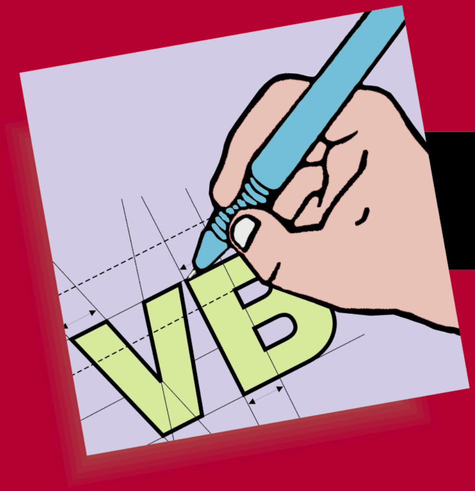
VBScript



Programmer's Reference

A comprehensive guide to version 5 of the VBScript language and syntax, together with practical demonstrations of its usage in context - be it server-side programming with ASP or WSH, or scripting in Internet Explorer with DHTML or HTML components.

wrox
PROGRAMMER TO PROGRAMMER™

Susanne Clark, Antonio De Donatis, Adrian Kingsley-Hughes, Kathie Kingsley-Hughes, Brian Matsik, Erick Nelson, Piotr Prussak, Daniel Read, Carsten Thomsen, Stuart Updegrave, Paul Wilton

Summary of Contents

Error Handling, Prevention and Debugging

Overview

Error handling, unlike some other features, has been one of the selling points of VBScript. In fact, until Version 5.0 of JScript had been released, VBScript held tremendous edge over JScript because of its error handling capabilities (at least on the server side of scripting). By now, you would expect volumes of literature on error handling to exist, but this couldn't be further from the truth. As the scripting hosts grow in their complexity, so do the general capabilities of scripted applications, and the end user's expectations. At the same time, however, the schedules get tighter and the workloads get bigger, making even ordinary bugs more difficult to catch. It seems that proper testing and error handling ends up on the back burner, and unjustifiably so, because simple error handling is not that difficult, as this chapter will show.

In this chapter, we will cover:

- How minute differences in hosts can affect runtime errors
- Different types of errors, and error display
- Basic handling of errors
- Strategies for handling errors in different situations
- Defensive coding strategies
- Debugging with Script Debugger and Visual InterDev
- Common errors and how to avoid them

Introduction

No matter how simple a VBScript project you are developing, there is always a need for effective error handling and debugging. If a project worked just fine the last time you tested it, it can be hard to see how error handling and debugging are at all relevant. In reality, script execution will depend on a variety of factors, starting with the user, and ending with the physical environment in which the project runs. To understand what the problems are, let's first consider the user.

Users rarely do what we expect them to do.

While many problems can be avoided by giving the user precise instructions and a clear interface in the first place, often the user does not take time to read the instructions, or does not understand them. So *you* understanding the way *your* project works might make it more difficult for *you* to anticipate the sort of mistakes a *user* might make. Thorough testing is therefore essential, especially when considering issues such as what happens if the user enters text when you are expecting numerical input? Does the script validate the data? Does the browser generate an error if the user clicks in the web page before a sub-procedure is completed? Even if all the user entered data is valid, there still exist possibilities that the user entered data may not work with other parts of the script – for instance, the data may represent a duplicate record, which will not be accepted by the database. Will the server script generate the error?

Next, the *dynamically* generated scripts which often depend on each other, are another common source of errors. What if one of the procedures does not perform exactly as in the test case or if the list box, containing an array of choices, is not present, or empty? Similarly, scripts depending on some components may not be able to access them. Perhaps the user has different *security preferences* than you anticipated, or does not have an appropriate component loaded on the system, or appropriate permissions to run it. What does your program do? Will your script attempt to log the error, or ask the user to file a report?

Finally, hardware issues may be responsible for serious deviations in script executions. The servers can be down, the client computer may be low on memory, or the disk that the script is trying to access may simply crash. Will an operator be alerted about a major malfunction?

In order to handle the error, you must anticipate it before it happens. Although defensive programming goes hand in hand with error handling, you'll have to figure out the trade offs, and choose the best technique for the situation.

Is there a part of the script that doesn't work *exactly* the way you expected it to and, while the user will quite probably discover it, you might not have noticed? These are all good reasons for emphasizing the following:

Thorough testing, debugging and error handling are vital for a project of any size.

Undeniably, it is a chore to plan for errors before they happen, but in the long term, it is well worth the effort and is a valuable habit to get into. Note that there is more to error handling than the glorified `On Error` statement (dealt with later in this chapter). Before you rush out to use it, you should realize that it is as often used to handle errors as it is to cover up bugs and sloppy coding. Error handling is therefore also about good programming practices and testing methods, as well as using the `On Error` statement to handle the true exceptions.

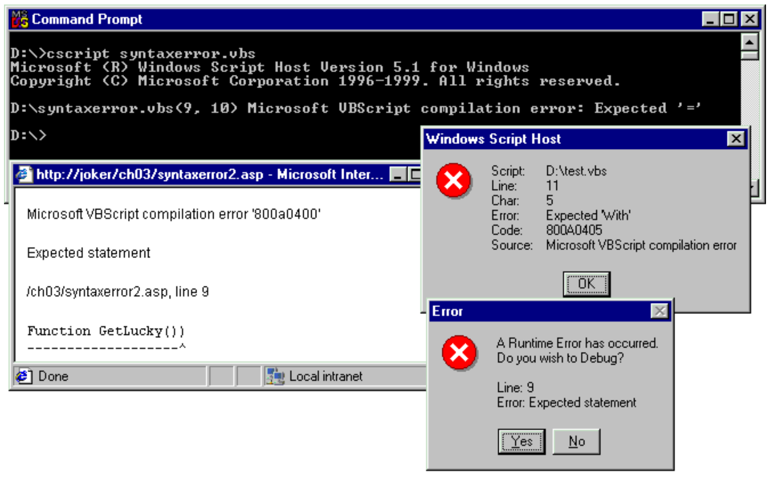
Errors are not the evil we are led to believe and they can happen for bad or good reasons. As long as they are caught and handled properly (sometimes, you will not be able to correct an error), your programs will run smoothly. Understanding the differences in the types of errors, the situations in which they occur, and the simple error-handling techniques available, is paramount to graceful error handling. The environment that your script feeds from, the complexity and your own understanding of the language and the language's facilities, all combine to produce a vast source of possible errors.

Note that error handling associated with the Script Control (Chapter 16) is slightly different than with other hosts. Although error handling within the Script Control works similarly to other scripting hosts, there is also the possibility of handling errors via the host (e.g. the VB application itself), with a distinction between compilation and run-time errors. For more specific information, consult the above chapter on Script Control.

Error handling and debugging can also give a much more professional finish to a project, as well as a sense of security that it will be able to stand up to at least some of what the users are going to throw at it. However, it is not only the user that can make mistakes: the errors can lurk in the script itself. Let's look at some other types of errors that can afflict your code.

Seeing the Error of your Ways

Error messages that are displayed by the host identify the line number and the nature of the error (see the figure below). Depending on the host, and on the nature of the error, the error code may be displayed as a decimal number (such as 1024) , a hexadecimal one (such as 800a0400), or simply as the text message identifying the error. If the error code is hexadecimal, and begins with 800A, it is thrown by the scripting engine, and the remaining four digits can be converted to its decimal representation. (They are covered in Appendix E, and are additionally listed in VBScript's help files, but without the hexadecimal representations.) Errors thrown by COM components and Windows are usually shown using hexadecimal codes.



```
D:\>cscript syntaxerror.vbs
Microsoft (R) Windows Script Host Version 5.1 for Windows
Copyright (C) Microsoft Corporation 1996-1999. All rights reserved.

D:\syntaxerror.vbs(9, 10) Microsoft VBScript compilation error: Expected '='

D:\>
```

```
Microsoft VBScript compilation error '800a0400'

Expected statement

/ch03/syntaxerror2.asp, line 9

Function GetLucky())
-------------------^
```

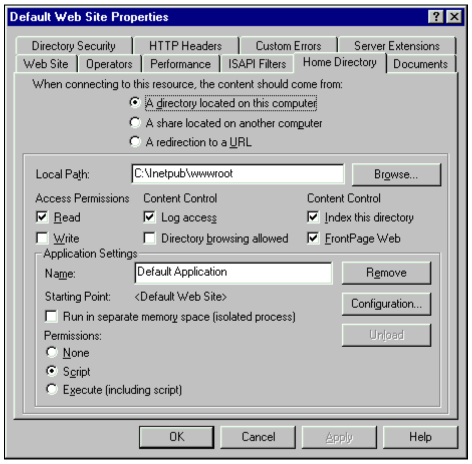
```
Script:  D:\test.vbs
Line:    11
Char:    5
Error:   Expected 'With'
Code:    800A0405
Source:  Microsoft VBScript compilation error
```

```
A Runtime Error has occurred.
Do you wish to Debug?

Line: 9
Error: Expected statement
```

Now, to make things more complex for the beginner, sometimes these error messages will not show up, or will show up in a disguised format. These problems can be caused by the configuration of the host and, at least in the debugging stage of the project, the host ought to be configured to handle errors in the way you want it to behave. Only two of the hosts can change the way in which errors are displayed: the ASP engine (the IIS server) and IE5.

Enabling Error Display in IIS

Error handling in IIS is done on a Web application level (note: this does not apply to PWS). Generally, each Web application will have an application initialization file – the `global.asa` file – in its directory (see Chapter 14 for a description of this file and of ASP in general). If you are working with a newly installed server, there is no need to override any of the settings. If you are inheriting a server, or just an IIS application, and errors messages are not being displayed, you should edit the application's properties through the IIS' MMC (Microsoft Management Console) as shown in the figure below. In Windows 2000 it is called Internet Information Services. Other errors (especially HTTP) can also be configured using the Custom Errors tab (for more information you should consult an IIS reference available with installation of the Option Pack, or with Windows 2000).


Default Web Site Properties

Directory Security | HTTP Headers | Custom Errors | Server Extensions
Web Site | Operators | Performance | ISAPI Filters | Home Directory | Documents

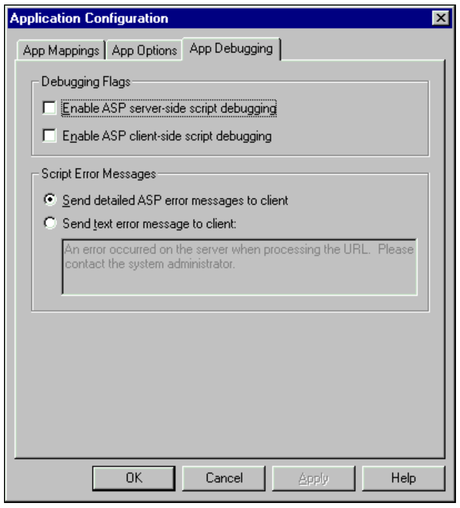
When connecting to this resource, the content should come from:
- [✓ A directory located on this computer]
- A share located on another computer
- A redirection to a URL

Local Path: C:\Inetpub\wwwroot

Access Permissions	Content Control	Content Control
☑ Read	☑ Log access	☑ Index this directory
☐ Write	☐ Directory browsing allowed	☑ FrontPage Web

Application Settings
Name: Default Application
Starting Point: <Default Web Site>
☐ Run in separate memory space (isolated process)
Permissions:
- None
- [✓ Script]
- Execute (including script)

Unfortunately, the error settings are hidden within the many options of the directory or file, or of the application, and to change the error options you have to locate the appropriate application, and then hit the Configuration… button to set the options available on the figure below. Obviously, in order to see the messages, the Send detailed ASP error messages to client option should be selected. Once your application is debugged and in production mode, the other option is preferred as occasionally an error may expose critical information about your system to the end user (e.g. critical variables, or a database name of the application), especially in a situation when custom error handlers are not available. Debugging flags, as seen on the screenshot, do not need to be modified as they are usually handled by Visual InterDev. If you are using the free debugger (downloadable from the Microsoft Scripting site), you must set the flags by yourself, and start the debugger before calling any of the ASP pages.


Application Configuration

App Mappings | App Options | App Debugging

Debugging Flags
☐ Enable ASP server-side script debugging
☐ Enable ASP client-side script debugging

Script Error Messages
- [✓ Send detailed ASP error messages to client]
- Send text error message to client:

An error occurred on the server when processing the URL. Please contact the system administrator.

Enabling Error Display in IE 5.0

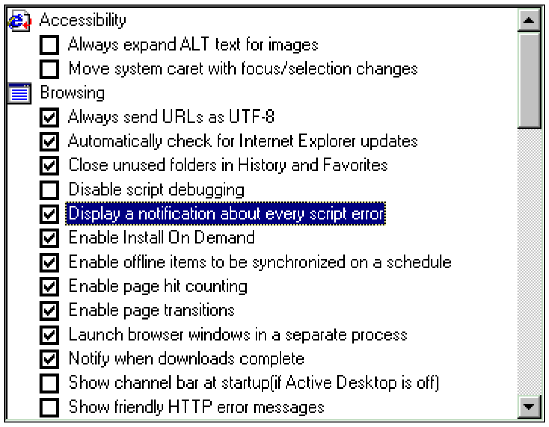
Although Internet Explorer 5.0 has an improved error display over previous versions, it has introduced several options that may cause some confusion. Essentially, there are two modes – 'debug' and 'run' – in which Internet Explorer can operate, each of which has certain quirks. The preferred mode of error display and debugging (at least, for developers in development mode) is the 'debug' mode with the use of the Script Debugger. However, when in 'production' and 'testing' modes, debugging should be disabled (see the note below). The standard (and free) script debugger may be downloaded from the Microsoft Scripting site:

`http://msdn.microsoft.com/scripting/debugger/default.htm.`

An alternative to the script debugger is the Visual InterDev application environment (which includes the script debugger), as well as the Microsoft Office 2000 element, the Microsoft Script Editor. Script debugger is also installed with Windows 2000. After the script debugger has been installed on the system, IE can display two different types of error messages, and allow the option of entering the debugger once the error has been found.

With IE 4, similar steps can be taken to disable and enable debugging. There is no option to hide error display as in IE 5.

Internet Explorer error settings are neatly kept away from the end users and sometimes can be frustrating to locate. From the Tools menu you have to choose Internet Options… and then the Advanced tab to see the advanced IE settings as seen in the figure:



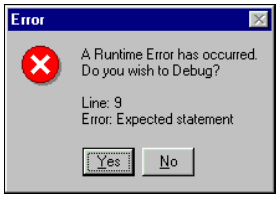
Accessibility
- [ ] Always expand ALT text for images
- [ ] Move system caret with focus/selection changes

Browsing
- [✓] Always send URLs as UTF-8
- [✓] Automatically check for Internet Explorer updates
- [✓] Close unused folders in History and Favorites
- [ ] Disable script debugging
- [✓] Display a notification about every script error
- [✓] Enable Install On Demand
- [✓] Enable offline items to be synchronized on a schedule
- [✓] Enable page hit counting
- [✓] Enable page transitions
- [✓] Launch browser windows in a separate process
- [✓] Notify when downloads complete
- [ ] Show channel bar at startup(if Active Desktop is off)
- [ ] Show friendly HTTP error messages

There are two options of interest to us: Disable script debugging and Display a notification about every script error.

With the first option selected, the debugger is disabled, and depending on the selection of the second option, either the so-called 'user friendly' error dialog box is set for the browser, or the error icon in the lower-left corner of the Internet Explorer. Although the dialog box shown below may be more *user* friendly, it may not be *programmer* friendly. The error code is not displayed (only the text of the error code – which forces you to dig through error code tables in case you would want to handle the error in code), but the line number is displayed correctly. The dialog box displayed below is for the same snippet of code as the dialog window shown in the following figure, but the line number in there is wrong (although after going into the debug mode the correct error is highlighted).

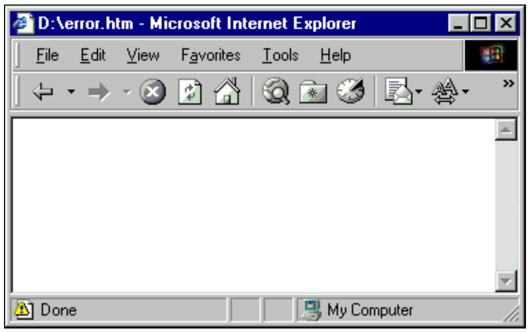
The disable script debugging option works only when the script debugger is installed on the system; and additionally the browser has to be restarted before changes to this option can take effect. Since this chapter will strictly work in developer mode, all of the IE errors will be presented in the 'debug' mode, as displayed in the figure below – do not check the Disable Script Debugging option:


Error

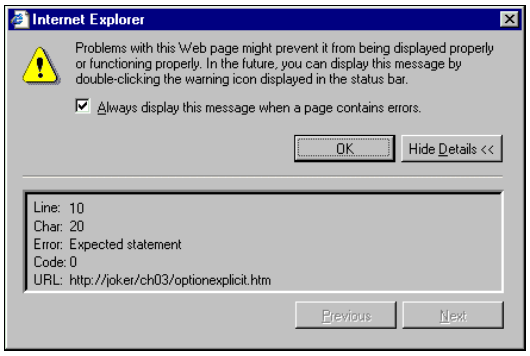
A Runtime Error has occurred.
Do you wish to Debug?

Line: 9
Error: Expected statement

The second option of interest – Display a notification about every script error – works when the debug option is de-selected (Disable script debugging), and it enables suppression of errors. When this option is cleared, an icon appears on the status bar to inform the user that an error has occurred; the error is then displayed by clicking on the icon. The yellow sign with an exclamation mark indicates that there was an error on the page, as seen in the snapshot below.



Obviously, this is the least desirable setup from the developer's point of view. However, it might be the default setup on your client's browser, which may prevent your client from reporting any unhandled errors back to you. When the Display Notification About Every Error option is checked, or the user clicks on the error icon (from the snapshot above), the following dialog will appear.


Internet Explorer

Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.

☑ Always display this message when a page contains errors.

```
Line:  10
Char:  20
Error:  Expected statement
Code: 0
URL:  http://joker/ch03/optionexplicit.htm
```

You should be aware of these subtle differences in error display, especially since this setting is in your end users' control. Your error handling mechanism may be disabled because of it, or the end user may not be able to see that an error has occurred, and be surprised that the page does not work.

Note that when not 'debugging' scripts, the Disable Script Debugging option should be selected at all times. When in debug mode, the scripting engine, upon interception of an error, automatically invokes the debugger, and prompts the user if the debugger should be opened. Although this is nice, the standard client error handlers are ignored. Even if there is an error handler capable of correcting the error it will not be invoked. It would be nice if the debugger would start only as the last resort, but this is not the case. This problem does not apply to ASP's Visual InterDev debugging options.

Other hosts are 'dumber', in a sense that errors are always displayed (with the exception of WSH 2.0 – now in beta), and debugging is not possible. Different coding and debugging strategies are discussed later in this chapter.

Different Types of Errors

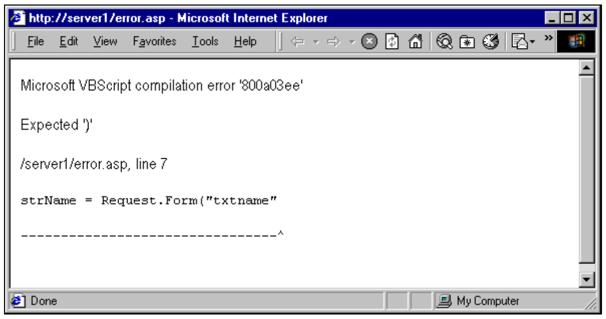
There are three types of errors that can burrow their way into your lovingly crafted VBScript (or any other scripting or programming language for that matter). The three types are not equally severe, the syntax errors will halt the execution of the script, run-time errors will invoke an error handler, and logical errors will most commonly contaminate data in your application, and often cause other run-time errors to occur.

Syntax Errors

VBScript, like all other programming or scripting languages, follows set rules for construction of statements. Before the script is run, the scripting engine parses all of the code, converting it into tokens. When an unrecognizable structure or an expression is encountered (for example, if you mistype a keyword or forget to close a loop), a syntax error is generated. Luckily, syntax errors can usually be caught during development phase, with minimal testing of the code.

In some programming environments, syntax errors are called pre-processor, compilation, or compile-time errors. If your script includes a syntax error, the script will not execute and the host immediately informs the user of an error.

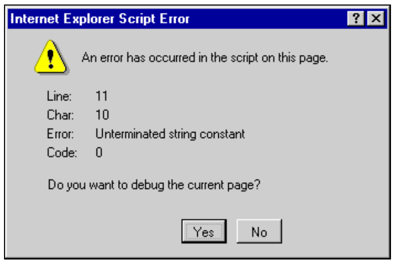
Those of you who are used to writing applications using Visual Basic will be used to having syntax errors highlighted by the interpreter in the IDE as soon as you move from the line containing the syntax error. This is a very useful feature that unfortunately is not available when using VBScript since the script is not interpreted until it is executed. What happens depends on what you are doing. If the syntax error is in a script being run at the server (as in an ASP-based application – see Chapter 14) then the error text is simply passed through and displayed in the client browser instead of the requested page, as shown in the figure below:


http://server1/error.asp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
My Computer

```
Microsoft VBScript compilation error '800a03ee'

Expected ')'

/server1/error.asp, line 7

strName = Request.Form("txtname"
--------------------------------^
```

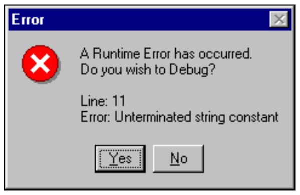
If the syntax error is in a client side script being run at the browser, the document loads but the script that contains the error prevents it from running properly.

What exactly happens depends on where in the script the error occurs. However, each time the script is run, the error message will be displayed. Here is the error message in Internet Explorer 4.0:



```
An error has occurred in the script on this page.

Line:   11
Char:   10
Error:  Unterminated string constant
Code:   0

Do you want to debug the current page?
```

Here is the same error as seen by Internet Explorer 5.0. Notice how the syntax error is confusingly referred to as a run-time error:



```
A Runtime Error has occurred.
Do you wish to Debug?

Line: 11
Error: Unterminated string constant
```

Syntax, and run-time errors are easier to spot than logic errors (which we will look at shortly) because they always result in an error message being generated. Needless to say, with proper understanding of VBScript, syntax errors are not a major concern.

Syntax errors tend to pop-up in several circumstances:

- When something is missing from the code – parentheses, keywords (especially in blocks), statement elements, or when the keywords are simply out of place.
- When a keyword is misspelled or used incorrectly.
- When you try to use a VB or VBA keyword that is not implemented by VBScript.
- When you use keywords that are not supported by the scripting engine (certain keywords may be phased out, and others added).

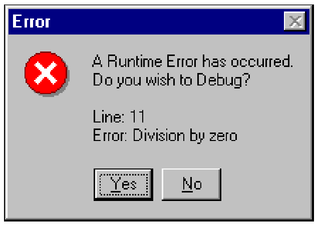
Unfortunately, VBScript does not support conditional compilation (the ability to run different code depending on environment settings). Hence, when writing code for different versions of browsers, or scripting engines, you may either have to 'know-the-version', or use JScript.

As you may expect, code executed as part of `Eval()` or `Execute` and `ExecuteGlobalstatements` is not parsed before the script is run, and can generate runtime errors (but are exempt from the `Option Explicit` rules). Special attention has to be paid when generating dynamic code. Appendix E shows all 53 of VBScript's Syntax Errors and their codes. All of these errors, with an exception of the first two – Out Of Memory and Syntax Error – are relatively easy to diagnose and correct, but all of these errors (such as Expected '(' or Expected 'If') should really be caught when the program is run the first time.

Runtime Errors

The second, and most common type of error (at least to the general public), is the runtime error. A runtime error occurs when a command attempts to perform an action that is invalid. For example, a runtime error occurs if you try to divide by zero:

```
Sub window_onload()
Ans = 200/0
Msgbox Ans
End Sub
```



A Runtime Error has occurred.
Do you wish to Debug?

Line: 11
Error: Division by zero

The various conditions that can result in runtime error depend on the language you are scripting with. A condition that might cause a runtime error in VBScript might not cause an error in JScript (for example, attempting to divide by zero in JScript doesn't generate an error). The result of a runtime error is similar to that of a syntax error – the script is halted and an error message is displayed.

Unlike the syntax errors, which pop-up when the script is loaded, runtime errors show up during script execution by the scripting engine. Runtime errors can occur as a result of bad coding (which should really be caught during the debugging and testing stage of the project), and as a result of unusual circumstances that may or may not be prevented. There are many factors that can contribute to a runtime error, all depending on the conditions under which the script is run.

The main reasons for these 'unusual circumstances' are:

- Certain security options may be turned on or off. For example, your script may try to access a component that has not been marked as "safe for scripting". In the tests you've carried out the Internet Explorer has been set to trust the component; however, during final release, the script crashes because of different security settings on client browsers.
- Components may or may not be available. Here, you might assume that a component is readily available on the client system, and not provide installation information when referencing the component. When the component is not available, the script will cause a run-time error.
- Platform differences. VBScript may be available on many platforms (including Unix, or Alpha) but the features supported by each platform may vary, especially when using external components.
- Configuration may be totally different (you should not expect an HTA based script to run 100% as an HTML based script).
- Finally, the computer might be under unusual stress. Scripts that use unusual amount of system resources (memory or CPU time, for example) may behave unexpectedly, especially when other scripts and applications contend for the same resources. Applications can often time out, and raise an error directly to the script, or, in other cases, terminate a script.

Technically, when the runtime error occurs, the script execution is stopped and the VBScript engine invokes an exception handler (it is considerably weaker in its functionality than the VB or VBA exception handler). There are several options at this point, but we will defer them to a later section – **What can we do about errors?**. The most essential error handler in question is the `On Error Resume Next` statement, which unfortunately requires a little foreknowledge into the possibility of an error occurring at the right time and at the right line of code (as you have to perform error testing immediately after the error occurs) in order to be able to use it. Internet Explorer additionally provides `window.onerror` and `element.onerror` events that can be bound to functions, which is covered in Appendix E. If no error handler is present, the error is reported back to the client.

Thus, runtime errors provide us with the possibility of taking some action. In order to correct the error in VBScript, you will need to know the decimal version of the error number (which is also provided as a hexadecimal code, for cases when VBScript throws an error, and passes it to the host): a full listing of VBScript runtime errors is provided in Appendix E. The majority of these errors (such as Division by zero or Variable is undefined), however, are simply a result of poor programming, and really should be caught during the debugging and testing stage of the project, rather than corrected by some overly complex error handler.

Non VBScript Runtime Errors

Usage of outside components and files (Automation Objects) can also be a cause of runtime errors. Although some of the errors listed below will be thrown in reference to improper usage of other components and files, you can also expect to see a lot of errors that may either be raised by the component or the operation system. For instance, the `ADODB.Recordset` component may raise the following error:

Microsoft OLE DB Provider for ODBC Drivers error '80004005'
[Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified

This is probably the most common COM failure error (which, in this case, actually has a useful description). This particular error – 80004005 (called SCODE) – is raised by a number of COM components, and sometimes contains useful information, as in the case above. Most of the time, though, you will end up scratching your head, wondering what the error message might mean. Good sources of information about errors are the appropriate documentation and Microsoft's Personal Online Support Site at `http://support.microsoft.com/support/search/`.

When trying to find out the meanings of error messages (after you realize it is not an error based in your VBScript), you may use the following list as a rule of thumb to identify a potential source of error:

8007xxxx	Windows errors (you may convert the xxxx hex code to decimal and use `net helpmsg dddd` in DOS window to find out the meaning of the error)
800Axxxx	ADO errors
80005xxx	ADSI errors

Knowledge of error codes thrown by components and windows is essential in the development of error handling functions, as the majority of error handling functions often rely on outside components.

Additionally, some components, such as ADO, contain their own Errors collection, which may expose more than a single error that occurred. In case of ADO, the Errors collection contains information pertaining to a single operation involving a given provider. You should research a given component not only for the errors it might raise through automation, but also about its internal error handling capabilities.

Problems with Option Explicit

If you come to VBScript with a good VB or VBA background, you are probably accustomed to the usage of `Option Explicit` statement as the very first line in your program. Kudos to you, but you should not expect the same behavior in VBScript. Expect a lot more work on your behalf. From now on this is a runtime error.

The `Option Explicit` statement is one of the many statements transplanted from VB into VBScript. It is particularly useful in identifying undeclared and misspelled variables, or variables that are being used beyond their scope. When a script contains the `Option Explicit` statement before any other statements, the scripting engine expects all variables to be declared explicitly by using any of the `Dim`, `Private`, `Public`, or `ReDim` statements, and only to be used within their scope (except for dynamically executed code associated with `Eval`, `Execute` and `ExecuteGlobal`). Unfortunately, unlike in VB or VBA, using `Option Explicit` causes the runtime error 500 Undeclared Variable; as you can imagine, this severely limits its usefulness when used in combination with the `On Error Resume Next` statement.

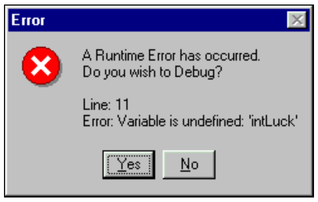
Let us demonstrate this with an example. The following code contains two undeclared variables, one that has global scope, and one within the scope of the `GetLucky()` function. The power of `Option Explicit` is easily identifiable:

```
<SCRIPT LANGUAGE=vbscript>
<!--
Option Explicit

Dim intMyNumber, intResult       ' Declare variables
intLucky = 10                     ' Undeclared variable generates syntax error
intMyNumber = 10                 ' Declared variable does not generate error
intResult = GetLucky()

Function GetLucky()
   Dim intMyNumber                  ' Declare variable local in scope
   intLuck = 3                      ' Undeclared variable: wanted to change
                                    ' intLucky - error
   intMyNumber = 4 + intLucky       ' Now have 14 instead of 7 like we wanted
   GetLucky = intMyNumber
End Function
-->
</SCRIPT>
```

After the first run, we see that `intLucky` is not declared, and we proceed to fix the error:

```
Dim intMyNumber, intResult, intLucky               ' Declare variables
```

Now, as the screenshot below shows, we find another error (an undeclared or misspelled variable), which is easy to correct. Clearly, we wanted to change the global variable, `intLucky`, and the `Option Explicit` statement helps us to identify our mistake. Without the `Option Explicit` statement at the start of the script, various mistakes of this nature are likely to pass unnoticed, causing odd or unwanted results at runtime.



A Runtime Error has occurred.
Do you wish to Debug?

Line: 11
Error: Variable is undefined: 'intLuck'

With the obvious usefulness of the `Option Explicit` statement, why should we be unhappy with it? Well, because it is a runtime error, and consequently, undeclared variables will not show up during parse stage, and its detection may even be negated by the use of `On Error` statement (with either Error being overwritten, or cleared) – something that is the opposite in the VB environment.

If the `GetLucky()` function had not been executed (some functions will not be called each time the script is run, depending on user responses), the undefined variable error would never have materialized. Secondly, it creates complications when you are creating error-handling functions. Essentially, when handling exceptions, you are expecting something more significant than an undeclared variable, in other words you are expecting a true exception, and not just a simple programming mistake.

Rarely will you try to correct this mistake, and you will probably have to consider an undeclared variable as a critical error, which should be caught early in the development stage. Unfortunately, this will throw you off because of the manner in which it will be introduced – the error may exist in a rarely accessed procedure, and the error reporting procedure may not be prepared to identify this type of error. Although error handling is discussed in more detail later in the chapter, consider a simple illustration of what might go wrong. Let us add `On Error Resume Next` – a footstep of error handling immediately after `Option Explicit` to the code above, as following:

```
Option Explicit
On Error Resume Next
```

Now, when running the script, `Option Explicit` is essentially neutralized, and the error is not easily caught. If a generic error handler were available, it would inform us that an error has occurred, but it would not tell us the line where the error occurred.

When writing an error handler, remember to provide reporting functionality for generic errors, including undefined variables. A callout label in such a handler may prove invaluable. Although you may not know the exact line number where the error occurred, at least you will be aware of its proximity.

In any way, when combining `Option Explicit` with `On Error Resume Next` you have to be extremely careful in the way you test for errors, create a scope for an exception handler, override the default exception handler, and, finally, clear the exception handler (via `On Error Goto 0`). More on Error Handling specifics is available in the Appendix E.

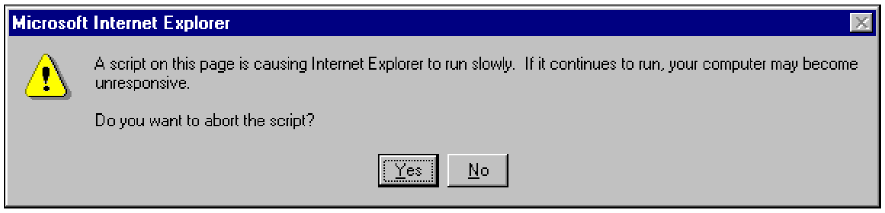
Logic Errors

Logic errors, or **bugs**, are the most difficult of all the errors to catch and track down. By their nature, these errors are caused when a valid script (no syntax, or runtime errors) produces undesirable results. For example, a script that asks for the user's password before letting them proceed, but which still allows them to proceed whether the password is correct or not, would have a logic error. Likewise a script that totaled-up an order form but which did not handle the tax right would be a logic error. A script might be designed to convert measurements from one unit to another (Fahrenheit to Celsius, for example) but if the formula is wrong, you have a logic error. In other words, VBScript will always do what you tell it to do, not what you thought or meant to tell it to do. The scripting engine will not generate an error message – your script will simply produce unexpected output; however, logic errors' side effects often include creation of other errors as well.

As always though, there are exceptions to the no-error-message rule for logic errors. This is in relation to infinite loops. For example:

```
Sub window_onload()
    Dim intX
    Do Until(intX)
        If intX < 10 Then
            inX = intX - 1
            'the above line has a mistake in the variable name
        End If
    Loop
End Sub
```

If your script contains a script that takes a long time to process then the VBScript DLL will eventually time out and display the following error message:



Microsoft Internet Explorer

A script on this page is causing Internet Explorer to run slowly. If it continues to run, your computer may become unresponsive.

Do you want to abort the script?

This allows you to stop the script before the system becomes unstable. However, it does not provide you with any clues as to what or where the error is.

Identification of logic errors is beyond the scope of this chapter. The most common types of errors will include bad calculation formulas, incorrect usage of operators, improper rounding, and generally problems with conditional statements, loops, and general lack of validation of data. The only way to reduce the occurrence of these is through full testing of borderline outcomes. There are testing tools, such as Visual Test, which will simplify repetitive testing processes (including regression testing), and the debugger (available with IE, or Visual InterDev), which will help you step through the code, look at the contents of variables, and the calling stack. In a proper test you will be required to feed the script a lot of data (good, borderline, and bad) and compare the output against the output you have calculated (or figured out) manually. Some tips on testing are:

- Check, double check, then recheck again any formulae you have used in your script, to make sure that they return the correct results.
- Work out the results that you expect – try out all the different combinations.
- Consider how the user might impact a calculation by, for example, entering zero or a negative number – does the script cope with this?
- Check that the knock-on effects of any actions are there – for example, if a customer deletes an item from their order, be sure to check that the item is removed AND the order total changed.
- Do not just check things to see if they work, also check what happens under circumstances where you know they should not work.

Only careful testing can help you spot logic errors in your projects.

Unfortunately, there are no other good techniques for catching logic errors. VBScript does not support anything like `Debug.Assert` which is found in its parent languages, and even though you might create an object with similar functionality on your own, you will also have to remove the additional code during the release stage on your own (this is not the case with VB and VBA). There are some guidelines we can follow:

- Testing (as mentioned above) is essential to eliminate logic and runtime errors.
- Use encapsulation within VBScript classes to reduce the chances of logic errors occurring.
- Whenever you can, re-use old code that has been thoroughly tested and that you know from experience works (one may say that the only good code is old code, which is crazy considering that the Internet reinvents itself every few months).
- Always adhere to coding conventions – these increase the overall clarity of your code.
- Adoption of good programming practices, particularly at design time, dramatically reduces the complexity of your code.

The only marginally practical technique is to treat possible logic errors as runtime errors, by raising an error. By testing and validating the critical values internally in the key subroutines and functions (at least, checking the input parameters), you may be able to find areas in which your code is producing an undesirable output. When you find that data is not valid within a certain predefined range, you may raise an error, and break execution within that procedure. This will, unfortunately, only cover a small percentage of logic errors; we re-emphasize that only a stringent testing method can identify all of the logic errors within your script.

Finally, logic errors are sometimes a by-product of a high degree of complexity. Proper encapsulation, variable scoping, and use of VBScript classes will undoubtedly reduce the likelihood of logic errors occurring. Following this to the extreme, the best approach is to simply re-use old and trusted code, whether by use of includes (in HTML and ASP) or through the use of various components (`.wsc`, `.htc`, `.dll`, `.ocx`).

What Can We Do About Errors?

There are two things we can do with an error:

- Get rid of it completely
- Handle it

Because it isn't possible to make a script completely bomb-proof (since errors can be caused not only by mistakes in the script itself but also by actions taken by the user), there is a real need to implement a method by which errors are dealt with more effectively than simply flashing the error message dialog box at the user.

Remember that to most users the error messages will be incomprehensible.

We will look at how we get rid of errors later in this chapter, when we come to debugging, but for now let us look at what is meant by handling errors and how we go about doing it. Also take a look at Appendix E, which includes complete syntax, and many examples of error handling.

Handling Errors

The process of error handling involves detecting the error as it occurs and dealing with it effectively. How we choose to deal with errors depends on the type of error, what caused it and the consequences resulting from it. The simplest thing we can do with an error is ignore it and to do this we use the `On Error Resume Next` statement.

On Error Resume Next

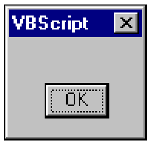
The `On Error` statement enables error handling in the script that we are writing. The only thing that we can do with the `On Error` statement in VBScript is to `Resume Next`. What this means is that an error in the script in any procedure, instead of being fatal and causing the script execution to stop, is overlooked and the execution continues with the next statement following the error or with the statement following the most recent call out of the procedure containing the `On Error Resume Next` statement. In other words:

`On Error Resume Next` is the VBScript equivalent of telling the interpreter to ignore any errors and carry on regardless!

The `On Error Resume Next` statement must come **before** any statements in the procedure you want it to protect. So for instance the following snippet of script, where we divide by zero, will not generate an error:

```
Sub window_onload()
On Error Resume Next
     x = 3/0
     Msgbox x
End Sub
```

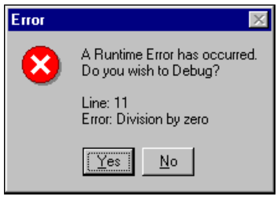
It will simply resume execution of the script, in this case, by displaying a message box with a meaningless result:



However, if we place the statement after the error, we lose all the protection that it offers us:

```
Sub window_onload()
     x = 3/0
     Msgbox x
On Error Resume Next
End Sub
```

This time the error is handled by the host, and the message is generated as normal:



A Runtime Error has occurred.
Do you wish to Debug?

Line: 11
Error: Division by zero

This statement might seem to be all we need to know for effective error handling – it isn't. This is because it is really the error-handling equivalent of brushing dirt underneath the carpet - sure, you don't see it, but the result isn't really ideal. Using it can lead to some odd results, as the divide by zero example above shows. There are few scripts that can be expected to function properly after one line has been ignored because of an error: usually, this will cause another error further down the line.

Remember that when using the `On Error Resume Next` statement that the error has still occurred. All it has done is hidden the standard error message response. While it is useful at times to include the `On Error Resume Next` statement in code, a much better way of dealing with errors is to actually *handle* them. To do that we use the `Err` object.

Err Object

The `Err` object holds information about the last error that occurred. It is a feature that is available for use at all levels of your script and there is no need to create an instance of it in your code as it is an intrinsic object with global scope (see Appendix E for a more detailed description). This object has five properties and two methods.

Err Object Properties

Property	Comment
`Description`	Sets or returns a descriptive string associated with an error.
`HelpContext`	Sets or returns a context ID for a topic in a help file.
`HelpFile`	Sets or returns a fully qualified path to a help file.
`Number`	Sets or returns a numeric value specifying an error – this is the `Err` object's default property. It can be used by automation objects (ActiveX) to return a SCODE (status code).
`Source`	Sets or returns the name of the object or application that originally generated the error.

Err Object Methods

Method	Comment
`Clear`	Clears all property settings of the `Err` object.
`Raise`	Used to generate a runtime error.

Using the Err Object

Let's look at how we can use the `Description`, `Number` and `Source` properties, and the `Clear` and `Raise` methods of the `Err` object. The other properties refer to custom help files that can be created for specific errors that the user might come across.

The first thing to remember about using `Err` to handle errors is that you need to have `On Error Resume Next` set before hand; otherwise, the script execution will be cut short and your error handling script will be wasted!

```
…
On Error Resume Next
…
```

Now we can set to work handling the error our way. The first thing to do is to generate an error, and to do this we could simply write a script with a deliberate error in it. However, we have no need as VBScript provides a way to generate errors on demand – the `Raise` method. Using this method we can generate any error we want, with just one line. All we need to know is the number of the error (given in Appendix E) that we wish to create.

So, if we want to generate an overflow error, for example, we raise error number 6:

```
On Error Resume Next
Err.Raise 6
```

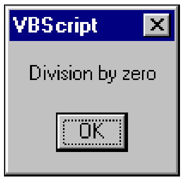
Or, for a custom error, we can use `vbObjectError` constant. The programmer can define error numbers above this constant to create and handle errors specific to the script.

```
On Error Resume Next
Err.Raise vbObjectError + 1, "something is wrong", "Custom Error"
```

If you want to see the error messages generated by these, simply comment out the `On Error Resume Next` statement, or create a procedure to display the error.

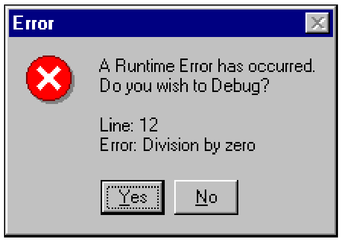
Now we have our error, let's look at how we can handle it. The property to use is the `Description` property. This is used to set or return a textual description of the error. If we use the default description, we simply get the standard error message. For example, here is our error-handled divide by zero:

```
On Error Resume Next
Err.Raise 11
MsgBox (Err.Description)
```



```
VBScript
Division by zero
```

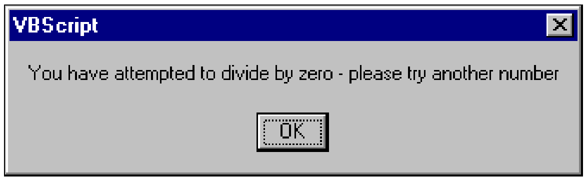
Here is how it would be unhandled:



```
Error
A Runtime Error has occurred.
Do you wish to Debug?

Line: 12
Error: Division by zero
```

Not much of an improvement, is it? However, we can create a message that is a little more meaningful:

```
On Error Resume Next
Err.Raise 11
Err.Description = "You have attempted to divide by zero " _
   & "- please try another number"
MsgBox (Err.Description)
```

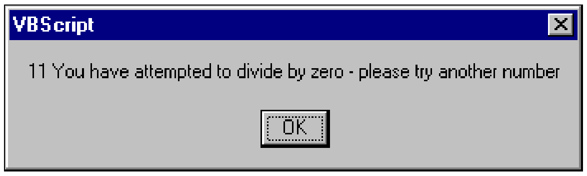

VBScript

You have attempted to divide by zero - please try another number

This example is preferable because it gives the user a clear and unambiguous explanation of what has happened and what they need to do next.

We can do the same thing with the error number, this time using the `Number` property:

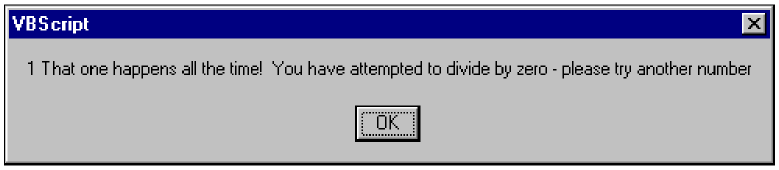
```
On Error Resume Next
Err.Raise 11
Err.Description = "You have attempted to divide by zero " _
   & "- please try another number"
MsgBox (Err.Number & " " & Err.Description)
```


VBScript

11 You have attempted to divide by zero - please try another number

This property also allows us to set or return our own number to an error (setting your own number might be useful if you want to include an easy to use guide with your VBScript project). This is not the best way in which user-defined errors can be created, it is more advisable to use the `vbObjectError` constant, this is explained in Appendix E:

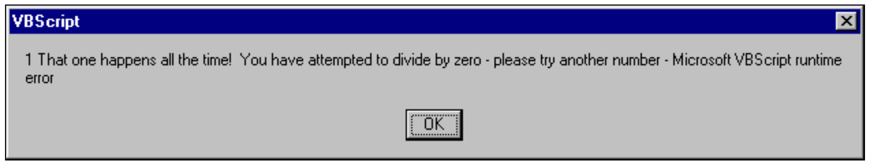
```
On Error Resume Next
Err.Raise 11
Err.Number = 1
Err.Description = "You have attempted to divide by zero " _
   & "- please try another number"
MsgBox (Err.Number & " " & Err.Description)
```


VBScript

1 That one happens all the time! You have attempted to divide by zero - please try another number

If we want to know what generated the error we can use the `Source` property:

```
On Error Resume Next
Err.Raise 11
Err.Number = 1
Err.Description = "You have attempted to divide by zero " _
   & "- please try another number"
MsgBox (Err.Number & " " & Err.Description & " - " & Err.Source)
```



VBScript

1 That one happens all the time! You have attempted to divide by zero - please try another number - Microsoft VBScript runtime error

Using `Source` is helpful in tracking down errors when using VBScript to automate Microsoft Office tasks. For example, if using script to access Microsoft Excel, and it generates a division-by-zero error, Microsoft Excel sets `Err.Number` to its own error code for that error, and sets `Source` to `Excel.Application`. Note that if the error is generated in another object called by Microsoft Excel, Excel intercepts that error and re-sets `Err.Number` to its own code for division by zero. It does, however, leave the other `Err` object properties (including `Source`) as set by the object that generated the error.

Once the error is handled, we want to get rid of it completely. To do this, we use the `Clear` method:

```
On Error Resume Next
Err.Raise 11
Err.Number = 1
Err.Description = "That one happens all the time!" & _
      "You have attempted to divide by zero - please try another number"
MsgBox (Err.Number & " " & Err.Description & " - " & Err.Source)
Err.Clear
```

`Clear` is used explicitly to clear the `Err` object after an error has been handled. VBScript calls the `Clear` method automatically whenever any of the following statements are executed:

- `On Error Resume Next`
- `On Error Goto 0`
- `Exit Sub`
- `Exit Function`

Remember to remove any lines in your script that raise errors when you have finished testing your error-handling code!

Remember that errors are like aches and pains - they point to something being wrong, either with the script itself or with the way it is being used. There is a tendency to think that, given all the power that VBScript has to offer, we should try to fix these problems 'on the fly'. So if someone divides by zero, it's easy to think that you could simply use VBScript to put another number into the sum. The danger here is that you create more problems in trying to 'fix' it, and this can lull the user into the false sense of security that everything is OK when it isn't. Only attempt this kind of error handling when you can be **absolutely sure** you know what the problem is.

A good alternative to using message boxes is to create custom help files and refer to these using the `Err` object properties `HelpContext` and `HelpFile`. These allow us to point to specific entries in a custom help file created for the project in question - giving the whole project a professional and polished feel.

For a project of any size, it is useful to log any errors that occur so that they can be studied later. This is particularly useful for large ASP-based projects, where the error might lurk otherwise undetected – although aggravated users can often points these out to you!

Handling Errors

So far we have identified the syntax and the simple techniques associated with error handling. Obviously, we cannot ever hope that errors "will just not happen", and even if it were possible to eliminate all of the errors from the code (through very defensive programming), the cost of developing such software would probably be quite prohibitive.

Thou shall not underestimate the importance of error handling. Something *will* go wrong—will your program handle it gracefully when it does? A program can never be considered professionally done without a well thought-out and consistent error-handling scheme.

By now, based on the examples shown previously, we know that we can handle errors in three different ways:

- Ignore the errors altogether (the script stops), and allow the default error handler provided by the host to deal with the error.
- Try to intercept errors in-line, immediately after a suspect operation that could create an error.
- Push the error up the call stack, and create either generic error handlers, or procedure specific handlers that can anticipate the problems arising from the procedure.

If you are not familiar with the term "call stack", imagine that as each function or sub is called, it is placed on top of a stack. When a procedure calls another procedure (or even itself), the second procedure is placed on top of the stack. If the second procedure does not have an error handler and an error occurs (or `Err.Raise` is used), the error is pushed "up the call stack". The remaining piece of the second procedure is ignored, and the first procedure has a chance to handle the error. Since procedures are often nested, you can easily control errors by placing error handling routines in key procedures. You have to be aware that certain statements will reset the `Err` object, and your error handling has to come before that. Please see Appendix E for examples of using the calling stack to handle errors.

Note that it is also a good idea to have a bottom-line, generic error handler available at all times. More often then not, the error handler will be written with a specific purpose in mind – checking whether a file exists, or whether an SQL string executed correctly. In such circumstances, there can be other errors that we have not accounted for – undeclared variables, bad parameters, etc. These should be either passed on up the call stack, by raising a custom error, or passed on to another, more generic procedure.

So, what can be done, after an error is intercepted? Perhaps the sky is the limit, and only creativity and limited time budget will prevent you from treating the error the way you want it treated – in other words fixed. There are no out-of-the box solutions here, only loose guidelines. The simplest thing to do is obviously to display the error in the most meaningful way. As you go on, you should try to log the problem (if script is running unattended), or at least provide a simple facility for the end user to report the problem. Going further, you may try to fix the problem on the fly – perhaps it is just a simple exception (such as an out-of-bound array call), or a user error that can be retried. Then, if you can't fix it, gracefully fall back on the user-friendly error message, and log the error. More often than not, errors that cannot be easily handled will expose the weakness of your program, rather than a configuration problem that prevents the program from executing. Make the first few users your test subjects if you cannot test all of the exceptional permutations personally.

When writing an error handler, make sure it is bug proof. Test it more than any other procedure, preferably with the use of home-built test suites in order to see how it behaves with different data (either raise errors, or call it with simulated data), and in under different circumstances. Errors that are not found in development (computer low on memory, lack of appropriate permissions, etc) will unfortunately rear their ugly heads in production. Cross-application interactions as well as an increased user load on an IIS server may effectively disable some of the poorly written error handling procedures. It is also a good idea (or even standard practice) to get *someone else* to test it as thoroughly as possible as well.

Step #1: Diagnose What Went Wrong

Error diagnosis is obviously a large part of error handling and, unfortunately, there is no easy way to jump into error handling without making sacrifices. There are just too many error codes in VBScript alone for us to write code that will anticipate all of the possible errors, never mind writing code for all of the possible errors caused by outside components. The common technique is to debug early for the most common errors (bad parameters, undeclared variables, etc.) and write your error handling function around only those errors that you are anticipating.

For instance, working with ADSI (one of the common components automated by VBScript), we can pull out the most prevalent ADSI errors and put them into a common error handler, which may be invoked whenever an error is diagnosed. This may even happen when your script is executing correctly. For instance, if we want to add a new user to a domain, with a username that already exists, it will be less expensive in terms of programming and computer resources to check for an error when adding a new user rather than attempt to find out if the user exists.

The code below performs a select case against the `Number` property of the `Err` object, allowing the programmer to decide what happens when a given error occurs. Due to the number of possible errors, the listing is edited for brevity's sake; the snippet also adjusts for the poor error descriptions of ADSI:

```
<%
Sub adsiErr()
    Dim blnIsErrorFixed
    blnIsErrorFixed = False
    Select case Err.Number
        case &H80005000:                       ' Invalid ADSI pathname
            blnIsErrorFixed = fixErrorPath()
            case &H80005001:                   ' Unknown Domain Object
            call logError("Unknown Domain Object")
            call displayError("Unknown Domain Object")
            Err.Clear
' Bunch of case statements deleted, see real file
        case &H80004005:                       ' now the ambiguous COM Error
            call logError()
            call displayError()
            Err.Clear
        case &H800708B0:                       ' Unable to add, User Exists
            blnIsErrorFixed = fixUserExistsError()
        case else:                             ' unaccounted error, log it,
                                               ' display it
            call logError()
            call displayError()
            Err.Clear
    End Select
    If Not blnIsErrorFixed Then Response.End
End Sub
%>
```

This semi-generic error-handling procedure is sufficient to cover the majority of errors that can be attributed to ADSI. It can be called in-line, as well as after a procedure call – the code below is slightly edited:

```
Option Explicit
Dim objComputer, objGroup, strGroupName
On Error Resume Next

' Get object for computer, call error handler inline
Set objComputer = GetObject("WinNT://" & Request.Form("DomainName"))
If Err Then adsiErr()

strGroupName = Request.Form("GroupName”)

' Create the New Group, call error handler afterwards
Call createNewGroup( objComputer, objGroup, "group", strGroupName )
If Err Then adsiErr()
```

Regardless of whether or not the error handling routine is generic, the same principles will always apply, except when we're only interested in displaying and logging the error (where we would just use `case else:` from the previous code). The error identification template will always be the same, but with a specific error the template may be slightly smaller – and you may use a less generic function. For instance, because – after the call to the `createNewGroup()` subroutine – we were only expecting an Unable to Add, User Exists error (because we were already able to establish a connection with the domain) we could have automatically called `fixUserExistsError()` as it was the most likely error to occur.

Step #2: Attempt to Correct the Error

After you have identified the error, you should obviously attempt to fix it, if possible, if not, you may just follow the next two steps. In some circumstances, the error will be a result of a user action, or input. Since VBScript is commonly found in ASP type applications, the most common errors lie in the database or file handling, as a direct response to user interaction. We'll look at a detailed database and a COM object example at the end of the section. Here, this code allows the user to correct the error. In case of potential user errors, the best approach is to validate the data that will be used by the other components.

The code below tests if a string entered into an HTML form is a date. If the string entered is not a date, the procedure throws an error, and for practical purposes, invalidates the form, and displays a simple error message:

```
<%@ Language=VBScript %>
<%
Option Explicit
Dim strDate, strError, datDate, blnError, blnCanContinue
blnError = False
blnCanContinue = False
strDate = ""
strError = ""

Sub HandleError()                           ' this will handle Error string
    strError = "<font color=red><b>" & Err.description & "</b></font>"
    blnError = True
End Sub

Sub CheckDate                               ' Sub that checks the date
    strDate = Request.Form("strDate")
    If Not IsDate(strDate) Then Err.Raise vbObjectError + 1, , _
          "Not a Date<br>"
    datDate = CDate(strDate)
    blnCanContinue = True
End Sub

If Request.Form("strDate").Count = 1 Then  ' form was entered
    On Error Resume Next
    CheckDate
    If (Err.Number > vbObjectError) Then HandleError
End If

%>
```

```
<HTML>
<HEAD>
<TITLE>Try Again</title>
</HEAD>
<BODY>
<% If blnCanContinue = False Then
    If blnError = True Then Response.Write strError
%>
<form action="tryagain.asp" method="POST">
Enter a date: <INPUT type="text" id=strDate name=strDate value="<% = strDate
%>"><br>
</form>
<% Else %>
Date is OK: <% = strDate %>
<% End If %>
<P>&nbsp;</P>

</BODY>
</HTML>
```

Correction of run-time errors can be extremely difficult and is not really recommended – perhaps it is some other part of the script creating the error, and attempts at correcting it will cause more problems. As a rule of thumb, you should establish default values for critical variables, and check the validity of the variables used by procedures. When the variable is out of valid range, substitute it with the default value.

When attempting to correct the error you should think hard whether you can indeed fix it. Chances are that if you can anticipate it, you should be able to fix it. Perhaps a database server may be down, and you may be able to "switch" to a backup server, maybe user entered backward slashes "\" in a URL textbox instead of forward slashes "/", or simply an array is too small, and you might have to `ReDim` it. Usually, it is the unanticipated error that cannot be fixed with a backup plan.

Step #3: Come Up with a User-Friendly Error Message

A user-friendly error message goes a long way to show that you at least care a little bit. There is nothing more annoying than the default error message provided by the host. Not only is it more confusing to the user, but also offers no recourse of action. A user-friendly error message can contain some of the following information:

- An apology
- A plea to report the error, along with some nifty report form (or log the error, if possible)
- A more understandable explanation of the error
- Steps that the user can take to recover from the error

Obviously, the error message, as well as, any reporting utility will depend on the host and the nature of the error. With IE, it is fairly easy to create a new window with a form that would include an error reporting mechanism (shown in the code below). Other errors will require similar techniques, and may even include auto reporting via a logging mechanism.

```
<script language=VBScript>
Function onErrorHandler(message,url,line)
   dim strHTML, objWindow
   strHTML = "<HTML><HEAD>" & vbCrLf
   strHTML = strHTML & "<TITLE>An error has occurred!</TITLE></HEAD><BODY>" _
      & vbCrLf
   strHTML = strHTML & "<FONT FACE='sans-serif'>" _
      & "<FONT COLOR=darkred SIZE=+1>" & vbCrLf
   strHTML = strHTML & "<B>We are sorry!</B></FONT>" & vbCrLf
   strHTML = strHTML & "<FONT SIZE=-1><BR>Something went wrong " _
      & "while processing this page."
   strHTML = strHTML & "<P>To help the web administrator " _
      & "identify the problem," & vbCrLf
   strHTML = strHTML & "please provide a brief explanation of " _
      & "how the error occurred,"
   strHTML = strHTML & "and press the submit error button below. " _
      & "This will help us"
   strHTML = strHTML & "identify and fix the error." & vbCrLf
   strHTML = strHTML & "<FORM ACTION=""mailto:bugs@wrox.com"">" & vbCrLf
   strHTML = strHTML & "<Error Description:<BR><TEXTAREA NAME=desc ROWS=5"
   strHTML = strHTML & " COLS=30></TEXTAREA>" & vbCrLf
   strHTML = strHTML & "<INPUT TYPE=hidden name=error VALUE=""" _
      & message & """>" & vbCrLf
   strHTML = strHTML & "<INPUT TYPE=hidden name=file VALUE=""" _
      & url & """>" & vbCrLf
   strHTML = strHTML & "<INPUT TYPE=hidden name=line VALUE=""" _
      & line & """>" & vbCrLf
   strHTML = strHTML & "<P><INPUT TYPE=SUBMIT " _
      & "VALUE=""Submit Error Information"">" & vbCrLf
   strHTML = strHTML & "</FORM></FONT></FONT></BODY></HTML>"
   set objWindow = window.open("")
   objWindow.document.body.innerHTML= strHTML
   onErrorHandler = true
End Function

Set window.onerror = GetRef("onErrorHandler")
</script>
```

The code listing above is essentially suited to a fatal DHTML error, where script continuation may prove impossible. Other hosts will use a little variation on the theme above. A similar approach should be used in ASP, with an exception of automatic logging of the error, a few changes in an error message, and changes in the last few lines:

```
     Response.Clear
     Response.Write strHTML
     Response.End
```

Other hosts may require a simple use of a `MsgBox` function, and logging of the error. The baseline attempt at displaying the error should contain the vital information. The following code function can be used to return information for errors that do not have a custom display. It can be used with practically any host, as the returning string can either be sent to the browser or another text handler.

```
Const cHTML = 1
Const cString = 2

Function UnknownError(intOutputConst)
   If Err = 0 Then UnknownError = ""
   Dim strOutput
   strOutput = ""
   If intOutputConstant = cHTML Then
     strOutput = strOutput & "<font name='sans-serif' color=red>"
     strOutput = strOutput & "<b>An Error Has Occurred</b><br>"
     strOutput = strOutput & "Error Number= #" & Err.number & "<br>"
     strOutput = strOutput & "Error Descr: " & Err.description & "<br>"
     strOutput = strOutput & "Error Source: " & Err.source & "<br>"
     strOutput = strOutput & "</font>" & vbCrLf
   Else
     strOutput = strOutput & "An Error Has Occurred" & vbCrLf & vbCrLf
     strOutput = strOutput & "Error Number= #" & Err.number & vbCrLf
     strOutput = strOutput & "Error Descr: " & Err.description & vbCrLf
     strOutput = strOutput & "Error Source: " & Err.source & vbCrLf
     strOutput = strOutput & vbCrLf
   End If
   Err.Clear
   UnknownError = strOutput
End Function
```

Step #4: Attempt to Log the Error

Contrary to popular opinion, error logging is actually easy to accomplish. There are several different ways in which it can be achieved and you may log to: the Windows log, a database, a file, or in some circumstances, via email. When the severity of an error is high (say, a hard drive failure), you should not just log the error (hoping that some day, someone will read it), but forward it to the operator or system administrator – email, SMS page, and netsend are just few of the possibilities. Under best circumstances, you could simply log the error, and the log monitoring software could decide about the severity of error, and appropriately relay the message to an available human operator.

When logging an error to a database, file or an e-mail, you can choose what information to include in the error log on top of the default information about the standard error information. Common information entities, which can be included, are:

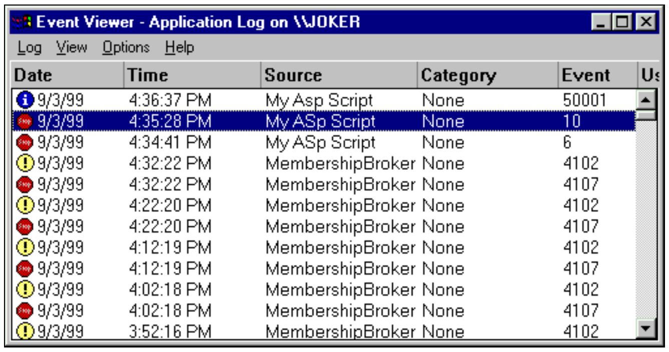
- Date and time of the error
- File or application that created the error
- Scripting Engine information
- Account under which user is executing the script
- Key variables used by the script (a mini core dump)

Obviously, with the number of additional variables, you might end up building a fully-fledged help desk system, along with the tools to analyze the wealth of errors.

Instead of duplicating the article, you can download the source code for the logging component, compile it, and use it, simply by looking at the `WroxLogGroup.vbg` Visual Basic project group in the support files for Chapter 4. Usage of the component is fairly simple:

```
Const cError = 1          ' define log constants
Const cWarning = 2
Const cInformation = 4

Sub LogError(intErrorType)
   Dim oEvent
   Set oEvent = Server.CreateObject("WroxLog.Event")
   oEvent.Application = "My ASP Script Name"
   oEvent.Description = Err.Description
   oEvent.EventID = Err.Number
   oEvent.LogType = intErrorType
   oEvent.WriteEvent
   Set oEvent = Nothing
   Err.Clear
End Sub

' Now Fake a call to the Sub
On Error Resume Next
Err.Raise 6
If Err Then Call LogError(cError)
```

Windows NT Log provides a neat summary of all errors that occurred on the computer, and include date, time, application name (source) and error ID. When the user double-clicks on the error, more detailed information, including error description (string insertion in our primitive case) is presented (although for a full, user friendly description, error IDs and their descriptions would have to be added to the registry).



Date	Time	Source	Category	Event	Us
ℹ 9/3/99	4:36:37 PM	My Asp Script	None	50001	
⛔ 9/3/99	4:35:28 PM	My ASp Script	None	10	
⛔ 9/3/99	4:34:41 PM	My ASp Script	None	6	
⚠ 9/3/99	4:32:22 PM	MembershipBroker	None	4102	
⛔ 9/3/99	4:32:22 PM	MembershipBroker	None	4107	
⚠ 9/3/99	4:22:20 PM	MembershipBroker	None	4102	
⛔ 9/3/99	4:22:20 PM	MembershipBroker	None	4107	
⚠ 9/3/99	4:12:19 PM	MembershipBroker	None	4102	
⛔ 9/3/99	4:12:19 PM	MembershipBroker	None	4107	
⚠ 9/3/99	4:02:18 PM	MembershipBroker	None	4102	
⛔ 9/3/99	4:02:18 PM	MembershipBroker	None	4107	
⚠ 9/3/99	3:52:16 PM	MembershipBroker	None	4102	

The last option is to use the Windows Script Host `LogEvent` method of the `WshShell` object. Error logging via WHS 2.0 (covered in Chapter 10) is simple (it can be done from any host except for IE), with the only drawback being the inability to change the source of the error, and the event ID (error number) – all of this data has to be included in the error description itself. Here is an ASP based sample, which can be used with the `UnknownError` function shown in the last snippet of code in **Step #3**:

```
Set WshShell = Server.CreateObject("WScript.Shell")
WshShell.LogEvent 1, UnknownError(cString)
```

The `LogEvent` method will use the same constants as shown in the code above. They are standard constants for writing to the NT log. Depending on the constant used, you will be able to identify errors through the NT log either visually (different icons), or by searching for particular errors:

Value	Description
0	Success
1	Error
2	Warning
4	Information
8	Audit Success
16	Audit Failure

Be More Aggressive with Reporting and Testing

Script debugging is an increasingly popular testing technique thanks to a fairly robust debugger included with Microsoft Visual InterDev, IE and Office 2000. Still, the process of starting the debugger (without even mentioning the horrors of installation), and stepping through the code may take the joy out of identifying errors. Often, you might create your own reporting functions, in order to speed up the process of testing.

General Environment Check-Up

The environment on which the script is deployed may be different from the development environment. Therefore, before you attempt to test the waters in real life, you should ensure that everything works, based on your own development platform. The following function checks the basics for you:

```
Function EnvironmentTest(sPad, blnShowServer)
   Dim strReport, oConn
   strReport = "Environment Report" & sPad
   strReport = strReport & "Scripting engine=" & ScriptEngine() & sPad
   strReport = strReport & "Buildversion = " & ScriptEngineBuildVersion() _
      & sPad
   strReport = strReport & "Majorversion = " & ScriptEngineMajorVersion() _
      & sPad
   strReport = strReport & "Minorversion = " & ScriptEngineMinorVersion() _
      & sPad

   strReport = strReport & sPad
   set oConn = Server.Createobject("ADODB.Connection")
   strReport = strReport & "ADO version = "
   strReport = strReport & oConn.version & sPad
   set oConn = Nothing
   If blnShowServer Then
      strReport = strReport & sPad
      strReport = strReport & "Server Software ="
      strReport = strReport & Request.Servervariables("server_software") _
         & sPad

      strReport = strReport &"Script Timeout = " & Server.ScriptTimeout _
         & " seconds" & sPad
      strReport = strReport & "Session Timeout = " & Session.Timeout _
         & " minutes" & sPad
   End If
   EnvironmentTest = strReport
End Function

Response.Write EnvironmentTest("<br>", True)
```

ADO Error Report

ADO always seems to create odd errors whenever you least expect it, perhaps because there are so many differences between providers. The function below alleviates the problem of trying to figure out what went wrong. This is probably the most useful reporting function, especially when working with a database application. As you attempt to carry out some dynamic SQL building, more often than not you discover that something is seriously wrong. The following function produces a neat report:

```
Function ErrorADOReport(strMsg, oConn, strSQL, sPad)
   ' produce a meaningful error report for an ADO connection object
   ' display title - strMsg, sql used - strSQL, and use different pad sPad
   Dim intErrors, i, strError
   strError = "Report for: " & strMsg & sPad & sPad
   intErrors = oConn.Errors.Count
   If intErrors = 0 Then
       ErrorADOReport = strError & "- no Errors" & sPad
       Exit Function
   End If
   strError = strError & "ADO Reports these Database Error(s) executing:" _
      & sPad
   strError = strError & strSQL & sPad
   For i = 0 To intErrors- 1
       strError = strError & "Err #" & oConn.errors(i).number
       strError = strError & " Descr:" & oConn.errors(i).description & sPad
   Next
   strError = strError & sPad
   ErrorADOReport = strError
End Function
```

This function simply looks at the errors collection of the ADO connection object to enumerate through all of the errors in the collection. The function can be used from other hosts, by passing a different line terminator, or “pad”, as one of the arguments in order to achieve the appropriate formatting.

To continue with the listing, the following snippet of code shows how the function is called, and displays the results, by simulating an error in the SQL statement:

```
On Error Resume Next
Set objConn = Server.CreateObject("ADODB.Connection")
objConn.Open "DSN=pubs;uid=sa;pwd="
strSQL = "select * from authors where fafa < a"
Set objRS = Server.CreateObject("ADODB.Recordset")
objRS.Open strSQL, objConn

Response.Write ErrorADOReport("open authors table", objConn, strSQL, "<br>")
```

The results of the function clearly show what went wrong, displaying the SQL statement in question, as well as all of the errors associated with it (some people attempt to debug SQL statements without even dumping the SQL statement):

```
Report for: open authors table

ADO Reports these Database Error(s) executing:
select * from authors where fafa < a
Err #-2147217900 Descr:[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid
column name 'fafa'.
Err #-2147217900 Descr:[Microsoft][ODBC SQL Server Driver][SQL Server]Invalid
column name 'a'.
```

COM Components

Another common script breaker is the failure of COM components referenced in the script. In order to test whether the components can be opened, you may create a mini-test studio that will attempt to create components, and if the component cannot be created, display the error. Changes in server configuration and installation of other components are frequent culprits of these errors. Your application may be working one day, but all of a sudden, it throws a number of errors:

```
Dim oDict, oTmp, strItem
Set oDict = Server.CreateObject("Scripting.Dictionary")
oDict.Add 1, "adodb.recordset"
oDict.Add 2, "adodb.connection"
oDict.Add 3, "adodb.command"
oDict.Add 4, "SoftArtisans.FileUp"
oDict.Add 5, "SoftArtisans.SACheck"
oDict.Add 6, "scripting.filesystemobject"
oDict.Add 7, "cdonts.newmail"

For Each strItem In oDict.Items
   On Error Resume Next
   Set oTmp = Server.CreateObject(strItem)
   If Err Then
      Response.Write strItem & " - failed. Error #" & Err.number _
         & " - " & Err.description & "<br>"
   Else
       Response.Write strItem & " - success<br>"
   End If
   Err.Clear
   oTmp = Nothing
Next
```

Similar component testing script can be developed for WSH by changing line breaks, output mechanism, and by changing `Server.CreateObject` to `Wscript.CreateObject`. Here is a sample output created by the script:

```
adodb.recordset - success
adodb.connection - success
adodb.command - success
SoftArtisans.FileUp - failed. Error #-2147319779 - 006~ASP 0177~Server.CreateObject
Failed~Library not registered.
SoftArtisans.SACheck - success
scripting.filesystemobject - success
cdonts.newmail - success
```

Defensive Programming

Probably the best way to prevent bugs is though defensive programming, combined with proper testing. Errors tend to occur as the complexity of the program increases. Unfortunately, full coverage of defensive programming is a topic for an entire book, not a sub-section of the chapter (see Code Complete by Steve McConnell, Microsoft Press, 1993 or Bug Proofing Visual Basic by Rod Stephens, John Wiley and Sons, 1998), or just stick to the following rules of thumb:

- Stick to a proper naming scheme.
- Validate data types using IsXXXX functions, such as IsDate, IsNumeric or IsObject, and create your own data validation functions such as IsEmail, IsCCNumber to make sure your procedures can actually handle the data.
- Use constants, not magic variables.
- Limit the scope of variables, objects and errors.
- Don't use clever programming when something obvious might suffice, even if it takes more programming.
- Reuse as much “stable” code as possible through includes, and components.
- Use parenthesis with complex expressions.
- Watch out for use of & and +.
- Watch out for variable scope.
- Watch out for array size.
- Declare and initialize your variables and objects.
- Watch out for endless loops.
- Encapsulate as much code as possible in VBScript Classes (covered in Chapter 8).
- Start with minimal functionality, and avoid optimization until later.

Debugging

The term debugging has been wrongly attributed to the pioneer programmer, Grace Hopper. In 1944, Hopper, a young Naval Reserve officer, went to work on the Mark I computer at Harvard, becoming one of the first people to write programs for it. As Admiral Hopper, she later described an incident in which a technician is said to have pulled an actual bug (a moth, in fact) from between two electrical relays in the Mark II computer. In his book, *The New Hacker's Dictionary*, Eric Raymond reports that the moth was displayed for many years by the Navy and is now the property of the Smithsonian. Raymond also notes that Admiral Hopper was already aware of the term when she told the moth story.

The word bug was used prior to modern computers to mean an industrial or electrical defect.

For a long time now, debugging has been the sore point of scripting languages. Even though the script debugger has been available for quite some time, it has been difficult to install and use. Needless to say, it has not gained too much popularity. Still, successful installation of ASP script debugging on your development server will pay for itself tenfold. There are two debuggers available, one that can be downloaded with Internet Explorer, and another that can be installed with Visual InterDev, or MS Office 2000. The freely downloadable script debugger is actually integrated into InterDev, however, the InterDev interface offers more choices, and it allows for smooth debugging of ASP scripts. In this section we will discuss the concepts behind the InterDev debugger, as it is more robust (includes the easiest ASP debugging) and more intuitive to use. Depending on your needs, you may use the MS Script Editor (which is similar in its functionality to InterDev), and its debugger, or the Script Debugger (which has only some of the options of InterDev) downloadable from the Microsoft Scripting site `(http://msdn.microsoft.com/scripting/).`

To launch the free script debugger from Microsoft Internet Explorer, use the View menu, choose Script Debugger. Script Debugger starts, and then opens the current HTML source file.

If you want to start the script editor from within Office 2000 applications, use the Tools menu, choose Macro, and then Microsoft Script Editor.

If you are interested in switching debuggers, you can manipulate the registry to do so:

```
HKEY_CLASSES_ROOT\CLSID\{834128A2-51F4-11D0-8F20-
00805F2CD064}\LocalServer32
```

The default registry entry contains the path to the debugger, in case of the InterDev setup on my computer it is: `C:\WINNT\System32\mdm.exe` , to change it to the script debugger, I could enter `<path>\msscrdbg.exe` instead. In the registry, you can look for MDM Debug Session Provider Class.

Debugging with InterDev

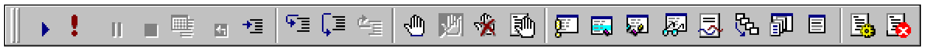
In order to set up the debugging you have to follow the directions included in the set-up instructions, including those for the InterDev server components that are available later on during the set up of Visual Studio. The best conditions for the set-up are a local development Windows NT Server with IIS that doubles as your InterDev workstation. The applications set-up is fairly fast, and the debugging process is a lot smoother (as well as easier to set up) than if the server and client were set up separately. In order to enable ASP debugging, you must also choose the Automatically Enable ASP server-side debugging on launch, which is available in the Launch pad of the Project's Properties window. When you quit your debugging session, Visual InterDev restores the server debugging settings and out-of-process setting to their previous values.

Additionally, InterDev offers just-in-time debugging, and can go automatically into debug mode whenever an error is encountered when executing a client script.

Do NOT install a debugger, or debug, on a production machine. The InterDev Debugger uses incredible resources on the system, and runs the application out of process on a single thread. Essentially, changes are made to IIS and MTS that make them run very slow.

Script debugging allows you to identify syntax, runtime and logic errors by inspection of both the script code and the contents of its variables during the execution of the script. Once your code is at a stage when it can be debugged, you'll be interested in setting up breakpoints (in order to pause the execution, or 'play', of the script), stepping through lines of script one by one, and inspecting the values of variables and objects.

There are several different ways in which these things can be accomplished. The two main ways are with the Debug Menu, which should switch on automatically once you start 'playing' the script, or can be switched on manually; and with the Code Window (or its shortcut menu). Let's take a look at the Debug Menu first:



The tables below contain a description of each group of buttons on the Visual InterDev Debug Menu shown above. Since the debug menu is also shared with Visual J++, some of the elements, related to threading etc. are not used when debugging with InterDev:

Group I	
Start	Begins debugging of the project by starting the script selected from the Project shortcut menu; this button can also be used to continue the running of the script.
Start Without Debugging	Project is executed, but the debugger is not started.

Group I	
Pause	Allows you to pause a running script at any time in order to start debugging it and/or to inspect the values of its variables.
Stop	Stops the debugging session altogether.
Detach All Processes	Used with J++
Restart	Restarts the application after any type of interruption.
Run To Cursor	After the execution has been paused, this allows you to set the point within your script where the execution will continue up to.

Group II	
Step Into	Allows you to execute the next line of code.
Step Over	Executes the next procedure as if it were a single line of code.
Step Out	Executes the remaining lines within a procedure.

Group III	
Insert Breakpoint	Inserts a breakpoint at the current line.
Enable/Disable Breakpoint	Toggles breakpoint status, allowing breakpoints to be turned 'on' or 'off'.
Clear All Breakpoints	Erases all of the breakpoints.
Breakpoints	Shows all of the breakpoints and their advanced properties within the Advanced Window.

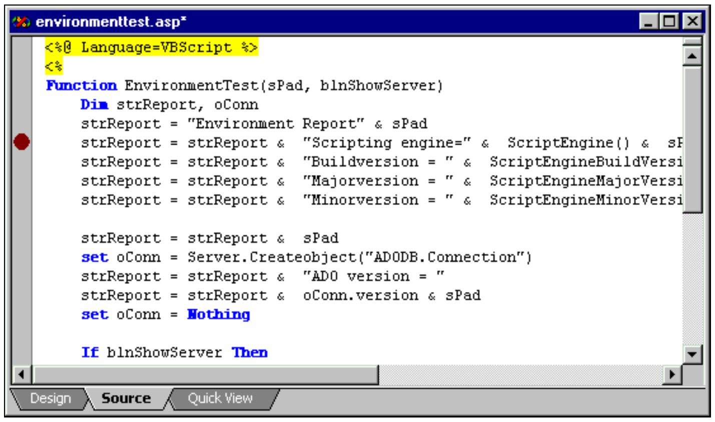
Group IV	
Immediate	Opens the Immediate Window.
Autos	Opens the Autos Window.
Locals	Opens the Locals Window.
Watch	Opens the Watch Window.
Threads	Opens the Threads Window (only used with J++).

Table Continued on Following Page

Group IV	
Call Stack	Opens the Call Stack Window.
Running Documents	Opens the Running Documents Window.
Output	Opens the Output Window.

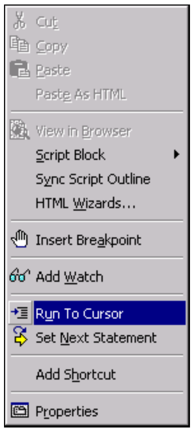
Group V	
Processes	Used with J++
Java Exceptions	Used with J++

The Code Window within Visual InterDev debugger (with some sample code) is shown below, this is the sample file created previously:



```
<%@ Language=VBScript %>
<%
Function EnvironmentTest(sPad, blnShowServer)
    Dim strReport, oConn
    strReport = "Environment Report" & sPad
    strReport = strReport &  "Scripting engine=" &  ScriptEngine() &  sP
    strReport = strReport &  "Buildversion = " &  ScriptEngineBuildVersi
    strReport = strReport &  "Majorversion = " &  ScriptEngineMajorVersi
    strReport = strReport &  "Minorversion = " &  ScriptEngineMinorVersi

    strReport = strReport &  sPad
    set oConn = Server.Createobject("ADODB.Connection")
    strReport = strReport &  "ADO version = "
    strReport = strReport &  oConn.version & sPad
    set oConn = Nothing

    If blnShowServer Then
```

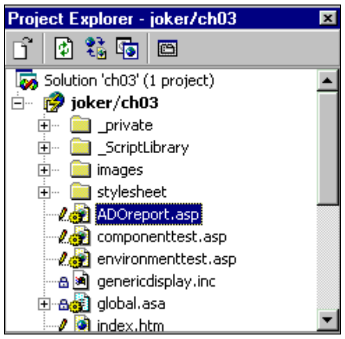
To get your program to pause automatically during a debug run, you have to set breakpoints. These can either be based on certain conditions (e.g. breakpoint reached 5 times, or a certain expression changes), or on a particular line of code. You may either click the mouse in the left-margin area of the window to toggle a breakpoint, or you can use either the Debug Menu or the Code Window shortcut menu:



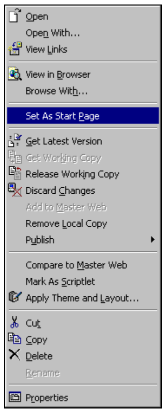
When you right-mouse click in the Code Window, the shortcut menu shown in the screenshot above pops up. There are four interesting items here:

Insert Breakpoint	By clicking Insert Breakpoint, it automatically adds a breakpoint at the line where your cursor is located, unless the line is empty, a declaration, or a comment.
Add Watch	By clicking Add Watch over a variable, or an object, it automatically adds it to the Watch Window. It is a very useful feature, as it allows you to concentrate on the few variables that you are actually interested in examining, rather than looking at the entire stack of variables in the Locals Window.
Run To Cursor	The Run To Cursor option allows you to execute a number of lines of code between the current location and the line that the cursor is pointing to. This is similar to placing a temporary breakpoint, and then continuing execution of the code until that breakpoint. This is useful when you are tangled in a long, complex loop and simply want to get out of the loop as fast as possible.
Set Next Statement	The Set Next Statement option allows you to execute an arbitrary line of code.

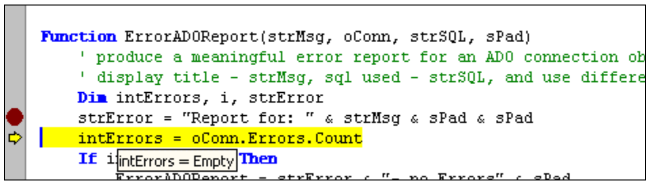
In order to start debugging an individual page, a start page needs to be set in the project explorer window. If your script depends on other pages (e.g. you are testing a page that requires values from a form), set it to the first page that is needed for the script to run properly. First go to the project window, and select a file:



Afterwards, you can right-mouse click on the file for the pop-up menu to appear:

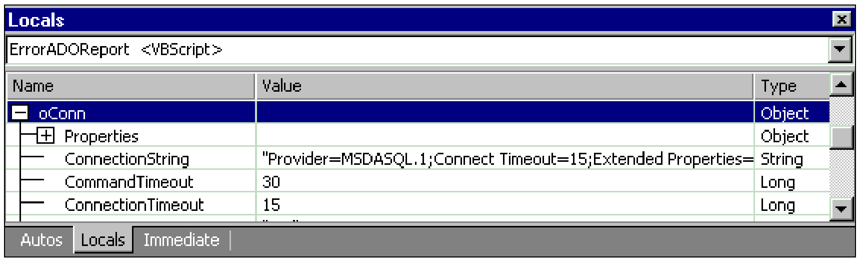


In order to start debugging, you should select the file as a Start Page. This is similar to VB's concept of a particular form (or code) being executed when the Start button is pressed.



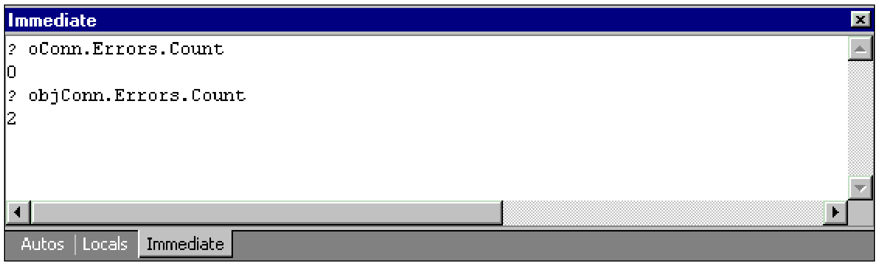
```
Function ErrorADOReport(strMsg, oConn, strSQL, sPad)
    ' produce a meaningful error report for an ADO connection ob
    ' display title - strMsg, sql used - strSQL, and use differe
    Dim intErrors, i, strError
    strError = "Report for: " & strMsg & sPad & sPad
    intErrors = oConn.Errors.Count
    If i intErrors = Empty Then
```

During the actual debugging process, the Code Window (above) comes alive. The majority of the features on the Debug Menu are available, and you can hover the mouse over variables to see their current values. Additionally, you may use some of the windows to perform specific actions. At this stage you may freely step though the code. Code stepping, like a dance, is a certain skill that needs to be acquired. First, you need to place your breakpoints in critical areas (or use the advanced breakpoints that can be set programmatically), and then test different 'stepping' possibilities - especially stepping over long routines, running to cursor, and finally, continuing the script to the next breakpoint (by pressing Start).



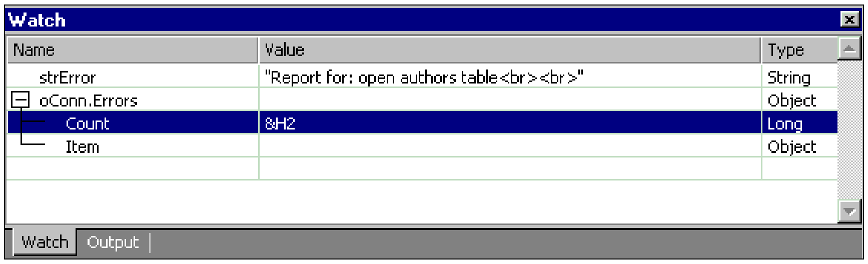
Name	Value	Type
oConn		Object
Properties		Object
ConnectionString	"Provider=MSDASQL.1;Connect Timeout=15;Extended Properties=	String
CommandTimeout	30	Long
ConnectionTimeout	15	Long

The Locals Window (above) is the most complex of all the windows and, in the long run, the least useful. It contains all the objects, variables, and object collections – along with their names, values and subtypes – that are currently within scope of reach (global and local variables), depending on your position in the script. Because some objects, such as the Connection object shown, may have many collections and properties, this window simply becomes too small for its own good. What normally happens is that you end up frantically chasing a few variables around with the use of the scrollbar. On the other hand, if you have only a few local variables, then it is very friendly and easy to use.



```
? oConn.Errors.Count
0
? objConn.Errors.Count
2
```

The Immediate Window (above) is the internal hacking tool for your script. With this window, you can inspect and change the values of variables within your script (if you'd rather not do this with either the Locals or Watch Windows), or run related or unrelated code. This window also provides a good opportunity to thoroughly test your scripts by feeding the procedures illegal values (by changing the value in the Value column), and then testing how the error handler will be able to cope with the problem. The Immediate Window can also give you a deeper insight into some of the interactions that occur between different variables that would otherwise be impossible, considering that many of the variables will only have values at runtime.

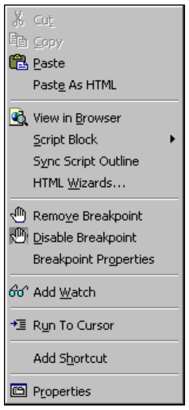

Watch

Name	Value	Type
strError	"Report for: open authors table "	String
⊟ oConn.Errors		Object
Count	&H2	Long
Item		Object

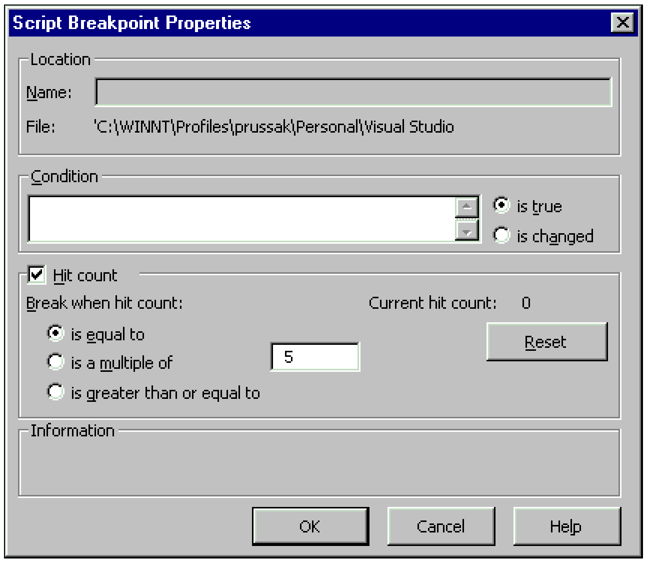
The Watch Window (above) is the user-friendlier version of the Locals Window, and has many of the same features. For example, you are able to inspect the types, names and values of specific variables, chosen by you, currently within scope. This has some benefits over the Locals Window, such as being able to observe when a particular variable comes into scope, as opposed to all of the variables that are displayed in locals window. Additionally, you may watch a particular property of a variable, which allows you to cut through the maze of + and - signs that would otherwise be displayed in the object model within the locals window. Watch Window can list a collection or a property of an object directly within the Watch Window, by specifying the member directly in the Name column. In the example above, the `oConn.Errors` collection is specified, as opposed to the entire `oConn` object. Entries within the Watch Window can be added directly from the Code Window's shortcut menu, and the values manipulated.

Advanced Breakpoints

The final interesting feature of the debugger is the ability to set smart breakpoints, by choosing Breakpoint Properties from the pop-up menu, available when your mouse is set over a breakpoint:



This is the same menu as seen previously, but based on the context, you have the capability of removing the breakpoint, disabling it, or setting some advanced properties, as seen in the screen below:


Script Breakpoint Properties
Location
Name:
File: 'C:\WINNT\Profiles\prussak\Personal\Visual Studio
Condition
[✓ is true] is changed
[✓ Hit count]
Break when hit count:
Current hit count: 0
[✓ is equal to] is a multiple of / is greater than or equal to
5
Reset
Information
OK
Cancel
Help

Although the location property is disabled in InterDev debugging, the other two properties make debugging smoother. You can:

- Define a conditional expression for the breakpoint. And pause execution when expression is true, or it changes.
- Specify the number of times a breakpoint should be hit before pausing code execution, using a variety of conditions. This property can also be changed when the script is paused, and the actual number of hits monitored.

Some other aspects of debugging not described here in detail are:

- Autos Window - displays variables within scope of the current line of execution.
- Output Window - displays status messages at runtime, not used.
- Call Stack - displays all procedures within the current thread of execution; useful when you want to jump between procedures, or stack frames.
- Threads Window - displays threads used by the application (for J++ debugging only).

Common Errors and How To Avoid Them

No matter what language you use or what you are doing, there are some errors that just keep on cropping up. Here are some of the more common ones, along with some good tips for avoiding them:

Problem	Suggestion for avoiding
Wrong data type in a variable, such as expecting a text value from a property instead of a number.	Explicitly declare variables, even if not required. In VBScript, use the `Dim` statement. Use naming conventions to help you remember variable types, such as `txtUserName` for a string, `fEnd` for a flag, and `intCounter` for an integer, etc.
Not understanding what objects are available in a given context, such as attempting to use the Internet Explorer object model in a script running on a different browser.	Be aware what objects your scripts have access to and what the scope or context is of an object. Be aware that objects (such as browser built-in objects) are not an inherent part of a language such as VBScript.
Not understanding a function or procedure or calling the incorrect function.	Double-check that the function you are calling performs the task you want it to.
Incorrect arguments for functions or arguments passed in the wrong order or not understanding what values a function or procedure returns.	Check syntax for functions whenever using them. Avoid relying on default argument values.
Not understanding a data structure, such as the object model for a browser, or trivial misunderstandings such as assuming that an array index begins with 1 instead of 0.	Check documentation for information about structure.
Typographic errors, such as misspelling a variable name or keyword, or forgetting to close a bracket.	Use consistent names to help avoid confusion. Type the closing portion of a statement as soon as you type the opening portion.
Unexpected data, such as a user typing in a string when prompted for a number.	Anticipate errors introduced by users and create error-handling routines.
Not understanding language conventions, such as using the wrong type of quotation marks to enclose literals. This is a really easy mistake to make when switching between languages.	Familiarize yourself with the operators and conventions of the language you are using.

Summary

In this chapter we looked at the process of handling errors and debugging VBScript code.

After configuring the host to display errors appropriately, we began by looking at the three types of error possible and how they are caused:

- ❑ Syntax errors.
- ❑ Runtime errors.
- ❑ Logic errors.

Having looked at the errors we then looked at how we can handle them. First, we looked at how we use the `On Error Resume Next` statement and then the `Err` object and its five properties and two methods. These methods and properties allow us to create a custom response to errors and also to `Raise` and `Clear` errors. We then briefly looked at other ways to handle errors, such as by creating custom help files to aid the user.

We then looked in more detail at the steps involved in dealing with errors:

- ❑ Diagnose what went wrong.
- ❑ Attempt to correct the error.
- ❑ Come up with a user friendly error message.
- ❑ Attempt to log the error.

We then covered some points on defensive programming before looking at the process of debugging VBScript code and how the InterDev debugger can help to make this vital process easier. Finally, we gave a list of some common errors to be aware of, and how to avoid them.

It is impossible to cover the whole topic of error handling and debugging in one chapter, or even in one book. Every script is different and so are the errors associated with it. This chapter's aim was to provide you with the basic strategies for finding and eliminating errors, and handling the remainder that the user might come across.