

12 How to...?

This chapter contains answers to questions »How to...« and additional information for developing scripts. Use this chapter as a guide, if you need some information about how to solve special questions.

Programming tricks

Below you will find a collection of programming tricks and techniques (some already mentioned in other chapters), which deals with WSH script programming in general.

How to debug?

To check a variable's value, you can use the *Echo* method within your script. Add the statement shown below in VBScript:

```
WScript.Echo "Variable xyz: ", xyz
```

In JScript the arguments must be enclosed in parenthesis and separated with commas:

```
WScript.Echo ("Variable xyz: ", xyz);
```

If you like to know whether a special branch within the script is executed, you may add also an *Echo* call into the code, indicating that the inspected program point is reached.

To test the program step by step, I recommend the Microsoft Script Debugger (or the Microsoft Script Editor's debugging features in the case Visual Studio or Microsoft Office 2000 is installed). Inset the commands *stop* (VBScript) or *debugger* into the script's source code to launch the debugger automatically.

NOTE: Some people reported problems with these keywords. In this case add a simple *Echo* call and launch your debugger manually. A more detailed description of these topics may be found in chapter 2.

At the time I wrote this chapter I got the information, that WSH 2.0 supports a flag to enable debugging. WSH 2.0 supports XML format within script files, and there is an option to suppress debugging.

Also the *PrimalSCRIPT* script editor I mentioned in chapter 2 probably supports in version 2.x a kind of »in-place« debugging. You can execute a compile/syntax check within the editor window and launch also an external debugger.

How to handle Run-time errors?

VBScript supports run-time Error handling with the statement.

```
On Error Resume Next
```

The statement causes after a run-time error that the next statement will be executed. The error code can be retrieved from the *Err* object.

```
valx = WSH.RegRead ( "...", "xxxx")
If Err <> 0 Then
....
```

After a run-time error occurs, the value of the *Err* object is not equal 0. The error code may be retrieved from the *Number* property (*Err.Number*). I have used this technique within several chapters to handle run-time errors.

Use the statement:

```
On Error GoTo 0
```

to disable run-time error handling within the script. After this statement, run-time errors are handled by WSH.

TIP: The error numbers returned from *Err.Number* are described in the VBScript/JScript help (msdn.microsoft.com/scripting). But these help files cover only language errors. Error numbers caused from the operating system are not described. Brad Martinez wrote a neat program *Win32 Error Codes* which displays an error text after entering the error code into a dialog box window. The module may be downloaded from <http://members.aol.com/btmzt/vb>.

JScript run-time error handling

In JScript run-time error handling is supported since the script-engine version 5.0 using the *try {...} catch (e) {...}* sequence:

```
try
{
var valx = WSH.RegRead ( "...", "xxxx");
}
catch(e)
{
if (e != 0)
WScript.Echo ("Error during Registry access");
}
```

The *try* keyword must be set in front of a statement. Enclose the statement or a block of statements in brackets {...}. The *catch(e)* statement is called, if a run-time error occurs. The variable *e* receives the error object. This error object can be evaluated in the statements following in the *catch* block, which must be enclosed also in brackets {...}.

NOTE: Further information about error handling may be found in previous chapters of this book and in the VBScript/JScript help files (msdn.microsoft.com/scripting).

JScript (language engine 3.x) doesn't supports *try ... catch* error handling. There is a free ActiveX control called *ScriptX* to make JScript more close to VBScript. For instance *ScriptX* allows you to call a method of any object and analyze the success/failure code. Parameters are packed into a JScript array (*args*) and passed by reference. The return values are passed back as *args.value*. So, two goals are achieved: *ByRef* semantic and error handling. Below is a small sample that calls *obj.Method(a, b, c)*:

```
function CallAX(obj, a, b, c)
```

```
{ // pack params to array
args = new Array(a, b, c);
// same as: args.value = obj.Method(a, b, c)
error = factory.js.InvokeByRefResult(obj, "Method", args);
if ( error != 0 )
    { alert("AX Error: " + error); }
else
    {
    // 'return_value + a + b + c' on output.
    alert( args.value + args[0] + args[1] + args[2] )
    }
}
```

This control supports also a couple of other neat operation. The specs for ScriptX may be found at <http://www.meadroid.com/scriptx/ScriptXspec.htm>. The control itself can be downloaded from <http://www.meadroid.com/scriptx/>.

How use WSH- and Script-Engine properties?

WSH properties may be accessed using the *WScript* object. Below are VBScript statements to query some properties:

```
WScript.Echo "Application: " & WScript.Application
WScript.Echo "Name: " & WScript.Name
WScript.Echo "Version: " & WScript.Version
WScript.Echo "FullName: " & WScript.FullName
WScript.Echo "Path: " & WScript.Path
```

To check the Script-Engine properties, use the following code:

```
WScript.Echo "Script-Engine: ", ScriptEngine()
WScript.Echo "Version: ", ScriptEngineMajorVersion(), ".", _
    ScriptEngineMinorVersion()
WScript.Echo "Build: ", ScriptEngineBuildVersion()
```

Further details may be found in chapter 4 in the section »Working with the WScript object«.

How to get the script's path?

Sometimes it comes in handy to know the path to your script. For instance, if you like to load a document file, store the file into the folder of your script and try to get the path. This is much more flexible as working with absolute paths. WSH owns no function or method to retrieve the path. Therefore many programmers created their own solutions to get the script path from the *Script-FullName* property of the *WScript* object. I have used the following VBScript code within my samples:

```
Function GetPath
' Retrieve path to the script file
DIM path
path = WScript.ScriptFullName ' script file name
GetPath = Left(path, InstrRev(path, "\"))
End Function
```

In JScript you may use the following function to retrieve the path:

```
function GetPath ()
// Retrieve the script path
{
    var path = WScript.ScriptFullName; // script name
    path = path.substr(0,path.lastIndexOf("\\")+1);
    return path;
}
```

Both functions depends on the fact that a path must end with a backslash "\". If WSH becomes available on other machines (like Macintosh or Unix), this method fails. Then the following construction may be applied to retrieve the path:

```
path = Left(Wscript.ScriptFullName, _
            Len(Wscript.ScriptFullName) - Len(Wscript.ScriptName))
```

This statement cuts the script file name from the full name by a simple »subtraction«. So it will be independent on the file naming convention. If you like to avoid this trick, you can let the *FileSystemObject* do the job for you. This object provides a method to retrieve the parent folder of a given path. Use the following statements to retrieve the path:

```
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
Path = fso.GetParentFolderName(Wscript.ScriptFullName)
```

The advantage of this construction is: as long as Microsoft implements the *GetParentFolderName* in a right order, you will get the path (independent of the operating system). The penalty you have to pay: You create a file system object (which costs extra memory) and execute a method (which costs extra time).

How to get the current directory?

Some programmers ask for code to retrieve the current directory (such a function is available in VBA). The current directory within a script is identical to the directory from which the script is executed. So you can use either the code shown in the previous section or the following statements:

```
Dim fso
set fso = WScript.CreateObject("Scripting.FileSystemObject")
' CurrentDir = fso.GetAbsolutePathName("") ' or use the following syntax
CurrentDir = fso.GetAbsolutePathName(".")
```

This sequence uses the *FileSystemObject* to retrieve the path using the *GetAbsolutePathName* method with the parameter ".". This will be demonstrated in the following VBScript listing, which shows the current directory and the script path in a message box.

```
' *****
' File:    CurrentDir.vbs (WSH sample in VBScript)
' Author:  G. Born
'
' Retrieves the current directory
' *****
Option Explicit
```

```
WScript.Echo "Script Path:", GetPath(), vbCrLf, _
    "Current Directory:", CurrentDir()

Function CurrentDir
Dim fso
    set fso = WScript.CreateObject("Scripting.FileSystemObject")
    CurrentDir = fso.GetAbsolutePathName(".")
End Function

Function GetPath
' Retrieve path to the script file
DIM path
path = WScript.ScriptFullName ' script file name
GetPath = Left(path, InstrRev(path, "\"))
End Function
' End
```

Listing 12.1.
CurrentDir.vbs

NOTE: The file *CurrentDir.vbs* is located in the folder *\Samples\chapter12*.

How to calculate Date differences

Would you calculate date differences? This can be done in VBScript using the *DateDiff* function. According to the VBScript help you must specify the interval in the first parameter, the other two parameters are the dates to be used to calculate the difference.

```
WScript.Echo DateDiff("d", Now, "1.1.2000") & _
    " days left to the millennium big bang..."
```

The statement above uses the "d" (days) interval to calculate the days left to the year 2000.

NOTE: Take care that the date separator depends on the locale settings of your operating system.

How to use Event handling?

The Windows Script Host provides a feature to handle external events of automation objects. A script can call a method of an object or access its properties, and the object can call back an event handling procedure implemented within the script. To enable this call back feature, you must link the script's event handler to the outgoing interface of the external automation object. This need to be done when instantiating the object with the *CreateObject* method. This method supports an optional parameter in which a prefix of the names of all event procedures may be passed. The following statement uses the prefix *Window_* for the event handler procedure and links it to the Internet Explorer *Application* object:

```
Set oIE4 = WScript.CreateObject("InternetExplorer.Application", "Window_")
```

This means: All event handlers within the script must begin with the name *Window_*. Within a VBScript program we can use for instance the following procedure to catch an *OnUnload* event:

```
Sub Window_OnUnload()
```

```
' is raised on closing of the document
End Sub
```

The procedure name consists of the prefix (which we have already defined within the *CreateObject* method), an underscore and the name of the event (here I have used *OnUnload()*). It is required that the external object can raise such an event (if the event handler shall make sense).

Implement an event handler in VBScript

Let's have a real world example. The following little script demonstrates how to handle external events in VBScript. It uses a procedure to handle external events. The script launches the Internet Explorer and load the file *TextForm.htm* located in the script file's folder (**Figure 12.1**).

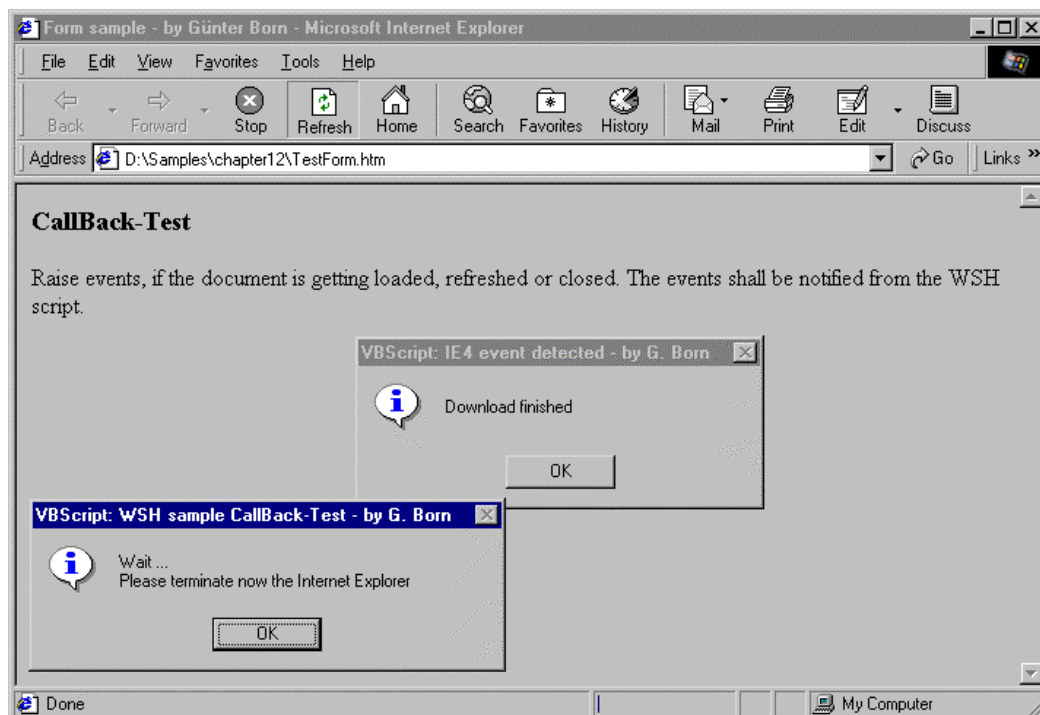


Figure 12.1.

A document shown in Microsoft Internet Explorer and some dialogs caused by the WSH script

After the script forces Internet Explorer to load the HTML document, several events are fired within this application. The events *DownloadBegin* and *DownloadComplete* will be raised twice for begin and end of the download. If you close the Internet Explorer window, an *OnQuit* event occurs. Dialog boxes created within the WSH script show all these events. These dialogs are shown before the Internet Explorers window becomes visible. The *MsgBox* call at the script's end forces the WSH script to stop. Although the script itself waits for closing the dialog box, all events are indicated by dialog boxes (created from the event handlers within the script - see **Figure 12.1**). Details may be obtained from the following code listing.

NOTE: It will be no problem, if the script terminates before the Internet Explorer is quitted. All events raised from the Internet Explorer are send to WSH, which passes these events to the script. If the script or an event handling procedure doesn't exists, WSH detects

this and the event isn't processed. So you don't have to worry about a kind of »disconnecting« the event handling function from your object.

```

' *****
' File:      IEEventHandling.vbs (WSH sample in VBScript)
' Author:    (c) G. Born
'
' Demonstrates how to use an event handling to
' handle IE events within a script.
' *****
Option Explicit
Dim WSHShell      ' declare variables
Dim oIE4
Dim path
Dim Title, Title1

Title = "WSH sample CallBack-Test - by G. Born"
Title1 = "IE4 event detected - by G. Born"

' *** get path to the script file, because the Form
' *** (HTML) must be located in this folder
path = GetPath() + "TestForm.htm"      ' File's path

' *** launch Internet Explorer, define CallBack-Prefix ***
Set oIE4 = WScript.CreateObject( _
    "InternetExplorer.Application", "Window_")
oIE4.navigate path      ' Form
oIE4.visible = 1        ' visible

' Stop script by displaying a message box. The dialogs from
' the event procedures should work, even if the script waits
' to close the message box.
MsgBox "Wait ..." + vbCRLF + _
    "Please terminate now the Internet Explorer" + _
    vbCRLF, vbOkOnly + vbInformation, Title

' User clicked the OK button
WScript.Quit()          ' terminate

' *****
' Here are the event handlers
' *****

Sub Window_DownloadBegin()
' raised from loading a document in IE
MsgBox "Download begins", _
    vbOkOnly + vbInformation, Title1
End Sub

Sub Window_DownloadComplete()

```

```

' raised from ending the download in IE
MsgBox "Download finished", _
    vbOkOnly + vbInformation, Title1
End Sub

Sub Window_OnUnload()
' raised from closing a document in IE
MsgBox "Close document", _
    vbOkOnly + vbInformation, Title1
End Sub

Sub Window_OnQuit()
' raised from quitting IE
MsgBox "Quit Internet Explorer", _
    vbOkOnly + vbInformation, Title1
End Sub

'#####
Function GetPath
' Retrieve path to the script file
DIM path
path = WScript.ScriptFullName ' script file name
GetPath = Left(path, InstrRev(path, "\"))
End Function
' End

```

Listing 12.2.

IEEventHandler.vbs

NOTE: The program *IEEventHandler.vbs* is located in the folder `\Samples\chapter12`. This folder contains also the required HTML file *TestForm.htm*. Another sample how to use an event handler may be found also in chapter 6 within the section »Forms input with a Callback function«.

How it work's in JScript?

In JScript you may use the same technique to implement a callback function for an event handler. You must only take care about the JScript syntax. The following listing shows the details of the JScript implementation of the previous VBScript sample. You should note that not all the events implemented within the script must be raised from the Internet Explorer. For instance, I haven't seen the *OnUnload* event during executing the sample. Details may be found within the listing.

```

//*****
// File:      IEEventHandlering.js (WSH sample in JScript)
// Author:    (c) G. Born
//
// Demonstrates how to use event handling to
// handle IE events within a script.
//*****

```



```
var path = GetPath() + "TestForm.htm";           // file must exists

// *** launch Internet Explorer, define CallBack-Prefix ***
var oIE4 = WScript.CreateObject(
    "InternetExplorer.Application", "Window_");
oIE4.navigate (path);                          // Form
oIE4.visible = 1;                              // visible

// Stop script by displaying a message box. The dialogs from
// the event procedures should work, even if the script waits
// to close the message box.
WScript.Echo ("Wait ...\n",
    "Please terminate now the Internet Explorer");

// User clicked the OK button
WScript.Quit();                                // ready

//*****
// Here are the event handlers
//*****

function Window_DownloadBegin()
// raised from loading a document in IE
{
    WScript.Echo ("IE event detected: Download begin");
}
function Window_DownloadComplete()
// raised from ending the download in IE
{
    WScript.Echo ("IE event detected: Download end");
}

function Window_OnUnload()
// raised from closing a document in IE
{
    WScript.Echo ("IE event detected: Document close");
}

function Window_OnQuit()
// raised from quitting IE
{
    WScript.Echo ("IE event detected: Quit IE4");
}
//#####
function GetPath ()
// Retrieve the script path
{
    var path = WScript.ScriptFullName; // script name
    path = path.substr(0,path.lastIndexOf("\\")+1);
```

```
return path;
}
// End
```

Listing 12.3.

IEEventHandling.js

NOTE: The sample *IEEventHandling.js* is located in the folder *\Samples\chapter12*.

Script calls and parameters

This section contains a few remarks about frequently asked questions concerning how to execute a script from a script, pass arguments to the script and launch other programs.

How to launch a script using Drag & Drop?

It would be supremely welcome, if a script could be launched using Drag & Drop, this means for instance: simply drag a document file over the script file's icon. As soon as you drop the document, the script is launched and processes the document file as an argument. Unfortunately this does not work, because the Windows Shell does not accept *VBS*- and *JS*-files as feasible programs. You must use a trick and fake an executable script: create a simple batch program that calls itself the WSH script. A BAT file can be used within Drag & Drop operations. Let's have a closer look at this technique. Assume we have for instance the script *Param.vbs* in the folder *E:\Samples\chapter12*, which shall support Drag & Drop. Therefore create a BAT-file within the same folder containing the following command:

```
@Start WScript.exe E:\Samples\chapter12\Param.vbs %1 %2 %3
```

This command passes the parameters submitted to the BAT file as arguments to the script. Here I have used only three parameters, but you may insert the placeholder %1, %2 up to %9 into the command line, to submit nine parameters. Additionally you can set the properties for this Batch program, to execute the MS-DOS window minimized and close the window automatically after the program terminates (right-click the BAT file and select *Properties* in the shortcut menu). The script mentioned above shows the parameters submitted from the BAT file.

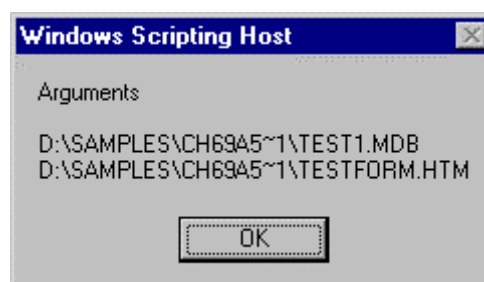


Figure 12.2.

Parameters submitted by Drag & Drop

If you drag for instance the file *TestForm.htm* over the batch program *Param.bat*, and drop the file, then the BAT file launches the WSH-Host and passes the script file name and the first three

parameters to this application. The program *Param.vbs* examines the submitted parameters and shows them in a dialog box. And these parameters are nothing else as the names of the files dragged to the bat file. In **Figure 12.2** we can see two file names, because I dragged two files to the bat file.

NOTE: The files *Param.bat*, *Param.pif* and *Param.vbs* are located in the folder *\Samples\chapter12*. Before you can test this example, you must copy the files into a folder, and edit the command in the bat file, to customize the path to your script file.

WSH scripts and the Windows NT-Scheduler

You may use the Windows NT scheduler to execute a script at a given time:

```
AT 18:00:00 /interactive "C:\Samples\chapter12\RunExit.vbs"
```

Within the AT command line you must write the whole path to the script file. And you must set the */interactive* flag, if the script requests some user interaction.

How to access script arguments?

I have mentioned the bat file trick to launch a WSH script per Drag & Drop. The script can access the submitted parameters using the *Arguments* property of the *WScript* object. This property returns a collection with all submitted arguments:

```
Set objArgs = WScript.Arguments      ' create object
For I = 0 to objArgs.Count - 1      ' all arguments
    text = text & objArgs(I) & vbCrLf ' fetch argument
Next
```

You may further samples for VBScript and JScript in chapter 4 within the section »Examining script parameters«.

How to call external applications?

To execute external applications you must use the *Run* method of the *WScript.Shell* object.

```
Set WSHShell = WScript.CreateObject ("WScript.Shell")
WSHShell.Run "%Windir%\Notepad.exe", 1
```

The first parameter submitted to the *Run* method defines the path with the executable program. The second parameter defines the window style. To execute a MS-DOS command, you need to use the following commands:

```
var WSHShell = WScript.CreateObject ("WScript.Shell");
WSHShell.Run ("%windir%\%comspec% /k dir C:\");
```

I have used in this case the environment variable *%comspec%* instead of *command.com* because *%comspec%* translates to *cmd.exe* in Windows NT and *command.com* in Windows 9x. This command written in JScript shows the content of your drive *C:* using the MS-DOS *DIR* command.

If the script shall wait till the launched process terminates, you must submit the value *true* within the third parameter:

```
Set WSHShell = WScript.CreateObject ("WScript.Shell")
```

```
WSHShell.Run "%Windir%\Notepad.exe", 1, true
```

If you launch the Windows-Explorers with this command, the *wait* mode fails. You may test this behavior with the file *RunWait.vbs* contained in the folder *\Samples\chapter12*. Edit the script and exchange the Excel call with a *Explorer.exe* call. This behavior may be explained, because the Explorer is a part of the Windows Shell and therefore always active. Launching *Explorer.exe* won't create a new process, instead a new thread is executed. Therefore the *wait* mode must fail!

Long file names within scripts

If your scripts contain long file names, or if you use long file names within the command to execute the script, you must enclose the paths in double-quotes:

```
WSHShell.Run "\"C:\Programs\Microsoft Office\Office\Excel.exe\"", 1, -1
```

A double-quote contained in a string must be written in VBScript as `""`. The first double-quote indicates the begin of the string. The two following double-quote characters `""` are signaling the double-quote which must be inserted into the string. In JScript a double-quote within a string must be written as `"\"`:

```
WSHShell.Run _  
  "\"C:\\Programs\\Microsoft Office\\Office\\Excel.exe\"", 1, -1
```

NOTE: When you build the command line by fetching a path name and then concatenating a string with the specific file name, you need to add double quotes around the resulting string (because the fetched path name may have an embedded space).

How to execute system calls using *Run*?

To call the Windows API from your WSH script, you need an ActiveX control (I have introduced this technique in chapter 12). But Windows 9x provides a few backdoors and helpers allowing you to access system routines from a script using the *Run* method.

Shutdown with RunDLL32.exe

In chapter 10 I have introduced an ActiveX control, which provides the *WSHExitWindows* method to shutdown, restart or logoff Windows 95/98. But you don't need this component to shutdown your Windows 95/98 from a WSH script. You may use the following command within the *Run* method to initiate a shutdown.

```
WSHShell.Run "%windir%\RunDll32.exe user,ExitWindows", 1, -1
```

The program *RunDll32.exe* is a helper to activate several Windows functions contained in (DLL) library files. The command given above *RunDll32* accesses the library *User.exe* and calls the *ExitWindows* API function, which is exported from this module.

TIP: You may use all DLL Libraries available under Windows within a *RunDLL32.exe* command. But there are restrictions: the names of the functions exported from the library are case sensitive (*exitwindows* is different from *ExitWindows*). And you can't pass arguments to a function using *Rundll32.exe*. The only option you have is to submit a string within the command-line. If the library examines the command-line, the option will be detected. But this is only true in rare cases.