete

Re: Is this a joke?
Assarbad 8-Jun-06 8:25

Well, at least one uses it - me - for years already and hopefully for years to come ...

If you are administrator you will appreciate the power of these scripts and you can often spare to use Perl.

And as a developer you will get in touch with it if you use the DDKBUILD scripts available out there.

Reply · Email · View Thread

Re: Is this a joke?
Kastellanos Nikos 8-Jun-06 11:21

5 years pass since my first post, and i had completely
forget about this.
Thanks for remind me.

Reply · Email · View Thread

Batch scripting will outlive us all.
the ops guy 9-Mar-08 22:55

The batch interpreter opens the batch file, reads a command line, closes the batch file, and executes the command. Then it reopens the batch file, seeks to the last position, reads a line, closes the batch file and executes. And so on. That means you can edit a batch while it is running. For this reason, batch scripting has been used heavily in operations for what, 35+ years now?

There are many other scripting languages that are a lot cleaner, safer, more powerful and easier to read (Perl, python, PHP, powershell just to name a few). But those are all memory resident. That's the price you pay for a more complex grammar. For some applications, only batch scripting is suitable.

Reply · Email · View Thread 5.00/5 (1 vote)

Hahaha
Pascal Binggeli 14-Feb-01 2:38

Now, I known I will not buy the book... (DOS/BATCH (only for NT) ... while it should be a Windows Scripting Host library)

Reply · Email · View Thread | Edit · Delete

Re: Hahaha

Anonymous 14-Feb-01 3:16

Thats a bit unfair.

Tim Hill has written a second book called Windows 2000 Windows Script Host, and that's probably what you are looking for.

I have both books and I can only say that they are worth every penny.

Reply · View Thread | Edit · Delete

Re: Heu sorry
Pascal Binggeli 14-Feb-01 10:16

Yes I admit It was unfair about the quality of the book. My apologize on that point.

Anyway an article about DOS/Batch should have been posted a least 4 years ago...

Reply · Email · View Thread | Edit · Delete

Re: Heu sorry
MLoibl 14-Feb-01 13:42

Its never too late.
BTW: Do you know this one:
SET CURRENT_DIR=%cd%
When you test it, you may recognize that it only works with Win2000, but not for NT4 (Win9x I don't know).
So I think it still is a good time to talk about batch-processing...
Does anyone have a solution how to get the current path into a variable unter NT4 ?

Reply · Email · View Thread

Re: Heu sorry
Anonymous 20-Feb-01 3:15

Try this:

for /f %i in ('cd') do set CURRENT_DIR=%i (command line)

for /f %%i in ('cd') do set CURRENT_DIR=%%i (batch file)

Reply · View Thread | Edit · Delete

Re: Heu sorry
Assarbad 8-Jun-06 8:17

Yes, if this article was about DOS-Batch you were right.

Sadly it is about NT Scripting which is a lot more powerful than DOS-Batch, although many Syntax elements are equal or similar.

Reply · Email · View Thread

Re: Hahaha
John Hamlin 1-Jan-02 15:15

Absolutely! I have both of Hill's books and use them BOTH daily.

Reply · Email · View Thread

Script file to shut down server at a certain time
SAI 31-May-01 0:49

hi,

I'm a budding systems administrator. Now i'm trying to write scripts at my work as it demands.

what I'm looking for is writing a script so that my NT4 Server shuts down at a certain time by disconnecting/stopping all other services which are running.

As i'm new to scripting, i would like to take guidance in doing so. So can neone give me the code so that I can start by implementing the code and improve based on the code and then I can try out new scripts.

I dont even have a faintest idea how to start this?

SAI

SAI

Reply · Email · View Thread | Edit · Delete

Re: Script file to shut down server at a certain time
Anonymous 14-Aug-01 20:00

resource kit

Reply · View Thread | Edit · Delete

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin