Microsoft

Microsoft VBSCRIPT

Step by Step

Ed Wilson

How to access your CD files

The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/VBSScriptSBS/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2006934395

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   1 0 9 8 7 6

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Martin DelRe
Project Editor: Maureen Williams Zimmerman
Copy Editor: Sarah Wales-McGrath
Technical Reviewer: David Holder
Indexer: Jeanne Busemeyer

Body Part No. X13-24158

Dedication

This book is dedicated to my best friend, Teresa.

Contents at a Glance

Table of Contents

Acknowledgments

The process of writing a technical book is more a matter of collaboration, support, and team work, than a single wordsmith sitting under a shade tree with parchment and pen. It is amazing how many people know about your subject after you begin the process.

I am very fortunate to have assembled a team of friends and well wishers over the past few years to assist, cajole, exhort, and inspire the words to appear. First and foremost is my wife Teresa. She has had the great fortune of reading 10 technical books in the past 11 years, while at the same time putting up with the inevitable encroachment of deadlines on our otherwise well timed vacation schedule. Claudette Moore of the Moore Literary Agency has done an awesome job of keeping me busy through all her work with the publishers. Martin DelRe at Microsoft Press has been a great supporter of scripting technology, and is a great person to talk to. Maureen Zimmerman, also of Microsoft Press, has done a great job of keeping me on schedule, and has made numerous suggestions to the improvement of the manuscript.

Introduction

Network administrators and consultants are confronted with numerous mundane and time-consuming activities on a daily basis. Whether it is going through thousands of users in Active Directory Users and Computers to grant dial-in permissions to a select group, or changing profile storage locations to point to a newly added network server, these everyday tasks must be completed. In the enterprise space, the ability to quickly write and deploy a Microsoft Visual Basic Script (VBScript) will make the difference between a task that takes a few hours and one that takes a few weeks.

As an Enterprise Consultant for Microsoft Corporation, I am in constant contact with some of the world's largest companies that run Microsoft software. The one recurring theme I hear is, "How can we effectively manage thousands of servers and tens of thousands of users?" In some instances, the solution lies in the employment of specialized software packages—but in the vast majority of the cases, the solution is a simple VBScript.

In Microsoft Windows Server 2003, enterprise manageability was one of the design goals, and VBScript is one path to unlocking the rich storehouse of newly added features. Using the techniques outlined in *Microsoft VBScript Step by Step*, anyone can begin crafting custom scripts within minutes of opening these pages. I'm not talking about the traditional Hello World script—I'm talking about truly useful scripts that save time and help to ensure accurate and predictable results.

Whereas in the past scripting was somewhat hard to do, required special installations of various implementations, and was rather limited in its effect, with the release of Microsoft Windows XP, Windows Server 2003, and Windows Vista, scripting is coming into its own. This is really as it should be. However, most administrators and IT professionals do not have an understanding of scripting because in the past scripting was not a powerful alternative for platform management.

However, in a large enterprise, it is a vital reality that one simply cannot perform management from the GUI applications because it is too time-constraining, too error prone, and, after a while, too irritating. Clearly there needs to be a better way, and there is. Scripting is the answer.

A Practical Approach to Scripting

Microsoft VBScript Step by Step will equip you with the tools to automate setup, deployment, and management of Microsoft Windows 2003 networks via the various scripting interfaces contained within the product. In addition, it will provide you with an understanding of a select number of VBScripts adaptable to your own unique environments. This will lead you into an awareness of the basics of programming through modeling of fundamental techniques.

The approach I take to teaching you how to use VBScript to automate your Windows 2003 servers is similar to the approach used in some executive foreign language schools. You'll learn by using the language. In addition, concepts are presented not in a dry academic fashion, but in a dynamic, real-life manner. When a concept is needed to accomplish something, it is presented. If a topic is not useful for automating network management, I don't bring it forward.

This is a practical, application-oriented book, so the coverage of VBScript, Windows Scripting Host, Active Directory Service Interfaces (ADSI), and Windows Management Instrumentation (WMI) is not exceedingly deep. This is not a reference book; it is a tutorial, a guide—a springboard for ideas, perhaps—but not an encyclopedia.

Is This Book for Me?

Microsoft VBScript Step by Step is aimed at several audiences, including:

- **Windows networking consultants** Anyone desiring to standardize and automate the installation and configuration of .NET networking components.
- **Windows network administrators** Anyone desiring to automate the day-to-day management of Windows .NET networks.
- **Windows Help Desk staff** Anyone desiring to verify configuration of remotely connected desktops.
- **Microsoft Certified Systems Engineers (MCSEs) and Microsoft Certified Trainers (MCTs)** Although scripting is not a strategic core competency within the MCP program, a few questions about scripting do crop up from time to time on various exams.
- **General technical staff** Anyone desiring to collect information, configure settings on Windows XP machines, or implement management via WMI, WSH, or WBEM.
- **Power users** Anyone wishing to obtain maximum power and configurability of their Windows XP machines either at home or in an unmanaged desktop workplace environment.

Outline of This Book

This book is divided into four parts, each covering a major facet of scripting. The following sections describe these parts.

Part I: Covering the Basics

Okay, so you've decided you need to learn scripting. Where do you begin? Start here in Part I! In Chapter 1, "Starting From Scratch," you learn the basics: what a script is, how to read it, and how to write it. Once you move beyond using a script to figure out what your IP address is and print it to a file, you need to introduce some logic into the script, which you do in Chapter 2 through Chapter 5. You'll learn how to introduce conditions and add some intelligence to allow the script to check some stuff, and then based upon what it finds, do some other stuff. This section concludes by looking at troubleshooting scripts. I’ve made some mistakes that you don’t need to repeat! Here are the chapters in Part I:

- Chapter 1, “Starting from Scratch”
- Chapter 2, “Looping Through The Script”
- Chapter 3, “Adding Intelligence”
- Chapter 4, “Working with Arrays”
- Chapter 5, “More Arrays”

Part II: Basic Windows Administration

In Part II, you dig deep into VBScript and WMI and really begin to see the power you can bring to your automation tasks. In working with the file system, you see how to use the file system object to create files, delete files, and verify the existence of files. All these basic tasks provide loads of flexibility for your scripts. Next, you move on to working with folders, learning how to use VBScript to completely automate the creation of folders and files on your servers and users’ workstations. In the last half of Part II, you get an in-depth look at the power of WMI when it is combined with the simplicity and flexibility of VBScript. Here are the chapters in Part II:

- Chapter 6, “Working with the File System”
- Chapter 7, “Working with Folders”
- Chapter 8, “Using WMI”
- Chapter 9, “WMI Continued”
- Chapter 10, “Querying WMI”

Part III: Advanced Windows Administration

This section will shave at least four points off your golf handicap because you’ll get to play an extra 18 holes a week due to the time you’ll save! At least three things are really painful when it comes to administering Windows servers: all those click, click, and save motions; all the time spent waiting for the screen to refresh; and loosing your place in a long list of users. Guess what? In this section, some of that pain is relieved. When Human Resources hires 100 people, you tell them to send you a spreadsheet with the new users, and then use a script to create those users. It takes 2 minutes instead of 2 hours. (Dude, that’s the front nine!) In addition to saving time, scripting your administrative tasks reduces the likelihood of errors. If you have to set a particular set of access control lists on dozens of folders, a script is the only way to ensure all the flags are set correctly. Here are the chapters in Part III:

- Chapter 11, “Introduction to Active Directory Service Interfaces”
- Chapter 12, “Writing for ADSI”

- Chapter 13, “Using ADO to Perform Searches”
- Chapter 14, “Configuring Networking Components”
- Chapter 15, “Using Subroutines and Functions”
- Chapter 16, “Logon Scripts”
- Chapter 17, “Working with the Registry”
- Chapter 18, “Working with Printers”

Part IV: Scripting Other Applications

Once you learn how to use WMI and VBScript to automate Windows Server 2003, the logical question is, “What else can I do?” Well, with the latest version of Microsoft Exchange and Internet Information Services (IIS), the answer is, “Quite a lot.” So in this part of the book, you look at using WMI and VBScript to automate other applications.

In IIS 6.0, nearly everything that can be configured via GUI tools can also be scripted. This enables the Web administrator to simplify management and to also ensure repeatable configuration of the Web sites from a security perspective.

In Exchange administration, many routine tasks can be simplified by using VBScript. In Part IV, you look at how to leverage the power of VBScript to simplify user management, to configure and administer Exchange, and to troubleshoot some of the common issues confronting the enterprise Exchange administrator. The chapters in Part IV are as follows:

- Chapter 19, “Managing IIS 6.0”
- Chapter 20, “Working with Exchange 2003”
- Chapter 21, “Troubleshooting WMI Scripting"

Part V: Appendices

The appendices in this book are not the normal “never read” stuff. Indeed, you will find yourself referring again and again to these five crucial documents. In Appendix A you will find lots of ideas for further work in developing your mastery of VBScript. Appendix B will save you many hours of searching for the “special names” that unlock the power of ADSI scripting. Appendix C helps you find the special WMI namespaces that enable you to perform many cool “tricks” in your scripting. And last but certainly not least is Appendix D, which contains my documentation “cheat sheet.” Actually, you will want to read it rather early in your scripting career. Appendix E contains the Special Folder Constants, which, as you will see in the very first script in the book, can provide easy access to some of the most vital folders on your workstation!

- Appendix A, “VBScript Documentation”
- Appendix B, “ADSI Documentation”

- Appendix C, "WMI Documentation"
- Appendix D, "Documentation Standards"
- Appendix E, "Special Folder Constants"

Finding Your Best Starting Point

This book will help you add essential skills for using VBScript to automate your Windows environment. You can use this book if you are new to scripting, new to programming, or switching from another scripting language. The following table will assist you in picking the best starting point in the book.

If you are	Follow these steps
New to programming	Install the practice files as described in the section "Installing the Practice Files on Your Computer" later in this Introduction.
	Learn the basic skills for using VBScript by working through Chapters 1-7 in order.
New to VBScript	Install the practice files as described in the section "Installing the Practice Files on Your Computer" later in this Introduction.
	Skim through Chapter 1, making sure you pay attention to the section on creating objects.
	Skim Chapter 2 and Chapter 3.
	Complete Chapter 4 through Chapter 7 in order.
Experienced with VBScript but are interested in using WMI	Install the practice files as described in the section "Installing the Practice Files on Your Computer" later in this Introduction.
	Skim Chapter 4, paying attention to handling arrays.
	Work through Chapters 8-10 in order. Complete Chapter 14.

About the Companion CD

The CD accompanying this book contains additional information and software components, including the following files:

- **Sample Files** The chapter folders contain starter scripts, some text files, and completed solutions for each of the procedures contained in this book. In addition, each of the scripts discussed in the book is contained in the folder corresponding to the chapter number. For instance, in Chapter 1 we talk about enumerating disk drives on a computer system. The script that makes up the bulk of our discussion around that topic is contained in the \My Documents\Microsoft Press\VBScriptSBS\ch01 folder. You'll also find many bonus scripts in the chapter folders. In addition to the sample files in the chapter folders, the CD includes a Templates folder, a Resources folder, a Supplemental folder, and a Utilities folder. These folders contain dozens of my favorite scripts and utilities I have written over the last several years to solve literally hundreds of problems. You will enjoy playing around with these and incorporating them into daily scripting tasks. For example, in the Templates folder you will find my WMITemplate.vbs script. By using it as a starter, you can write a custom WMI script in less than five seconds. By using the ADOSearchTemplate.vbs script as a starter, you can write a script that returns all the users in a particular OU in less than three seconds. In the Utilities folder you will find, for example, a script that converts bytes into kilobytes, megabytes, or gigabytes depending on the largest whole number that can be so created.

- **eBook** You can view an electronic version of this book on screen using Adobe Acrobat Reader. For more information, see the Readme.txt file included in the root folder of the Companion CD.
- **Tools and Resources** Additional tools and resources to make scripting faster and easier: Scriptomatic 2.0, Tweakomatic, EZADScriptomatic, WMI Admin Tools, WMI CodeCreator, WMI Diag.

Installing the Practice Files on Your Computer

Follow these steps to install the practice files on your computer so that you can use them with the exercises in this book.

1. Remove the companion CD from the package inside this book and insert it into your CD-ROM drive.

Note

2. Review the end user license agreement. If you accept the terms, select the accept option and then click Next.

   A menu will appear with options related to the book.
3. Click Install Code Samples.
4. Follow the instructions that appear.

The code samples are installed to the following location on your computer:

My Documents\Microsoft Press\VBScriptSBS\

Uninstalling the Practice Files

Follow these steps to remove the practice files from your computer.

1. In the Control Panel, open Add Or Remove Programs.
2. From the list of Currently Installed Programs, select Microsoft VBScript Step by Step.
3. Click Remove.
4. Follow the instructions that appear to remove the code samples.

System Requirements

- Special Folder Constants
- Minimum 233 MHz in the Intel Pentium/Celeron family or the AMD k6/Atholon/Duron family
- 64 MB memory
- 1.5 GB available hard disk space
- Display monitor capable of 800 x 600 resolution or higher
- CD-ROM drive or DVD drive
- Microsoft Mouse or compatible pointing device
- Windows Server 2003,Windows XP, or Windows Vista
- The MSI Provider installed on Windows Server 2003 (required for some of the WMI procedures)
- Microsoft Office Excel or Excel Viewer

Technical Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD-ROM. Microsoft Press provides corrections for books through the World Wide Web at *http:// www.microsoft.com/learning/support.*

To connect directly with the Microsoft Press Knowledge Base and enter a query regarding a question or an issue that you might have, go to *http://www.microsoft.com/learning/support/search.asp*.

If you have comments, questions, or ideas regarding this book or the companion CD-ROM, please send them to Microsoft Press using either of the following methods:

E-mail:

msinput@microsoft.com

Postal Mail:

Microsoft Press

Attn: Editor, *Microsoft VBScript Step by Step*

One Microsoft Way

Redmond, WA 980526399

Please note that product support is not offered through the preceding addresses.

For additional support information regarding this book and the CD-ROM (including answers to commonly asked questions about installation and use), visit the Microsoft Press Technical Support Web site at *www.microsoft.com/learning/support/books/*. For support information regarding Microsoft software, please connect to *http://support.microsoft.com*.

Chapter 3

Adding Intelligence

Before You Begin

To successfully complete this chapter, you need to be familiar with the following concepts, which were presented in Chapters 1 and 2:

- Declaring variables
- Basic error handling
- Connecting to the file system object
- Using *For Each...Next*
- Using *Do While*

After completing this chapter, you will be able to:

- Use *If...Then*
- Use *If...Then...ElseIf*
- Use *If...Then...Else*
- Use *Select Case*
- Use intrinsic constants

If...Then

If...Then is one of those programming staples (like fried chicken and mashed potatoes are staples in the southern United States). What's nice about *If...Then* is that it makes sense. We use this kind of logic all the time.

The basic operation is diagrammed here:

If	condition	Then	action
If	store is open	Then	buy chicken

The real power of *If...Then* comes into play when combined with tools such as those we looked at in Chapter 2, "Looping Through the Script." *If...Then* is rarely used by itself. Although you could have a script such as IfThen.vbs, you wouldn't find it very valuable.

IfThen.vbs

```
On Error Resume Next
Const a = 2
Const b = 3
Const c = 5
If a + b = c Then
WScript.Echo(c)
End If
```

In this script three constants are defined: *a*, *b*, and *c*. We then sum the numbers and evaluate the result by using the *If...Then* statement. There are three important elements to pay attention to in implementing the *If...Then* construct:

- The *If* and the *Then must* be on the *same* line
- The action to be taken must be on the *next* line
- You must end your *If...Then* statement by using *End If*

If any of these elements are missing or misplaced, your *If...Then* statement generates an error. Make sure you remember that *End If* is two words, and not one word as in some other programming languages. If you do not see an error and one of these elements is missing, make sure you have commented out *On Error Resume Next*.

Now that you have the basic syntax down pat, let's look at the following more respectable and useful script, named GetComments.vbs, which is in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03. *If* you put lots of descriptive comments in your Microsoft Visual Basic, Scripting Edition (VBScript) scripts, *Then* GetComments.vbs pulls them out and writes them into a separate file. This file can be used to create a book of documentation about the most essential scripts you use in your network. In addition, *If* you standardize your documentation procedures, *Then* the created book will require very little touch-up work when you are finished. (OK, I'll quit playing *If...Then* with you. Let's look at that code, which is described in the next few sections.)

GetComments.vbs

```
Option Explicit
On Error Resume Next
Dim scriptFile
Dim commentFile
Dim objScriptFile
Dim objFSO
Dim objCommentFile
Dim strCurrentLine
Dim intIsComment
Const ForReading = 1
Const ForWriting = 2
scriptFile = "displayComputerNames.vbs"
commentFile = "comments.txt"
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objScriptFile = objFSO.OpenTextFile _
  (scriptFile, ForReading)
```

```
Set objCommentFile = objFSO.OpenTextFile(commentFile, _
  ForWriting, TRUE)
Do While objScriptFile.AtEndOfStream <> TRUE
  strCurrentLine = objScriptFile.ReadLine
  intIsComment = Instr(1,strCurrentLine,"'")
  If intIsComment > 0 Then
    objCommentFile.Write strCurrentLine & VbCrLf
  End If
Loop
WScript.Echo("script complete")
objScriptFile.Close
objCommentFile.Close
```

Just the Steps To implement *If…Then*

1. On a new line in the script, type **If *some condition* Then**.
2. On the next line, enter the command you want to invoke.
3. On the next line, type **End If**.

Header Information

The first few lines of the GetComments.vbs script contain the header information. We use *Option Explicit* to force us to declare all the variables used in the script. This helps to ensure that you spell the variables correctly as well as understand the logic. *On Error Resume Next* is rudimentary error handling. It tells VBScript to go to the next line in the script when there is an error. There are times, however, when you do not want this behavior, such as when you copy a file to another location prior to performing a delete operation. It would be disastrous if the copy operation failed but the delete worked.

After you define the two constants, you define the variables. Listing variables on individual lines makes commenting the lines in the script easier, and the commenting lets readers of the script know why the variables are being used. In reality, it doesn't matter where you define variables, because the compiler reads the entire script prior to executing it. This means you can spread constant and variable declarations all over the script any way you want—such an approach would be hard for humans to read, but it would make no difference to VBScript.

Reference Information

In the Reference information section of the script, you define constants and assign values to several of the variables previously declared.

The lines beginning with Const of the GetComments.vbs script define two constants, *ForReading* and *ForWriting*, which make the script easier to read. (You learned about constants in Chapter 2.) You'll use them when you open the DisplayComputerNames.vbs file and the comments.txt file from the ch03 folder on the CD. You could have just used the numbers 1 and 2 in your command and skipped the two constants; however, someone reading the script needs to know what the numbers are doing. Because these values will never change, it is more efficient to define a constant instead of using a variable. This is because the computer knows that you will only store two small numbers in these two constants. On the other hand, if we declared these two as variables, then the operating system would need to reserve enough memory to hold anything from a small number to an entire object. These varients (as they are called) are easy on programmers, but are wasteful of memory resources. But it is, after all, just a scripting language. If we were really concerned about efficiency, and conservation of resources, we would be writing in C++.

The name of the file you are extracting comments from is stored in the variable *scriptFile*. By using the variable in this way it becomes easy to modify the script later so that you can either point it to another file or make the script read all the scripts in an entire folder. In addition, you could make the script use a command-line option that specifies the name of the script to parse for comments. However, by assigning a variable to the script file name, you make all those options possible without a whole lot of rewriting. This is also where you name the file used to write the comments into—the aptly named comments.txt file.

Quick Check

Q. Is it permissible to have *If* on one line and *Then* on a separate line?

A. No. Both *If* and *Then* must be on the same logical line. They can be on separate physical lines if the line continuation character (_) is used. Typically, *If* is the first word and *Then* is the last command on the line.

Q. If the *Then* clause is on a separate logical line from the *If...Then* statement, what command do you use?

A. *End If*. The key here is that *End If* consists of two words, not one.

Q. What is the main reason for using constants?

A. Constants have their value set prior to script execution, and therefore their value does not change during the running of the script.

Q. What are two pieces of information required by the *OpenTextFile* command?

A. *OpenTextFile* requires both the name of the file and whether you want to read or write.

Worker and Output Information

The Worker and Output information section is the core engine of the script, where the actual work is being done. You use the *Set* command three times, as shown here:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objScriptFile = objFSO.OpenTextFile _
  (scriptFile, ForReading)
Set objCommentFile = objFSO.OpenTextFile(commentFile, _
  ForWriting, TRUE)
```

Regarding the first *Set* command, you've seen *objFSO* used several times already in Chapter 2. *objFSO* is a variable name, which we routinely assign to our connection to the file system, that allows us to read and write to files. You have to create the file system object object (as it is technically called) to be able to open text files.

The second *Set* command uses our *objScriptFile* variable name to allow us to read the DisplayComputerNames.vbs file. Note that the *OpenTextFile* command requires only one piece of information: the name of the file. VBScript will assume you are opening the file for reading if you don't include the optional file mode information. We are going to specify two bits of information so that the script is easier to understand:

- The name of the file
- How you want to use the file—that is, read or write to it

By using variables for these two parts of the *OpenTextFile* command, you make the script much more flexible and readable.

The third *Set* command follows the same pattern. You assign the *objCommentFile* variable to whatever comes back from the *openTextFile* command. In this instance, however, you write to the file instead of read from it. You also use variables for the name of the comment file and for the option used to specify writing to the file.

The GetComments.vbs script reads each line of the DisplayComputerNames.vbs file and checks for the presence of a single quotation mark ('). When the single quotation mark is present, the script writes the line that contains that character out to the comments.txt file.

A closer examination of the Worker and Output information section of the GetComments.vbs script reveals that it begins with *Do While...Loop*, as shown here:

```
Do While objScriptFile.AtEndOfStream <> TRUE
  strCurrentLine = objScriptFile.ReadLine
  intIsComment = InStr(1,strCurrentLine,"'")
  If intIsComment > 0 Then
    objCommentFile.Write strCurrentLine & vbCrLf
  End If
Loop
WScript.Echo("script complete")
objScriptFile.Close
objCommentFile.Close
```

You first heard about the *Do While* statement in Chapter 2. *ObjScriptFile* contains a *textStream Object*. This object was created when we used the *openTextFile* method from the *fileSystem Object*. The *textStreamObject* has a property that is called *atEndOfStream*. As long as you aren't at the end of the text stream, the script reads the line of text and sees whether it contains a single quotation mark. *AtEndOfStream* is a property. It describes a physical location. Of course, we do not know where *AtEndOfStream* is located, so we use *Do While* to loop around until it finds the end of the stream.

To check for the presence of the <'> character, you use the *InStr* function, just as discussed in Chapter 2. The *InStr* function returns a zero when it does not find the character; when it does find the character, it returns a number representing the location in the line of text that the character was found.

If *InStr* finds the <'> character within the line of text, the variable *intIsComment* holds a number that is larger than zero. Therefore, you use the *If...Then* construct, as shown in the following code, to write out the line to the comments.txt file:

```
If intIsComment > 0 Then
  objCommentFile.Write strCurrentLine & vbCrLf
End If
```

Notice that the condition to be evaluated is contained within *If...Then*. If the variable *intIsComment* is larger than zero, you take the action on the next line. Here you use the *Write* command to write out the current line of the DisplayComputerNames.vbs file.

Use the *Timer* function to see how long the script runs

1. Open the GetComments.vbs script in Microsoft Notepad or the script editor of your choice.
2. Save the script as YourNameGetCommentsTimed.vbs.
3. Declare two new variables: *startTime* and *endTime*. Put these variables at the bottom of your list of variables, and before the constants. It will look like:

   ```
   Dim startTime, endTime
   ```

4. Right before the line where you create the FilesystemObject and set it to the *objFSO* variable, assign the value of the *Timer* function to the *startTime* variable. It will look like the following:

   ```
   startTime = Timer
   ```

5. Right before the script complete line, assign the value of timer to the *endTime* variable. It will look like:

   ```
   endTime = Timer
   ```

6. Now edit the *script complete* line to include the time it took to run. Use the *ROUND* function to round off the time to two decimal places. The line will look like the following:

   ```
   WScript.Echo "script complete. " & round(endTime-startTime, 2)
   ```

7. Save and run your script. Compare your script with GetCommentsTimed.vbs in \My Documents\Microsoft Press\VBScriptSBS\ch03 if desired.

Intrinsic Constants

You use the *vbCrLf* command to perform what is called a *carriage return* and *line feed*. *vbCrLf* is an *intrinsic constant*, which means that it is a constant that is built into VBScript. Because intrinsic constants are built into VBScript, you don't need to define them as you do regular constants. You'll use other intrinsic constants as you continue to develop scripts in VBScript language in later chapters.

vbCrLf has its roots in the old-fashioned manual typewriter. Those things had a handle on the end that rolled the plate up one or two lines (the line feed) and then repositioned the type head (the carriage return). Like the typewriter handle, the *vbCrLf* command positions the text to the first position on the following line. It's a very useful command for formatting text in both dialog boxes and text files. The last line in our *If...Then* construct is the *End If* command. *End If* tells VBScript that we're finished using the *If...Then* command. If you don't include *End If*, VBScript complains with an error.

The Platform SDK documents many other intrinsic constants that we can use with VBScript. Besides *vbCrLf,* the one I use the most is *vbTab*, which will tab over the default tab stop. It is helpful for indenting output. You can look up others in the Scripts56.chm file included in the Resources folder on the CD-ROM by searching for "intrinsic constants."

After using *End If*, you have the *Loop* command on a line by itself. The *Loop* command belongs to the *Do While* construct that began the Worker and Output information section. *Loop* sends the script execution back to the *Do While* line. VBScript continues to loop through, reading the text file and looking for ' marks, as long as it doesn't reach the end of the text stream. When VBScript reaches the end of the text stream from the DisplayComputerNames script, a message displays saying that you're finished processing the script. This is important, because otherwise there would be no indication that the script has concluded running. You then close your two files and the script is done. In reality, you don't need to close the files because they will automatically close once the script exits memory, but closing the files is good practice and could help to avoid problems if the script hangs.

Making a decision using *If...Then*

1. Open Notepad or the script editor of your choice. Navigate to the blankTemplate.vbs template in the \My Documents\Microsoft Press\VBScriptSBS\Templates directory.
2. Save the template as YourNameConvertToGig.vbs.
3. On the first non-commented line, type **Option Explicit**.
4. On the next line, declare the variable *intMemory*.
5. The next line begins the Reference section. Assign the value 120,000 to the *intMemory* variable. It will look like the following:

```
intMemory = 120000
```

6. Use the *formatNumber* function to remove all but the last two digits after the decimal place. It will look like the following:

```
intMemory = formatNumber(intMemory/1024)
```

7. If the value of *intMemory* is greater than 1024, then we want to convert it to gigabytes, print out the value, and then exit the script. We will use *formatNumber* to clean up the trailing decimal places. Your code will look like the following:

```
If intMemory > 1024 Then
intMemory = formatNumber(intMemory/1024) & " Gigabytes"
WScript.Echo intMemory
WScript.quit
End If
```

8. Under the *If...Then End If* construction, we assign the value to *intMemory,* as seen below.

```
intMemory = intMemory & " Megabytes"
```

9. Then we echo out the value of *intMemory*, as seen below:

```
WScript.Echo intMemory
```

10. Save and run the script. Compare your results to ConvertToGig.vbs in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03.

If...Then...ElseIf

If...Then...ElseIf adds some flexibility to your ability to make decisions by using VBScript. *If...Then* enables you to evaluate *one* condition and take action based on that condition. By adding *ElseIf* to the mixture, you can make multiple decisions. You do this in the same way you did it using the *If...Then* command. You start out with an *If...Then* on the first line in the Worker information section, and when you are finished, you end the *If...Then* section with *End If*. If you need to make additional evaluations, add a line with *ElseIf* and the condition.

Just the Steps To Use *If...Then...ElseIf*

1. On a new line in the script, type **If *some condition* Then**.
2. On the next line, enter the command you want to invoke.
3. On the next line, type **ElseIf** and the new condition to check, and end the line with **Then**.
4. On the next line, enter the command you want to invoke when the condition on the *ElseIf* line is true.
5. Repeat steps 3 and 4 as required.
6. On the next line, type **End If**.

You can have as many *ElseIf* lines as you need; however, if you use more than one or two, the script can get long and confusing. A better solution to avoid a long script is to convert to a *Select Case* type of structure, which is covered later in this chapter in the "Select Case" section.

Using the message box msgBox.vbs

1. Open Notepad or the script editor of your choice.
2. Define four variables that will hold the following: the title of the message box, the prompt for the message box, the button configuration, and the return code from the message box. The variables I used are *strPrompt, strTitle, intBTN, intRTN*. They are declared as follows:

   ```
   Dim strPrompt
   Dim strTitle
   Dim intBTN
   Dim intRTN
   ```

3. Assign values to the first three variables. *strPrompt* is what you want to display to the user. The title will appear at the top of the message box. The value contained in *strTitle* will appear at the top of the message box. The variable *intBTN* is used to control the style of the buttons you want displayed.

   ```
   strPrompt = "Do you want to run the script?"
   strTitle = "MsgBOX DEMO"
   intBTN = 3 '4 is yes/no 3 is yes/no/cancel
   ```

4. Now write the code to display the message box. To do this, we will use *intRTN* to capture the return code from pressing the button. We will use each of our three message box variables as well and the *msgBox* function. The line of code looks like the following:

   ```
   intRTN = MsgBox(strprompt,intBTN,strTitle)
   ```

5. If you run the script right now, it will run and display a message box, but you are not evaluating the outcome. To do that, we will use *If...Then...Else* to evaluate the return code. It will look like the following:

   ```
   If intRTN = vbYes Then
   WScript.Echo "yes was pressed"
   ElseIf intRTN = vbNo Then
   WScript.Echo "no was pressed"
   ElseIf intRTN = vbCancel Then
   WScript.Echo "cancel was pressed"
   Else
   WScript.Echo intRTN & " was pressed"
   End If
   ```

6. Save the script as YourNameMsgBox.vbs and run it. It should tell you which button was pressed from the message box. You would then tie in the code to the appropriate button instead of just echoing the return values. Compare your code with msgBox.vbs in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03 if required.

Let's examine a script that uses *If...Then...ElseIf* to detect the type of central processing unit (CPU) that is installed in a computer. Here is the CPUType.vbs script from the ch03 folder on the CD.

CPUType.vbs

```
Option Explicit
On Error Resume Next
Dim strComputer
Dim cpu
Dim wmiRoot
Dim objWMIService
Dim ObjProcessor
strComputer = "."
cpu = "win32_Processor='CPU0'"
wmiRoot = "winmgmts:\\" & strComputer & "\root\cimv2"
Set objWMIService = GetObject(wmiRoot)
Set objProcessor = objWMIService.Get(cpu)
If objProcessor.Architecture = 0 Then
  WScript.Echo "This is an x86 cpu."
ElseIf objProcessor.Architecture = 1 Then
  WScript.Echo "This is a MIPS cpu."
ElseIf objProcessor.Architecture = 2 Then
  WScript.Echo "This is an Alpha cpu."
ElseIf objProcessor.Architecture = 3 Then
  WScript.Echo "This is a PowerPC cpu."
ElseIf objProcessor.Architecture = 6 Then
  WScript.Echo "This is an ia64 cpu."
Else
  WScript.Echo "Cannot determine cpu type."
End If
```

Header Information

The Header information section contains the usual information (discussed in Chapters 1 and 2), as shown here:

```
Option Explicit
On Error Resume Next
Dim strComputer
Dim cpu
Dim wmiRoot
Dim objWMIService
Dim objProcessor
```

Option Explicit tells VBScript that you'll name all the variables used in the script by using the *Dim* command. *On Error Resume Next* turns on basic error handling. The *strComputer* variable holds the name of the computer from which we will perform the Windows Management Instrumentation (WMI) query. The *cpu* variable tells VBScript where in WMI we will go to read the information.

The *wmiRoot* variable enables you to perform a task you haven't performed before in previous scripts: split out the connection portion of WMI to make it easier to change and more readable. The variables *objWMIService* and *objProcessor* hold information that comes back from the Reference information section.

Reference Information

The Reference information section is the place where you assign values to the variables you named earlier in the script. The CPUType.vbs script contains these assignments:

```
strComputer = "."
cpu = "win32_Processor.deviceID='CPU0'"
wmiRoot = "winmgmts:\\" & strComputer & "\root\cimv2"
```

strComputer is equal to ".", which is a shorthand notation for the local computer that the script is currently executing on. With the *cpu* variable, you define the place in WMI that contains information about processors, which is *win32_Processor*. Because there can be more than one processor on a machine, you further limit your query to *CPU0*. It is necessary to use *CPU0* instead of *CPU1* because *win32_Processor* begins counting CPUs with 0, and although a computer always has a CPU0, it does not always have a CPU1. *DeviceID* is the key value for the WIN32_Processor WMI class. To connect to an individual instance of a processor, it is necessary to use the key value. The key of a WMI class can be discovered using wmisdk_book.chm from \My Documents\Microsoft Press\VBScriptSBS\resources, or by using the wbemTest.exe utility from a CMD prompt. In this script, you're only trying to determine the type of CPU running on the machine, so it isn't necessary to identify all CPUs on the machine.

Worker and Output Information

The first part of the script declared the variables to be used in the script, and the second part of the script assigned values to some of the variables. In the next section, you use those variables in an *If...Then...ElseIf* construction to make a decision about the type of CPU installed on the computer.

The Worker and Output information section of the CPUType.vbs script is listed here:

```
Set objWMIService = GetObject(wmiRoot)
Set objProcessor = objWMIService.Get(cpu)

If objProcessor.Architecture = 0 Then
  WScript.Echo "This is an x86 cpu."
ElseIf objProcessor.Architecture = 1 Then
  WScript.Echo "This is a MIPS cpu."
ElseIf objProcessor.Architecture = 2 Then
  WScript.Echo "This is an Alpha cpu."
ElseIf objProcessor.Architecture = 3 Then
  WScript.Echo "This is a PowerPC cpu."
ElseIf objProcessor.Architecture = 6 Then
  WScript.Echo "This is an ia64 cpu."
Else
  WScript.Echo "Cannot determine cpu type."
End If
```

To write a script like this, you need to know how *win32_Processor* hands back information so that you can determine what a 0, 1, 2, 3, or 6 means. By detailing that information in an *If...Then...ElseIf* construct, you can translate the data into useful information.

The first two lines listed in the preceding script work just like a normal *If...Then* statement. The line begins with *If* and ends with *Then*. In the middle of the *If...Then* language is the statement you want to evaluate. If *objProcessor* returns a zero when asked about the architecture, you know the CPU is an x86, and you use *WScript.Echo* to print out that data.

If, on the other hand, *objProcessor* returns a one, you know that the CPU type is a millions of instructions per second (MIPS). By adding into the *ElseIf* statements the results of your research into return codes for WMI CPU types, you enable the script to handle the work of finding out what kind of CPU your servers are running. After you've used all the *ElseIf* statements required to parse all the possible return codes, you add one more line to cover the potential of an unexplained code, and you use *Else* for that purpose.

Combine msgBox and CPU information

1. Open Notepad or the script editor of your choice.
2. Open the msgBox.vbs script and save it as YourNamePromptCPU.vbs.
3. At the very bottom of the newly renamed msgBox.vbs script, type **Sub subCPU**, as seen below:

   ```
   Sub subCPU
   ```

4. Open the CPUType.vbs script and copy the entire script to the clipboard.
5. Paste the entire CPUType.vbs script under the words *Sub subCPU*.
6. The first and second lines (from the CPUType.vbs script) that are pasted below the words *Sub subCPU* are not required in our subroutine. The two lines can be deleted. They are listed below:

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

7. Go to the bottom of the script you pasted under the words *Sub subCPU* and type **End sub**. We have now moved the CPU-type script into a subroutine. We will only enter this subroutine if the user presses the Yes button.
8. Under the code that evaluates the *vbYes* intrinsic constant, we want to add the line to call the subroutine. To do this, we simply type the name of the subroutine. That name is *subCPU*. The code to launch the script is seen below. Notice the only new code here is the word *subCPU*, everything else was already in the msgBox script.

   ```
   If intRTN = vbYes Then
   WScript.Echo "yes was pressed"
   subCPU
   ```

9. For every other button selection, we want to end the script. The command to do that is *WScript.quit*. We will need to type this command in three different places, as seen below:

```
ElseIf intRTN = vbNo Then
WScript.Echo "no was pressed"
WScript.quit
ElseIf intRTN = vbCancel Then
WScript.Echo "cancel was pressed"
WScript.quit
Else
WScript.Echo intRTN & " was pressed"
WScript.quit
End If
```

10. Save and run the script. Press the Yes button, and the results from CPUType should be displayed. Run the script two more times: Press No, and then press Cancel. On each successive running, the script should exit instead of running the script. If these are not your results, compare the script with the PromptCPU.vbs script in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03.

Quick Check

Q. How many *ElseIf* lines can be used in a script?

A. As many *ElseIf* lines as are needed.

Q. If more than one or two *ElseIf* lines are necessary, is there another construct that would be easier to use?

A. Yes. Use a *Select Case* type of structure.

Q. What is the effect of using *strComputer* = "." in a script?

A. The code *strComputer* is shorthand that means the local computer the script is executing on. It is used with WMI.

If...Then...Else

It is important to point out here that you can use *If...Then...Else* without the intervening *ElseIf* commands. In such a construction, you give the script the ability to make a choice between two options.

Just the Steps To use *If...Then...Else*

1. On a new line in the script, type **If *some condition* Then**.
2. On the next line, enter the command you want to invoke.
3. On the next line, type **Else**.

4. On the next line, type the alternate command you want to execute when the condition is not true.
5. On the next line, type **End If**.

The use of *If...Then...Else* is illustrated in the following code:

ifThenElse.vbs

```
Option Explicit
On Error Resume Next
Dim a,b,c,d
a = 1
b = 2
c = 3
d = 4
If a + b = d Then
  WScript.Echo (a & " + " & b & " is equal to " & d)
Else
  WScript.Echo (a & " + " & b & " is equal to " & c)
End If
```

In the preceding ifThenElse.vbs script, you declare your four variables on one line. You can do this for simple scripts such as this one. It can also be done for routine variables that are associated with one another, such as *objWMIService* and *objProcessor* from your earlier script. The advantage of putting multiple declarations on the same line is that it makes the script shorter. Although this does not really have an impact on performance, it can at times make the script easier to read. You'll need to make that call—does making the script shorter make the script easier to read, or does having each variable on a separate line with individual comments make the script easier to read?

When you do the *WScript.Echo* command, you're using a feature called *concatenation*, which puts together an output line by using a combination of variables and string text. Notice that everything is placed inside the parentheses and that the variables do not go inside quotation marks. To concatenate the text into one line, you can use the ampersand character (&). Because concatenation does not automatically include spaces, you have to put in the appropriate spaces inside the quotation marks. By doing this, you can include a lot of information in the output. This is one area that requires special attention when you're modifying existing scripts. You might need to change only one or two variables in the script, but modifying the accompanying text strings often requires the most work.

Using *If...Then...Else* to fix the syntax of output

1. Open the QueryAllProcessors.vbs script in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03 using Notepad or the script editor of your choice. Save it as YourNameQueryAllProcessorsSyntax.vbs.

2. Put a new function definition at the end of the script. (We will discuss user defined functions in just a few pages.) Use the word *Function* and give it the name *funIS*. Assign the input parameter the name *intIN*. The syntax for this line will look like the following:

   ```
   Function funIS(intIN)
   ```

3. Space down a few lines and end the function with the words *End Function*. This command will look like the following:

   ```
   End Function
   ```

4. Use *If...Then* to see if the *intIN* parameter is less than two. This line will look like:

   ```
   If intIN <2 Then
   ```

5. If the *intIN* parameter is less than two, then we want to assign the string "is a" the numeric value of *intIN* and the word *Processor*. The result will be that the script will use the singular form of the verb *to be*. This is seen in the line below:

   ```
   funIS = " is a " & intIN & " Processor "
   ```

6. If this is not the case, we will assign the string "are" and the number of processors to the name of the function. Finally we close out the *If...Then* construction by using *End If*. This is seen below:

   ```
   Else
   funIS = " are " & intIN & " Processors "
   End If
   ```

7. Right after we assign the *colItems* variable to contain the object that comes back from using the *execQuery* method, we want to retrieve the count of the number of items in *colItems*. We also want to build an output string that prints out a string stating how many processors are on the machine. We will use the *funIS* function to build up the remainder of the output line. It will look like the following:

   ```
   WScript.Echo "There" & funIS(colItems.count) & _
   "on this computer"
   ```

8. Save and run the script. You may want to compare your results with the QueryAllProcessorsSyntax.vbs script in the folder \My Documents\Microsoft Press\VBScriptSBS\ch03.

Select Case

When I see a *Select Case* statement in a script written in the VBScript language, my respect for the script writer goes up at least one notch. Most beginning script writers can figure out the *If...Then* statement, and some even get the *If...Then...Else* construction down. However, few master the *Select Case* construction. This is really a shame, because *Select Case* is both elegant and powerful. Luckily for you, I love *Select Case* and you will be masters of this construction by the end of this chapter!

Just the Steps To use *Select Case*

1. On a new line in the script, type **Select Case** and a variable to evaluate.
2. On the second line, type **Case 0**.
3. On the third line, assign a value to a variable.
4. On the next line, type **Case 1**.
5. On a new line, assign a value to a variable.
6. On the next line, type **End Select**.

In the following script, you again use WMI to obtain information about your computer. This script is used to tell us the role that the computer plays on a network (that is, whether it's a domain controller, a member server, or a member workstation). You need to use *Select Case* to parse the results that come back from WMI, because the answer is returned in the form of 0, 1, 2, 3, 4, or 5. Six options would be too messy for an *If...Then...ElseIf* construction. The text of ComputerRoles.vbs is listed here:

ComputerRoles.vbs

```
Option Explicit
'On Error Resume Next
Dim strComputer
Dim wmiRoot
Dim wmiQuery
Dim objWMIService
Dim colItems
Dim objItem

strComputer = "."
wmiRoot = "winmgmts:\\" & strComputer & "\root\cimv2"
wmiQuery = "Select DomainRole from Win32_ComputerSystem"

Set objWMIService = GetObject(wmiRoot)
Set colItems = objWMIService.ExecQuery _
  (wmiQuery)
For Each objItem in colItems
  WScript.Echo funComputerRole(objItem.DomainRole)
Next

Function funComputerRole(intIN)
 Select Case intIN
  Case 0
    funComputerRole = "Standalone Workstation"
  Case 1
    funComputerRole = "Member Workstation"
  Case 2
    funComputerRole = "Standalone Server"
  Case 3
    funComputerRole = "Member Server"
  Case 4
    funComputerRole = "Backup Domain Controller"
  Case 5
    funComputerRole = "Primary Domain Controller"
  Case Else
  funComputerRole = "Look this one up in SDK"
 End Select
End Function
```

Header Information

The Header information section of ComputerRoles.vbs is listed in the next bit of code. Notice that you start with the *Option Explicit* and *On Error Resume Next* statements, which are explained earlier in this chapter and in detail in Chapter 1, "Starting from Scratch." Next, you declare six variables. *wmiQuery* is, however, a different variable. You'll use it in the Reference information section, where you"ll assign a WMI query string to it. By declaring a variable and listing it separately, you can change the WMI query without having to rewrite the entire script.

objWMIService is used to hold your connection to WMI, and the variable *colItems* holds a collection of items that comes back from the WMI query. *objItem* is used to obtain an individual item from the collection. This was discussed in Chapter 2. The complete Header information section is listed below:

```
Option Explicit
On Error Resume Next
Dim strComputer
Dim wmiRoot
Dim wmiQuery
Dim objWMIService
Dim colItems
Dim objItem
```

Reference Information

The Reference information section assigns values to many of the variables named in the Header information part of ComputerRoles.vbs. The Reference information section of the script is listed here:

```
strComputer = "."
wmiRoot = "winmgmts:\\" & strComputer & "\root\cimv2"
wmiQuery = "Select DomainRole from Win32_ComputerSystem"
```

Two variables are unique to this script, the first of which is *wmiQuery*. In the Collection-OfDrives.vbs script discussed in Chapter 2, you embedded the WMI query in the *GetObject* command, which makes for a long line. By bringing the query out of the *GetObject* command and assigning it to the *wmiQuery* variable, you make the script easier to read and modify in the future. Next, you use the *colItems* variable and assign it to hold the object that is returned when you actually execute the WMI query.

Quick Check

Q. How is *Select Case* implemented?

A. *Select Case* begins with the *Select Case* command and a variable to be evaluated. However, it is often preceded by a *For Each* statement.

Q. How does *Select Case* work?

A. *Select Case* evaluates the test expression following the *Select Case* statement. If the result from this matches a value in any of the *Case* statements, it executes the code following that *Case* statement.

Q. What is the advantage of assigning a WMI query to a variable?

A. It provides the ability to easily use the script to query additional information from WMI.

Worker and Output Information

As mentioned earlier, WMI often returns information in the form of a collection (we talked about this in Chapter 2), and to work your way through a collection, you need to use the *For Each...Next* command structure to pull out specific information. In the Worker information section of ComputerRoles.vbs, you begin with making a connection into WMI. We do this by using *GetObject* to obtain a hook into WMI. Once we have made the connection into WMI, we then execute a WMI query and assign the resulting collection of items to a variable called *colItems*. We then use *For Each...Next* to pull one instance of an item from the collection so we can examine it. This is illustrated in code below. The interesting thing is the way we moved the *Select Case* statement from the middle of the script into a function called *funComputerRole*.

```
Set objWMIService = GetObject(wmiRoot)
Set colItems = objWMIService.ExecQuery _
  (wmiQuery)
For Each objItem in colItems
  WScript.Echo funComputerRole(objItem.DomainRole)
Next
```

funComputerRole function

To simplify the reading of the script, and to make it easier to maintain the script, we move the *Select Case* structure into a function we include at the bottom of the script. This moves the decision matrix out of the middle of the Worker section. It vastly simplifies the code (once, of course, you understand how a function works). In Figure 3-1, you can see that when we call the function, we use the name of the function. In parentheses, we include the parameter we wish to supply to the function. In the ComputerRoles.vbs script, we are retrieving a numeric value that corresponds to the role the computer plays in the network. If you look up the

WIN32_computerSystem class in the WMI Platform SDK (\My Documents\Microsoft Press\VBScriptSBS\Resources\WMI_SDKBook.chm), you will see an article that lists all the valuable properties this class can supply. I have copied the values for domain role into Table 3-1. Based on the chart that translates the values for computer role, I created the *funComputerRole* function.

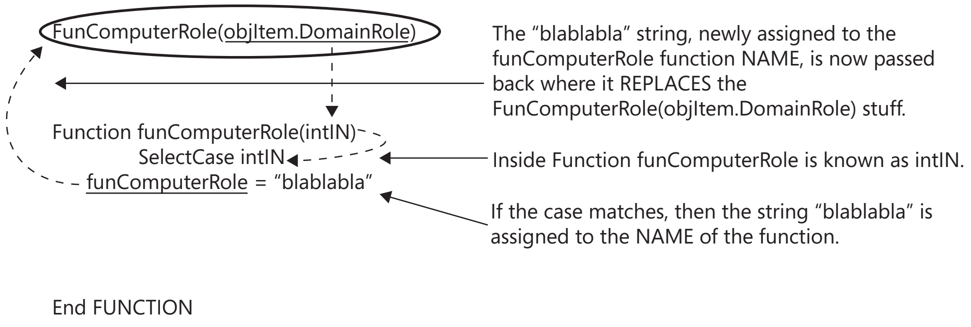
FunComputerRole(objItem.DomainRole)
The "blablabla" string, newly assigned to the funComputerRole function NAME, is now passed back where it REPLACES the FunComputerRole(objItem.DomainRole) stuff.
Function funComputerRole(intIN)
SelectCase intIN
funComputerRole = "blablabla"
Inside Function funComputerRole is known as intIN.
If the case matches, then the string "blablabla" is assigned to the NAME of the function.
End FUNCTION

Figure 3-1 Assign value to name of the function

In Figure 3-1, you see how the numeric value of the computer role is passed to the *funComputerRole* function. Once the number is inside the function, it is stored in the variable *intIN*. The *Select Case* statement evaluates the value of *intIN* and assigns the appropriate string to the name of the function itself. As you can see in Figure 3-1, the value that is assigned to the name of the function is passed back to the line of code that called the function in the first place. The *funComputerRole* function is listed below.

```
Function funComputerRole(intIN)
 Select Case intIN
  Case 0
    funComputerRole = "Standalone Workstation"
  Case 1
    funComputerRole = "Member Workstation"
  Case 2
    funComputerRole = "Standalone Server"
  Case 3
    funComputerRole = "Member Server"
  Case 4
    funComputerRole = "Backup Domain Controller"
  Case 5
    funComputerRole = "Primary Domain Controller"
  Case Else
  funComputerRole = "Look this one up in SDK"
 End Select
End Function
```

To find out how the *DomainRole* field is structured, you need to reference the Platform SDK for Microsoft Windows Server 2003. You will also be able to find other properties you can use to expand upon the ComputerRoles.vbs script. The value descriptions for domain roles are shown in Table 3-1.

Table 3-1 WMI Domain Roles from *Win32_ComputerSystem*

Value	Meaning
0	Standalone workstation
1	Member workstation
2	Standalone server
3	Member server
4	Backup domain controller
5	Primary domain controller

The first line of the *Select Case* statement contains the test expression—the number representing the role of the computer on the network. Each successive *Case* statement is used to evaluate the test expression, and to identify the correct computer role. The first of these statements is seen here:

```
Case 0
  strComputerRole = "Standalone Workstation"
```

The *strComputerRole* variable will be assigned the phrase “Standalone Workstation” if the text expression (intIN) is equal to 0. You will then use *strComputerRole* to echo out the role of the computer in the domain when we exit the *Select Case* construction.

You end the *Select Case* construction with *End Select*, similarly to the way you ended the *If...Then* statement with *End If*. After you use *End Select*, you use the *WScript.Echo* command to send the value of *strComputerRole* out to the user. Remember that the entire purpose of the *Select Case* construction in ComputerRoles.vbs is to find and assign the *DomainRole* value to the *strComputerRole* variable. After this is accomplished, you use the *Next* command to feed back into the *For Each* loop used to begin the script.

Modifying CPUType.vbs Step-by-Step Exercises

In this section, you will modify CPUType.vbs so that it uses a *Select Case* format instead of multiple *If...Then...ElseIf* statements. This is a valuable skill, because many of the scripts you will find have a tendency to use multiple *If...Then...ElseIf* statements. As you will see, it is relatively easy to make the modification to using *Select Case*. The key to success is to remove as little of the original code as possible.

1. Open CPUTypeStarter.vbs and save it as YourNameCPUType.vbs. It is located in the \My Documents\Microsoft Press\VBScriptSBS\Ch03\StepByStep folder.
2. Turn off *On Error Resume Next* by commenting out the line.

3. Turn the *If...Then* line into a *Select Case* statement. The only element you must keep from this line is *objProcessor.Architecture,* because it is hard to type. When you are finished, your *Select Case* line looks like the following:

   ```
   Select Case objProcessor.Architecture
   ```

4. Start your case evaluation. If *objProcessor.Architecture = 0*, you know that the processor is an x86. So your first case is *Case 0*. That is all you put on the next line. It looks like this:

   ```
   Case 0
   ```

5. Leave the *WScript.Echo* line alone.
6. *ElseIf objProcessor.Architecture = 1* becomes *Case 1*, which is a MIPS CPU. Delete the entire *ElseIf* line and enter **Case 1**.
7. Leave the *WScript.Echo* line alone.

   ElseIf objProcessor.Architecture = 2 becomes simply *Case 2*, as you can see here:

   ```
   Case 2
   ```

   Up to this point, your *Select Case* configuration looks like the following:

   ```
   Select Case objProcessor.Architecture
   Case 0
     WScript.Echo "This is an x86 cpu."
   Case 1
     WScript.Echo "This is a MIPS cpu."
   Case 2
     WScript.Echo "This is an Alpha cpu."
   ```

8. Modify the "*ElseIf objProcessor.Architecture = 3 Then*" line so that it becomes *Case 3*.
9. Leave the *WScript.Echo* line alone.

   The next case is *not* Case 4, but rather Case 6, because you modify the following line: "*ElseIf objProcessor.Architecture = 6 Then*". The *Select Case* construction now looks like the following:

   ```
   Select Case objProcessor.Architecture
     Case 0
       WScript.Echo "This is an x86 cpu."
     Case 1
       WScript.Echo "This is a MIPS cpu."
     Case 2
       WScript.Echo "This is an Alpha cpu."
     Case 3
       WScript.Echo "This is a PowerPC cpu."
     Case 6
       WScript.Echo "This is an ia64 cpu."
   ```

10. You have one more case to evaluate, and it will take the place of the *Else* command, which encompasses everything else that has not yet been listed. You implement *Case Else* by changing the *Else* to *Case Else*.

11. Leave the line *WScript.Echo "Cannot determine cpu type"* alone.
12. Change *End If* to *End Select*. Now you're finished with the conversion of *If…Then…ElseIf* to *Select Case*.
13. Save the file and run the script. If you need assistance, refer to the CPUTypeSolution.vbs script in the same folder you found the starter script.

One Step Further: Modifying ComputerRoles.vbs

In this lab, you'll modify ComputerRoles.vbs so that you can use it to turn on Dynamic Host Configuration Protocol (DHCP) on various workstations. This is the first script we use that calls a WMI method.

Scenario

Your company's network was set up by someone who really did not understand DHCP. In fact, the person who set up the network probably could not even spell DHCP, and as a result every workstation on the network is configured with a static IP address. This was bad enough when the network only had a hundred workstations, but the network has grown to more than three hundred workstations within the past couple of years. The Microsoft Excel spreadsheet that used to keep track of the mappings between computer names and IP addresses is woefully out of date, which in the past month alone has resulted in nearly 30 calls to the help desk that were traced back to addressing conflicts. To make matters worse, some of the helpful administrative assistants have learned to change the last octet in Transmission Control Protocol/Internet Protocol (TCP/IP) properties, which basically negates any hope of ever regaining a managed network. Your task, if you should choose to accept it, is to create a script (or scripts) that will do the following:

- Use WMI to determine the computer's role on the network and to print out the name of the computer, the domain name (if it is a member of a domain), and the user that belongs to the computer
- Use WMI to enable DHCP on all network adapters installed on the computer that use TCP/IP

Your research has revealed that you can use *Win32_ComputerSystem* WMI class to obtain the information required in the first part of the assignment.

Warning Keep in mind, this script will change network settings on the machine that this script runs on. Also, when run, it will need administrator rights to make the configuration changes. If you do not wish to change your TCP/IP settings, then do not run this script on your machine.

Part A

1. Open up the ComputerRoles.vbs file from \My Documents\Microsoft PressVB ScriptSBS\Ch03\OneStepFurther and save it as YourNameComputerRoles Solution.vbs.
2. Comment out *On Error Resume Next* so that you will receive some meaningful feedback from the Windows Script Host (WSH) run time.
3. *Dim* new variables to hold the following items:
   - ❑ *strcomputerName*
   - ❑ *strDomainName*
   - ❑ *strUserName*
4. Modify *wmiQuery* so that it returns more than just *DomainRole* from *Win32_ComputerSystem*. Hint: Change *DomainRole* to a wildcard such as *. The original wmiQuery line is seen below:

   ```
   wmiQuery = "Select DomainRole from Win32_ComputerSystem"
   ```

   The new line looks like this:

   ```
   "Select * from Win32_ComputerSystem"
   ```
5. Because *colComputers* is a collection, you can't directly query it. You'll need to use *For Each...Next* to give yourself a single instance to work with. Therefore, the assignment of your new variables to actual items will take place inside the *For Each...Next* loop. Assign each of your new variables in the following manner:
   - ❑ *strComputerName = objComputer.name*
   - ❑ *strDomainName = objComputer.Domain*
   - ❑ *strUserName = objComputer.UserName*
6. After the completion of the *Select Case* statement (*End Select*) but before the *Next* command at the bottom of the file, use *WScript.Echo* to return the four items required by the first part of the lab scenario. Use concatenation (by using the ampersand) to put the four variables on a single line. Those four items are declared as follows:
   - ❑ *Dim strComputerRole*
   - ❑ *Dim strcomputerName*
   - ❑ *Dim strDomainName*
   - ❑ *Dim strUserName*
7. Save the file and run it.

8. Modify the script so that each variable is returned on a separate line. Hint: Use the intrinsic constant *vbCrLf* and the ampersand to concatenate the line. It will look something like this:

   ```
   strComputerRole & vbCrLf & strComputerName
   ```

9. Save and run the file.
10. Use *WScript.Echo* to add and run a complete message similar to the following:

    ```
    WScript.Echo("all done")
    ```

11. Save and run your script. If it does not run properly, compare it with \My Documents\Microsoft PressVBScriptSBS\ch03\StepByStep\ComputerRoles Solution.vbs.

Part B

1. Open the YourNameComputerRolesSolution.vbs file and save it as YourNameEnableDHCPSolution.vbs.
2. Comment out *On Error Resume Next* so that you will receive some meaningful feedback from the WSH run time.
3. *Dim* new variables to hold the new items required for this script. Hint: You can rename the following items:
   - *Dim colComputers*
   - *Dim objComputer*
   - *Dim strComputerRole*
4. The new variables are listed here:
   - *colNetAdapters*
   - *objNetAdapter*
   - *DHCPEnabled*
5. Modify the *wmiQuery* so that it looks like the following:

   ```
   wmiQuery = "Select * from Win32_NetworkAdapterConfiguration where IPEnabled=TRUE"
   ```

6. Change the following *Set* statement:

   ```
   Set colComputers = objWMIService.ExecQuery (wmiQuery)
   ```

   Now, instead of using *colComputers*, the statement uses *colNetAdapters*. The line will look like the following:

   ```
   Set colNetAdapters = objWMIService.ExecQuery (wmiQuery)
   ```

7. Delete the *Select Case* construction. It begins with the following line:

   ```
   Select Case objComputer.DomainRole
   ```

   And it ends with *End Select*.

8. You should now have the following:

   ```
   For Each objComputer In colComputers
     WScript.Echo strComputerRole
   Next
   ```

9. Change the *For Each* line so that it reads as follows:

   ```
   For Each objNetAdapter In colNetAdapters
   ```

10. Assign *DHCPEnabled* to *objNetAdapter.EnableDHCP()*. You can do it with the following:

    ```
    DHCPEnabled = objNetAdapter.EnableDHCP()
    ```

11. Use *If…Then…Else* to decide whether the operation was successful. If DHCP is enabled, *DHCPEnabled* will be *0*, and you want to use *WScript.Echo* to echo out that the DHCP is enabled. The code looks like the following:

    ```
    If DHCPEnabled = 0 Then
      WScript.Echo "DHCP has been enabled."
    ```

12. If *DHCPEnabled* is not set to *0*, the procedure does not work. So you have your *Else* condition. It looks like the following:

    ```
    Else
      WScript.Echo "DHCP could not be enabled."
    End If
    ```

13. Conclude the script by using the *Next* command to complete the *If…Then…Next* construction. You don't have to put in a closing echo command, because you're getting feedback from the *DHCPEnabled* commands.

14. Save and run the script. Compare your script with the EnableDHCPSolution.vbs script in the \My Documents\Microsoft Press\VBScriptSBS\ch03\OneStepFurther folder.

Chapter 3 Quick Reference

To	Do This
Evaluate a condition using *If…Then*	Place the condition to be evaluated between the words *If* and *Then*
Evaluate one condition with two outcomes	Use *If…Then…Else*
End an *If…Then…Else* statement	Use *End If*
Use an intrinsic constant in a script	Type it into the code (it does not need to be declared, or otherwise defined)
Evaluate one condition with three outcomes	Use *If…Then…ElseIf*, or use *Select Case*
Evaluate one condition with four potential outcomes	Use *Select Case*

Chapter 15

Using Subroutines and Functions

Before You Begin

To work through the material presented in this chapter, you need to be familiar with the following concepts from earlier chapters:

- Reading text files
- Writing to text files
- Creating files
- Creating folders
- Using the *For...Next* statement
- Creating the *Select Case* statement
- Connecting to Microsoft Active Directory directory service
- Reading information from WMI

After completing this chapter, you will be able to:

- Convert inline code into a subroutine
- Call subroutines
- Perform Active Directory user management

Working with Subroutines

In this section, you'll learn about how network administrators use subroutines. For many readers, the use of subroutines will be somewhat new territory and might even seem unnecessary, particularly when you can cut and paste sections of working code. But before we get into the *how-to*, let's go over the *what*.

A subroutine is a named section of code that gets run only when something in the script calls it by name. Nearly every script we've worked with thus far has been a group of commands, which have been processed from top to bottom in a consecutive fashion. Although this consecutive processing approach, which I call *linear scripting*, makes the code easy for the network administrator to work with, it does not always make his work very efficient. In addition, when you need to perform a similar activity from different parts of the script, using the inline cut-and-paste scripting approach quickly becomes inefficient and hard to understand. This is where subroutines come into play. A subroutine is not executed when its body is defined in the code; instead, it is executed only when it is called by name. If you define a subroutine, and use it only one time, you might make your script easier to read or easier to maintain, but you will not make the script shorter. If, however, you have something you want to do over and over, the subroutine does make the script shorter. The following summarizes uses for a subroutine in Microsoft Visual Basic, Scripting Edition (VBScript):

- Prevents needless duplication of code
- Makes code portable and reusable
- Makes code easier to troubleshoot and debug
- Makes code easier to read and maintain
- Makes code easier to modify

The following script (LinearScript.vbs) illustrates the problem with linear scripting. In this script are three variables: *a*, *b*, and *c*. Each of these is assigned a value, and you need to determine equality. The script uses a series of *If Then...Else* statements to perform the evaluative work. As you can see, the code gets a little redundant by repeating the same statements several times.

LinearScript.vbs

```
Option Explicit
Dim a
Dim b
Dim c

a=1
b=2
c=3

If a = b Then
WScript.Echo a & " and " & b & " are equal"
Else
WScript.Echo a & " and " & b & " are not equal"
End If

If b = c Then
WScript.Echo b & " and " & c & " are equal"
Else
WScript.Echo b & " and " & c & " are not equal"
End If

If a + b = c Then
WScript.Echo a+b & " and " & c & " are equal"
Else
WScript.Echo a+b & " and " & c & " are not equal"
End If
```

OK, so the script might be a little redundant, although if you're paid to write code by the line, this is a great script! Unfortunately, most network administrators are not paid by the line for the scripts they write. This being the case, clearly you need to come up with a better way to write code. (I am telegraphing the solution to you now...) That's right! You will use a subroutine to perform the evaluation. The modified script uses a subroutine to perform the evaluation of the two numbers. This results in saving two lines of code for each evaluation performed. In this example, however, the power is not in saving a few lines of code—it's in the fact that you use one section of code to perform the evaluation. Using one section makes the script easier to read and easier to write.

Note Business rules is a concept that comes up frequently in programming books. The idea is that many programs have concepts that are not technical requirements but still must be adhered to. These are nontechnical rules. For instance, a business rule might say that when a payment is not received within 30 days after the invoice is mailed, a follow-up notice must be sent out, and a 1 percent surcharge is added to the invoice amount. Because businesses sometimes change these nontechnical requirements, such rules would be better incorporated into a separate section of the code (a subroutine, for example) as opposed to sprinkling them throughout the entire program. If the business later decides to charge an additional 1 percent surcharge after 60 days, this requirement can be easily accommodated in the code.

In the script you are currently examining, your business rules are contained in a single code section, so you can easily modify the code to incorporate new ways of comparing the three numbers (to determine, for example, that they are not equal instead of equal). If conditions are likely to change or additional information might be required, creating a subroutine makes sense.

Quick Check

Q. To promote code reuse within a script, where is one place you can position the code?

A. You can place the code within a subroutine.

Q. To make changing business rules easier to update, where is a good place to position the rules?

A. You can place business rules within a subroutine to make them easier to update.

Calling the Subroutine

In the next script you'll examine, SubRoutineScript.vbs, the comparison of *a*, *b*, and *c* is done by using a subroutine called *Compare*. To use a subroutine, you simply place its name on a line by itself. Notice that you don't need to declare the name of the subroutine because it isn't a variable. So, the script works even though you specified *Option Explicit* and did not declare the name used for the subroutine. In fact, you cannot declare the name of your subroutine. If you do, you will get a "name redefined" error.

Creating the Subroutine

Once you decide to use a subroutine, the code for creating it is very light. Indeed, all that is required is the word *Sub* followed by the name you will assign to the subroutine. In the SubRoutineScript.vbs script, the subroutine is assigned the name *Compare* by the following line: *Sub Compare*. That's all there is to it. You then write the code that performs the comparison and end the subroutine with the command *End Sub*. After you do all that, you have your subroutine.

SubRoutineScript.vbs

```
Option Explicit
Dim a, b, c
Dim num1, num2
a=1
b=2
c=3

num1 = a
num2 = b
compare

num1 = b
num2 = c
compare

num1 = a + b
num2 = c
compare

Sub Compare
If num1 = num2 Then
WScript.Echo (num1 & " and " & num2 & " are equal")
Else
WScript.Echo(num1 & " and " & num2 & " are not equal")
End If
End Sub
```

Just the Steps To create a subroutine

1. Begin the line of code with the word *Sub* followed by name of the subroutine.
2. Write the code that the subroutine will perform.
3. End the subroutine by using the *End Sub* command on a line by itself.

Creating Users and Logging Results

As your scripts become more powerful, they have a tendency to become longer and longer. The next script, CreateUsersLogAction.vbs, is nearly 80 lines long. The reason for this length is that you perform three distinct actions. First, you read a text file and parse the data into an array. Then you use this array to create new users and add the users into an existing group in Active Directory. As you create users and add them to groups, you want to create a log file and write the names of the created users. All the code to perform these actions begins to add up and can make a long script hard to read and understand. The subroutine becomes rather useful in such a situation. In fact, the subroutine used to create the log file is nearly 30 lines long itself because you need to check whether the folder exists or the log file exists. If the folder or file does not exist, you need to create it. If each is present, you need to open the file and append data to it. By placing this code into a subroutine, you are able to access it each time you loop through the input data you're using to create the users in the first place. After the user is created, you go to the subroutine, open the file, write to it, close the file, and then go back into *Do Until...Loop* to create the next user.

Note Holding the text file open might seem like an easier approach than closing the file, but I prefer to close the file after each loop so that I can guarantee the consistency of the file as a log of the accounts that are being created. Closing the file offers other benefits as well. It makes the operation more modular and therefore promotes portability. Making an open and a close part of the routine hides complexity that could arise.

If you kept the file open and wrote to the log file in an asynchronous manner, your log writer could get behind, and in the event of an anomaly, your log might not be an accurate reflection of the actual accounts created on the server. Here is the CreateUsersLogAction.vbs script.

CreateUsersLogAction.vbs

```
Option Explicit
On Error Resume Next
Dim objOU
Dim objUser
Dim objGroup
Dim objFSO
Dim objFile
Dim objFolder
Dim objTextFile
Dim TxtIn
Dim strNextLine
Dim i
Dim TxtFile
Dim LogFolder
Dim LogFile

TxtFile = "C:\UsersAndGroups.txt"
LogFolder = "C:\FSO"
LogFile = "C:\FSO\fso.txt"
Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objTextFile = objFSO.OpenTextFile _
  (TxtFile, ForReading)
```

```
Do Until objTextFile.AtEndOfStream
  strNextLine = objTextFile.ReadLine
  TxtIn = Split(strNextLine , ",")
  Set objOU = GetObject("LDAP://OU=mred," _
    & "dc=nwtraders,dc=msft")
  Set objUser = objOU.Create("User", "cn="& TxtIn(0))
  objUser.Put "sAMAccountName", TxtIn(0)
  objUser.SetInfo

  Set objGroup = GetObject _
    ("LDAP://CN="& TxtIn(1) & ",cn=users," _
    & “dc=nwtraders,dc=msft”)
  objGroup.add _
    "LDAP://cn="& TxtIn(0) & ",ou=Mred," _
    & "dc=nwtraders,dc=msft"
  Logging
Loop

Sub Logging
  If objFSO.FolderExists(LogFolder) Then
    If objFSO.FileExists(LogFile) Then
      Set objFile = objFSO.OpenTextFile _
         (LogFile, ForAppending)
      objFile.WriteBlankLines(1)
      objFile.Writeline "Creating User " & Now
      objFile.Writeline TxtIn(0)
      objFile.Close
    Else
      Set objFile = objFSO.CreateTextFile(LogFile)
      objFile.Close
      Set objFile = objFSO.OpenTextFile _
         (LogFile, ForWriting)
      objFile.WriteLine "Creating User " & Now
      objFile.WriteLine TxtIn(0)

      objFile.Close
    End If
  Else
    Set objFolder = objFSO.CreateFolder(LogFolder)
    Set objFile = objFSO.CreateTextFile(LogFile)
    objFile.Close
    Set objFile = objFSO.OpenTextFile _
       (LogFile, ForWriting)
    objfile.WriteLine "Creating User " & Now
    objFile.WriteLine TxtIn(0)
    objFile.Close
  End If
End Sub

WScript.Echo("all done")
```

Header Information

The Header information section of CreateUsersLogAction.vbs is used to declare all the variables used in the script. Thirteen variables are used in the script. The variable *i* is a simple counter and is not listed in Table 15-1 with the other variables.

Table 15-1 Variables Used in CreateUsersLogAction.vbs

Variable	Description
objOU	Holds connection to target OU in Active Directory.
objUser	Holds hook for *Create user* command; takes *TxtIn(0)* as input for user name.
objGroup	Holds hook for the *add* command; takes *TxtIn(1)* as input for name of group and *TxtIn(0)* as name of user to add.
objFSO	Holds hook that comes back from the *CreateObject* command used to create the *FileSystemObject*.
objFile	Holds hook that comes back from the *OpenTextFile* command issued to *objFSO*.
objFolder	Holds hook that comes back from the *CreateFolder* command issued to *objFSO* if the folder does not exist.
objTextFile	Holds the data stream that comes from the *OpenTextFile* command that is used to open the UsersAndGroups.txt file.
TxtIn	An array that is created from parsing *strNextLine*. Each field split by the comma becomes an element in the array. Holds user name to be created and the group that the user is to be added to.
strNextLine	Holds one line worth of data from the UsersAndGroups.txt file.
TxtFile	Holds path and name of text file to be parsed as input data.
LogFolder	Holds path and name of folder used to hold logging information.
LogFile	Holds path and name of text file to be used as the log file.

Reference Information

The Reference information section of the script is used to assign values to some of the variables in the script. In addition to the mundane items such as defining the path and title for the text file used to hold the users and groups, in this section, you create three constants that are used in working with text files.

Note If you create standard variable names, and you consistently use them in your scripts, you will make it easier to reuse your subroutines without any modification. For instance, if you use *objFSO* consistently for creating *FileSystemObject*, you minimize the work required to "rewire" your subroutine. Of course, using the Find and Replace feature of Microsoft Notepad, or any other script editor, makes it rather easy to rename variables.

These constants are *ForReading*, *ForWriting*, and *ForAppending*. The use of these constants was discussed in detail in Chapter 4, "Working with Arrays." The last two tasks done in the Reference information section of the script are creating an instance of the *FileSystemObject* and using the *OpenTextFile* command so that you can read it in the list of users that need to be created and the group to which each user will be assigned. Here is the Reference information section of the script:

```
TxtFile = "C:\UsersAndGroups.txt"
LogFolder = "C:\FSO"
LogFile = "C:\FSO\fso.txt"
Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objTextFile = objFSO.OpenTextFile _
  (TxtFile, ForReading)
```

Worker Information

The Worker information section of the script is where the users are actually created and assigned to a group. To work through the UsersAndGroups.txt file, you need to make a connection to the file. This was done in a previous Reference information section of the script, in which we assigned *objTextFile* to be equal to the hook that came back once the connection into the file was made. Think back to the pipe analogy (in Chapter 5, “More Arrays”), in which you set up a pump and pulled the text, one line at a time, into a variable called *strNextLine*. As long as data is in the text file, you can continue to pump the information by using the *ReadLine* command. However, if you reach the end of the text stream, you exit the *Do Until...Loop* statement you created.

```
Do Until objTextFile.AtEndOfStream
  strNextLine = objTextFile.ReadLine
  TxtIn = Split(strNextLine , ",")
  Set objOU = GeCreateUsersLogAction.vbstObject("LDAP://OU=Mred," _
    & "dc=nwtraders,dc=msft")
  Set objUser = objOU.Create("User", "cn="& TxtIn(0))
  objUser.Put "sAMAccountName", TxtIn(0)
  objUser.SetInfo

  Set objGroup = GetObject _
    ("LDAP://CN="& TxtIn(1) & ",cn=users," _
    & "dc=nwtraders,dc=msft")
  objGroup.add _
    "LDAP://cn="& TxtIn(0) & ",ou=Mred," _
    & "dc=nwtraders,dc=msft"
  Logging
Loop
```

Output Information

Once you create a new user and assign that user to a group, you need to log the script changes. To do this, you call a subroutine (in our script, called *Logging*) that opens a log file and writes the name of the new user that was created as well as the time in which the creation occurred. The first task the *Logging* subroutine does is check for the existence of the logging folder that is defined by the variable *LogFolder*. To check for the presence of the folder, you use the *Folder-Exists* method. If the folder is present on the system, you next check for the existence of the logging file defined by the *LogFile* variable. To check for the presence of the logging file, you use the *FileExists* method. If both of these conditions are copasetic, you open the log file by using the *OpenTextFile* command and specify that you will append to the file instead of over-writing it (which is normally what you want a log file to do). In writing to the file, you use two different methods: *WriteBlankLines* to make the log a little easier to read, and *WriteLine* to write the user name and the time that user was created in the log.

If, on the other hand, the log folder exists but the log file does not exist, you need to create the log file prior to writing to it. This is the subject of the first *Else* command present in the sub-routine. You use the *CreateTextFile* command and the *LogFile* variable to create the log file. After the file is created, you must close the connection to the file; if you do not, you get an error message stating that the file is in use. After you close the connection to the log file, you reopen it by using the *OpenTextFile* command, and then you write your information to the file.

The other scenario our subroutine must deal with is if neither the folder nor the log file is in existence, in which case you have to create the folder (by using the *CreateFolder* method) and then create the file (by using the *CreateTextFile* method). It is necessary to use *objFile.Close* to close the connection to the newly created text file so that you can write your logging informa-tion to the file. Once you write to the log file, you exit the subroutine by using the *End Sub* command, and you enter *Do Until...Loop* again. The *Logging* subroutine is shown here:

```
Sub Logging
  If objFSO.FolderExists(LogFolder) Then
    If objFSO.FileExists(LogFile) Then
      Set objFile = objFSO.OpenTextFile _
         (LogFile, ForAppending)
      objFile.WriteBlankLines(1)
      objFile.WriteLine "Creating User " & Now
      objFile.WriteLine TxtIn(0)
      objFile.Close
    Else
      Set objFile = objFSO.CreateTextFile(LogFile)
      objFile.Close
      Set objFile = objFSO.OpenTextFile _
         (LogFile, ForWriting)
      objfile.WriteLine "Creating User " & Now
      objFile.WriteLine TxtIn(0)

      objFile.Close
    End If
  Else
    Set objFolder = objFSO.CreateFolder(LogFolder)
    Set objFile = objFSO.CreateTextFile(LogFile)
    objFile.Close
    Set objFile = objFSO.OpenTextFile _
      (LogFile, ForWriting)
    objfile.WriteLine "Creating User " & Now
    objFile.WriteLine TxtIn(0)
    objFile.Close
  End If
End Sub

WScript.Echo("all done")
```

Using a subroutine to retrieve service information from WMI

1. Open the \My Documents\Microsoft Press\VBScriptSBS\Templates\wmiTemplate.vbs script in Notepad or your favorite script editor and save it as YourNameListServicesIn-Processes.vbs.
2. Comment out *On Error Resume Next.*
3. Edit the *wmiQuery* line so that you are selecting only *processID* and the name from *win32_Process*. You want to, however, exclude the process ID that equals 0. This query will look like the following:

   ```
   wmiQuery = "Select processID, name from win32_Process where processID <> 0"
   ```

4. In the Header section of the script, declare a new variable to hold the process ID. Call this variable *intPID.*
5. Inside the *For Each...Next* loop, delete all the *WScript.Echo* ": " & *objItem* lines except for one. The completed *For Each...Next* loop will look like the following:

   ```
   For Each objItem in colItems
       WScript.Echo ": " & objItem.
   Next
   ```

6. Modify the *WScript.Echo* line so that it prints out the name and the processed properties from the *win32_Process* class. Include labels with each property to identity each of the properties as belonging to a process. My code to do this looks like the following:

   ```
   WScript.Echo "Process Name: " & objItem.Name & " ProcessID: " & objItem.ProcessID
   ```

7. Store the value of the process ID in the variable *intPID*. This is seen below:

   ```
   intPID = objItem.ProcessID
   ```

8. Save and run YourNameListServicesInProcesses.vbs. There should be no errors at this point, and you should see an output similar to the following (of course your list of processes will be different). The process names and the process IDs seen in the output from your script will compare to the ones seen in Taskmanager.exe (refer to Figure 15-1).

   ```
   Process Name: System ProcessID: 4
   Process Name: smss.exe ProcessID: 776
   Process Name: csrss.exe ProcessID: 868
   Process Name: winlogon.exe ProcessID: 892
   Process Name: services.exe ProcessID: 940
   Process Name: lsass.exe ProcessID: 952
   Process Name: svchost.exe ProcessID: 1136
   ```

```
Process Name: svchost.exe ProcessID: 1228
Process Name: svchost.exe ProcessID: 1352
Process Name: spoolsv.exe ProcessID: 1640
Process Name: explorer.exe ProcessID: 832
Process Name: wmiprvse.exe ProcessID: 632
Process Name: wmiprvse.exe ProcessID: 1436
Process Name: WINWORD.EXE ProcessID: 3764
Process Name: svchost.exe ProcessID: 2428
Process Name: PrimalScript.exe ProcessID: 2872
Process Name: cscript.exe ProcessID: 584
Process Name: wmiprvse.exe ProcessID: 3652
```

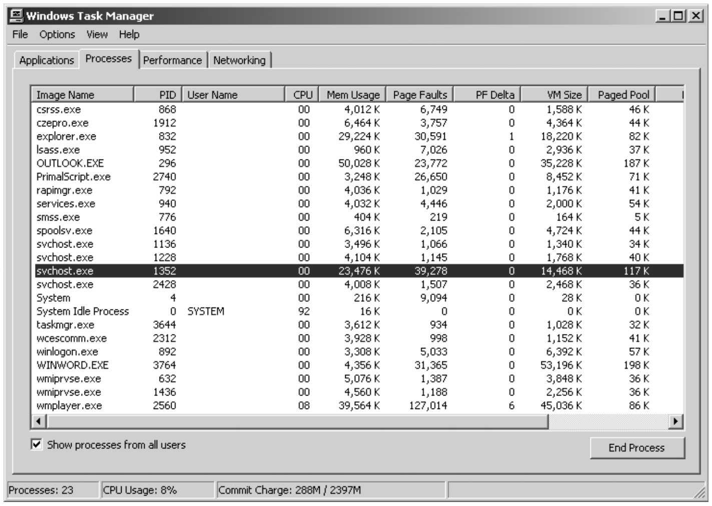
Windows Task Manager
File Options View Help
Applications Processes Performance Networking
Show processes from all users
End Process

Image Name	PID	User Name	CPU	Mem Usage	Page Faults	PF Delta	VM Size	Paged Pool
csrss.exe	868		00	4,012 K	6,749	0	1,588 K	46 K
czepro.exe	1912		00	6,464 K	3,757	0	4,364 K	44 K
explorer.exe	832		00	29,224 K	30,591	1	18,220 K	82 K
lsass.exe	952		00	960 K	7,026	0	2,936 K	37 K
OUTLOOK.EXE	296		00	50,028 K	23,772	0	35,228 K	187 K
PrimalScript.exe	2740		00	3,248 K	26,650	0	8,452 K	71 K
rapimgr.exe	792		00	4,036 K	1,029	0	1,176 K	41 K
services.exe	940		00	4,032 K	4,446	0	2,000 K	54 K
smss.exe	776		00	404 K	219	0	164 K	5 K
spoolsv.exe	1640		00	6,316 K	2,105	0	4,724 K	44 K
svchost.exe	1136		00	3,496 K	1,066	0	1,340 K	34 K
svchost.exe	1228		00	4,104 K	1,145	0	1,768 K	40 K
svchost.exe	1352		00	23,476 K	39,278	0	14,468 K	117 K
svchost.exe	2428		00	4,008 K	1,507	0	2,468 K	36 K
System	4		00	216 K	9,094	0	28 K	0 K
System Idle Process	0	SYSTEM	92	16 K	0	0	0 K	0 K
taskmgr.exe	3644		00	3,612 K	934	0	1,028 K	32 K
wcescomm.exe	2312		00	3,928 K	998	0	1,152 K	41 K
winlogon.exe	892		00	3,308 K	5,033	0	6,392 K	57 K
WINWORD.EXE	3764		00	4,356 K	31,365	0	53,196 K	198 K
wmiprvse.exe	632		00	5,076 K	1,387	0	3,848 K	36 K
wmiprvse.exe	1436		00	4,560 K	1,188	0	2,256 K	36 K
wmplayer.exe	2560		08	39,564 K	127,014	6	45,036 K	86 K

Processes: 23 CPU Usage: 8% Commit Charge: 288M / 2397M

Figure 15-1 Taskmanager.exe view of processes and services; Svchost is comprised of several services.

9. Create a subroutine called *SubGetServices*. This is seen below:

```
' *** subs are below ***

Sub subGetServices

End Sub
```

10. Inside your new subroutine, declare three variables that will be used in the secondary Windows Management Instrumentation (WMI) query. One variable will contain the query, one will contain the collection returned by the query, and one will hold the individual service instances retrieved from the collection. The three variables you want to declare correspond to the variables used in the main body of the script: *wmiQuery1*, *colItems1*, *objItem1*.

```
Dim wmiQuery1
Dim colItems1
Dim objItem1
```

11. Under the new variables you declared in the subroutine, assign a new query to *wmiQuery1* that selects the name from WIN32_Service, where *processID* is the same as the one stored in *intPID*. This query will look like the following:

```
wmiQuery1 = "Select name from win32_Service where processID = " & intPID
```

12. On the line following *wmiQuery1*, use the *colItems1* variable to hold the collection that is returned by using the *execQuery* method to execute the query contained in *wmiQuery1*. This code will look like the following:

```
Set colItems1 = objWMIService.ExecQuery(WmiQuery1)
```

13. Use *For Each...Loop* to walk through *colItems1*. Use *WScript.Echo* to print out the name of each service that corresponds to the query defined in *wmiQuery1*. This is seen below:

```
For Each objItem1 In colItems1
  WScript.Echo vbTab, "Service Name: ", objItem1.Name
Next
```

14. In the main body of the script, after the line where the process ID is assigned to *intPID*, call the subroutine you just created. The placement of the call to *subGetServices* is seen below:

```
intPID = objItem.ProcessID
subGetServices
```

15. Save and run your script. You should see a printout that now lists services running inside processes. Compare your results with the listing below. If you do not see something like this (realizing the processes and services will be different), compare your script with \My Documents\Microsoft Press\VBScriptSBS\ch15\ListServicesIn Processes.vbs.

```
Process Name: System ProcessID: 4
Process Name: smss.exe ProcessID: 776
Process Name: csrss.exe ProcessID: 868
Process Name: winlogon.exe ProcessID: 892
Process Name: services.exe ProcessID: 940
 Service Name: Eventlog
 Service Name: PlugPlay
Process Name: lsass.exe ProcessID: 952
 Service Name: ProtectedStorage
 Service Name: SamSs
Process Name: svchost.exe ProcessID: 1136
 Service Name: DcomLaunch
Process Name: svchost.exe ProcessID: 1228
 Service Name: RpcSs
Process Name: svchost.exe ProcessID: 1352
 Service Name: AudioSrv
 Service Name: CryptSvc
 Service Name: EventSystem
 Service Name: Nla
 Service Name: RasMan
 Service Name: SENS
 Service Name: ShellHWDetection
 Service Name: TapiSrv
 Service Name: winmgmt
Process Name: spoolsv.exe ProcessID: 1640
 Service Name: Spooler
Process Name: explorer.exe ProcessID: 832
Process Name: wmiprvse.exe ProcessID: 632
Process Name: wmiprvse.exe ProcessID: 1436
Process Name: WINWORD.EXE ProcessID: 3764
Process Name: svchost.exe ProcessID: 2428
 Service Name: stisvc
Process Name: wmplayer.exe ProcessID: 172
Process Name: Dancer.exe ProcessID: 1180
Process Name: PrimalScript.exe ProcessID: 1920
Process Name: wmiprvse.exe ProcessID: 3156
Process Name: cscript.exe ProcessID: 3468
```

Working with Functions

You have already used functions that are predefined in VBScript. These form the crux of most of the flexibility of VBScript. VBScript has more than 100 intrinsic functions, which provide the ability to perform string manipulation (*mid*, *len*, *instr*, *instrrev*), work with arrays (*array*, *ubound*, *lbound*, *split*, *join*), and control the way numbers are handled (*int*, *round*, *formatnumber*). With so much functionality, you might wonder why you need to create your own functions. Although you can write much code that does not require functions, there may be times when functions will make it easier to reuse your code. In addition, functions often make it easier to read and understand what the script is doing. In the VideoAdapterRam_HardCoded.vbs script, video RAM is retrieved in bytes. To convert the number into something more manageable, we convert it into megabytes by dividing the number by 1,048,576. This causes a readability problem, as seen below.

VideoAdapterRAM_HardCoded.vbs

```
Option Explicit
'On Error Resume Next
Dim strComputer
Dim wmiNS
Dim wmiQuery
Dim objWMIService
Dim colItems
Dim objItem

strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select AdapterRAM from win32_videoController"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery)

For Each objItem in colItems
  WScript.Echo "AdapterRAM: " & objItem.AdapterRAM/1048576
Next
```

Add a function to convert to megabytes

1. Open the \My Documents\Microsoft Press\VBScriptSBS\ch15\VideoAdapter Ram_HardCoded.vbs script in Notepad or the script editor of your choice and save it as YourNameVideoAdapterRam_UseFunction.vbs.
2. At the bottom of the script, begin the function by using the command function call and name the function *convertToMeg*. Define a single input to the function as *intIn*. Close out the function by using the *End Function* command. This is seen below:

   ```
   Function convertToMeg(intIN)
   End Function
   ```

3. Between the *Function* and the *End Function* statements, divide *intIN* by 1,048,576 and assign the results to *convertToMeg*, as seen below:

   ```
   convertToMeg = intIN/1048576
   ```

4. In the *WScript.Echo* line in the main script, call the *convertToMeg* function and supply *objItem.AdapterRAM* as the input parameter to the function. The modified output line is seen below:

   ```
   WScript.Echo "AdapterRAM: " & convertToMeg(objItem.AdapterRAM)
   ```

5. Save and run the script. You should see the amount of video RAM reported in megabytes. If not, compare your script with \My Documents\Microsoft Press\VBScriptSBS \ch15\ VideoAdapterRam_UseFunction.vbs.

Comparing intrinsic function and user defined function

1. Open the \My Documents\Microsoft Press\VBScriptSBS\Templates\Blank Template.vbs script in Notepad or the script editor of your choice and save the script as YourNameFunString.vbs.
2. Add *Option Explicit* to the first noncommented line.
3. Use *WScript.Echo* to print out a line that calls the VBScript *String* intrinsic function. Tell the *String* function to repeat a dash character 15 times. This is seen below:

   ```
   WScript.Echo "Intrinsic string function",VbCrLf,string(15,"-")
   ```

4. At the bottom of the script, create a function called *funString*. Have the function accept two input parameters called *intIN* and *strChar*. Close out the function by using *End Function*. This is seen below:

   ```
   Function funString(intIN,strChar)

   End Function
   ```

5. Inside the *funString* function, declare a variable to be used as a counter. Call this variable *j*.

6. Use a *For Each...Next* loop to build up the string of repeated characters. Use *j* to count from one to *intIN*.

   ```
   For j = 1 To intIN

   Next
   ```

7. Between *For j = 1* to *intIN* and *Next*, add *funString* to itself and to *strChar*, as seen below:

   ```
   funString = funString & strChar
   ```

8. Copy the previous *WScript.Echo* line and replace the *String* function with the *funString* function, as seen below:

   ```
   WScript.Echo "User string function",VbCrLf,funString(15,"-")
   ```

9. Save and run your script. The results from the two functions should be exactly the same. If they are not, compare your script with \My Documents\Microsoft Press\VBScriptSBS \ch15\FunString.vbs.

Using ADSI and Subs, and Creating Users Step-by-Step Exercises

In this section, you will expand the script used in this chapter. Instead of creating only a user, you will add information to the user. You will use a subroutine to perform logging.

1. Open Notepad or your favorite script editor.
2. Open the \My Documents\Microsoft Press\VBScriptSBS\ch15\StepByStep\CreateUsers.vbs file and save it as YourNameCreateMultipleUsersSolution.vbs.
3. Make sure you have a file called C:\fso\UsersAndGroups.txt, and run the CreateUsers.vbs file. Go into Active Directory Users And Computers (ADUC) and delete the users that were created.
4. Cut the code used to open the text file that holds the names of users to add to Active Directory. It is under the variable declarations, in the Reference information section of the script. It is five lines long. This code looks like the following:

   ```
   TxtFile = "C:\fso\UsersAndGroups.txt"
   Const ForReading = 1
   Set objFSO = CreateObject("Scripting.FileSystemObject")
   Set objTextFile = objFSO.OpenTextFile _
     (TxtFile, ForReading)
   ```

5. Paste the code after the *WScript.Echo* command at the end of the script.

6. Under the declarations, where the *txtFile* code used to be, type **ReadUsers**. This is the name of the new subroutine you will create. It will look like the following:

```
Dim objOU
Dim objUser
Dim objGroup
Dim objFSO
Dim objTextFile
Dim TxtIn
Dim strNextLine
Dim i
Dim TxtFile
dim boundary

ReadUsers
```

7. On the line before the code that reads *TxtFile*, which you copied to the end of your script, use the *Sub* command to create a subroutine called *ReadUsers*.
8. At the end of the subroutine, add the *End Sub* command. The completed subroutine looks like the following:

```
Sub ReadUsers
  TxtFile = "c:\fso\UsersAndGroups.txt"
  Const ForReading = 1
  Set objFSO = CreateObject("Scripting.FileSystemObject")
  Set objTextFile = objFSO.OpenTextFile _
    (TxtFile, ForReading)
End Sub
```

9. Save your work. Run the script to make sure it still works. Open Active Directory Users And Computers and delete the users that were created by running the script.
10. Modify the subroutine so that it is reading a text file called MoreUsersAndGroups.txt. This file is located in the \My Documents\Microsoft Press\VBScriptSBS\ch15\Step ByStep folder.

Note If you do not add the full path to the MoreUsersAndGroups.txt file, you will need to ensure you are running the script from the same directory where the file is located.

11. In the Worker section of the script that creates the user, use the *Put* method to add the user's first name, last name, building, and phone number. The Active Directory attributes are called *givenName*, *sn*, *physicalDeliveryOfficeName*, and *telephoneNumber*. Each of these fields is in the array that gets created, so you need to increment the array field. The completed code will look like the following:

```
Set objUser = objOU.Create("User", "cn="& TxtIn(0))
objUser.Put "sAMAccountName", TxtIn(0)
objUser.Put "givenName", TxtIn(1)
objUser.Put "sn", TxtIn(2)
objUser.Put "physicalDeliveryOfficeName", TxtIn(3)
objUser.Put "telephoneNumber", TxtIn(4)
```

12. Because the group membership field is the last field and you added fields to the text file, you need to increment the array index that is used to point to the group field. The new index number is 5, and the code will look like the following:

```
Set objGroup = GetObject _
  ("LDAP://CN="& TxtIn(5) & ",cn=users,dc=nwtraders,dc=msft")
```

13. Save the script and run it. After you successfully run the script, delete the users created in Active Directory.

One Step Further: Adding a Logging Subroutine

In this section, you add logging capability to the script you finished in the Step-by-Step exercise.

1. Open Notepad or some other script editor.
2. Open \My Documents\Microsoft Press\VBScriptSBS\ch15\OneStepFurther\CreateMultipleUsersStarter.vbs and save the file as YourNameCreateMultipleUsersLogged.vbs.
3. After the *objGroup.add* command statement but before the *Loop* command, add a call to the subroutine called *LogAction*. The modification to the script will look like the following:

```
  Set objGroup = GetObject _
    ("LDAP://CN="& TxtIn(5) & ",cn=users,dc=nwtraders,dc=msft")
  objGroup.Add _
    "LDAP://cn="& TxtIn(0) & ",ou=Mred,dc=nwtraders,dc=msft"
  LogAction
Loop
```

4. Under the *ReadUsers* subroutine, add a subroutine called *LogAction*. This will consist of the *Sub* command and the *End Sub* command. Leave two blank lines between the two commands. The code will look like the following:

```
Sub LogAction


End Sub
```

5. Save your work.
6. Open the \My Documents\Microsoft Press\VBScriptSBS\ch15\OneStepFurther\CreateLogFile.vbs file and copy all the variable declarations. Paste them under the variables in your script.
7. Delete the extra *objFSO* variable.

8. Copy the three reference lines from the CreateLogFile.vbs script and paste them under the variable declarations. This section of the script now looks like the following:

```
Dim objOU
Dim objUser
Dim objGroup
Dim objFSO
Dim objTextFile
Dim TxtIn
Dim strNextLine
Dim i
Dim TxtFile
Dim objFile      'holds hook to the file to be used
Dim message      'holds message to be written to file
Dim objData1    'holds data from source used to write to file
Dim objData2    'holds data from source used to write to file
Dim LogFolder
Dim LogFile
message="Reading computer info " & Now
objData1 = objRecordSet.Fields("name")
objData2 = objRecordSet.Fields("distinguishedName")
```

9. Modify the message so that it states that the code is creating a user, and use the element *TxtIn(0)* as the user name that gets created. This modified line will look like the following:

```
message="Creating user " & TxtIn(0) & Now
```

10. Move the message line to the line after you parse *strNextLine*. You do this because you are using an element of the array that must be an assigned value before it can be used.

```
strNextLine = objTextFile.ReadLine
TxtIn = Split(strNextLine , ",")
message="Creating user " & TxtIn(1) & Now
```

11. Modify the objData1 and objData2 data assignments. Use *TxtIn(0)* for the user field and *TxtIn(5)* for the group. The two lines will look like the following:

```
objData1 = TxtIn(0)
objData2 = TxtIn(5)
```

12. Copy the remainder of the script and paste it between the two lines used to create the subroutine. The completed section looks like the following:

```
Sub LogAction
  If objFSO.FolderExists(LogFolder) Then
    If objFSO.FileExists(LogFile) Then
      Set objFile = objFSO.OpenTextFile(LogFile, ForAppending)
      objFile.WriteBlankLines(1)
      objFile.Writeline message
      objFile.Writeline objData1
      objFile.Writeline objData2
      objFile.Close
    Else
      Set objFile = objFSO.CreateTextFile(LogFile)
      objFile.Close
      Set objFile = objFSO.OpenTextFile(LogFile, ForWriting)
      objfile.WriteLine message
      objFile.WriteLine objData1
      objFile.WriteLine objData2
      objFile.Close
    End If
  Else
    Set objFolder = objFSO.CreateFolder(LogFolder)
    Set objFile = objFSO.CreateTextFile(LogFile)
    objFile.Close
    Set objFile = objFSO.OpenTextFile(LogFile, ForWriting)
    objfile.writeline message
    objFile.WriteLine objData1
    objFile.WriteLine objData2
    objFile.Close
  End If
End Sub
```

13. Save and run the script.

Chapter 15 Quick Reference

To	Do This
Create a subroutine	Begin a line with the word *Sub* followed by the name of the subroutine. You end the subroutine by using the command *End Sub* on a line following your subroutine code.
Call a subroutine	Place the name of the subroutine on a line by itself at the place in your code where you want to use the subroutine.
Make code more portable and easier to read and troubleshoot, and to promote code reuse	Use a subroutine.

Index