

Table of Contents

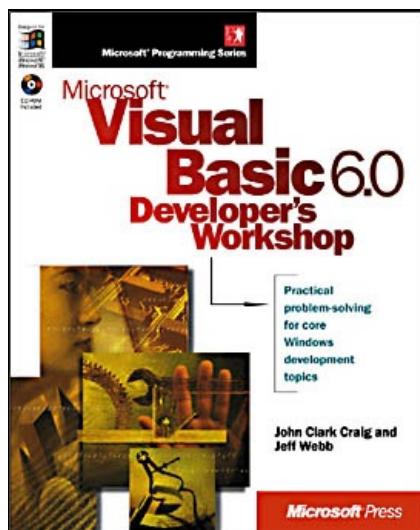
Cover	1
Dedications	2
Acknowledgments	3
About the Authors	4
Introduction	5
Programming Style	6
What's Been Left Out	7
How-To	8
Sample Programs	9
Windows 32-Bit Programming	10
Using the Companion CD-ROM	11
Part I: Getting Started With Visual Basic	12
Chapter One -- What's New in Visual Basic 6?	12
Edition Enhancements	13
Integrated Development Environment	14
Native Code Compiler	17
ActiveX	18
Even More New Internet Features	19
New and Enhanced Controls	20
Object-Oriented Features	22
Language Enhancements	23
Data Access	24
Internet	25
Chapter Two -- Programming Style Guidelines	26
Descriptive Naming	27
Checking Data Types	33
Scoping Things Out	34
Commenting While You Code	35
For More Information	36
Part II: Dear John, How Do I...	37
Chapter Three -- Variables	37
Simulate Unsigned Integers?	38
Work with True/False Data?	41
Use Byte Arrays?	42
Work with Dates and Times?	45
Work with Variants?	49
Work with Strings?	51
Work with Objects?	54
Work with Predefined Constants?	56
Create User-Defined Type (UDT) Structures?	59
Create New Data Types with Classes?	60
Chapter Four -- Parameters	62
Use Named Arguments?	63
Use Optional Parameters?	64
Pass Parameter Arrays?	65
Pass Any Type of Data in a Parameter?	66
Use Enums in Parameters?	67
Chapter Five -- Object-Oriented Programming	69
Choose Between an ActiveX EXE and an ActiveX DLL?	70
Create All My Objects in External ActiveX Components?	71

Create a New Object?	72
Use My New Object?	77
Set a Default Property for My Object?	80
Create and Use an ActiveX EXE?	81
Create an Object That Displays Forms?	84
Work with Collections of Objects?	86
Understand and Use Polymorphism?	94
Use Friend Methods?	95
Chapter Six -- ActiveX Controls	96
Create an ActiveX Control?	97
Debug a Control?	102
Compile and Register a Control?	104
Create a Design-Time Property?	105
Display a Property Pages Dialog Box?	108
Load a Property Asynchronously?	112
Create a Control for Use with a Database?	114
Use the DataRepeater Control?	117
Create a Container Control?	119
Chapter Seven -- Using Internet Components	121
Select the ActiveX Component to Use?	122
Understand Internet Protocol Layers?	123
Set Up Networking?	124
Communicate Using Winsock?	126
Create an FTP Browser?	132
Control Internet Explorer?	135
Chapter Eight -- Creating Internet Components	138
Create ActiveX Controls for Internet Use?	139
Use ActiveX Controls with VBScript?	140
Create DHTML Documents?	142
Create ActiveX Documents?	145
Create Webclasses?	147
Chapter Nine -- Creating Internet Applications	149
Choose an Application Type?	150
Create a DHTML Application?	151
Create IIS Applications?	155
Create ActiveX Document Applications?	161
Install ActiveX Documents over the Internet?	164
Install DHTML Applications over the Internet?	167
Deploy IIS Applications over the Internet?	168
Chapter Ten -- API Functions	169
Call API Functions?	170
Pass the Address of a Procedure to an API Function?	173
Understand ByVal, ByRef, and As Any in an API Function Declaration?	175
Easily Add API Declarations?	176
Use API Calls to Get System Information?	177
Add API Calls to an ActiveX Control?	182
Chapter Eleven -- Multimedia	186
Play a Sound (WAV) File?	187
Play a Video (AVI) File?	188
Play an Audio CD?	190
Chapter Twelve -- Dialog Boxes, Windows, and Other Forms	192
Add a Standard About Dialog Box?	193
Automatically Position a Form on the Screen?	197
Create a Floating Window?	199

Create a Splash (Logo) Screen?	201
Use a Tabbed Control?	203
Flash a Form to Get the User's Attention?	204
Move a Control to a New Container?	206
Chapter Thirteen -- The Visual Interface	207
Use the Lightweight Controls?	208
Add a Horizontal Scrollbar to a List Box?	209
Create a Toolbar?	211
Dynamically Change the Appearance of a Form?	213
Dynamically Customize the Menus?	214
Remove the Title Bar from a Form?	216
Create a Progress Indicator?	217
Use the Slider Control?	219
Use the UpDown Control?	220
Use the FlatScrollBar Controls?	222
Use the CoolBar Control?	224
Chapter Fourteen -- Graphics Techniques	225
Calculate a Color Constant from RGB, HSV, or HSL Values?	226
Convert Between Twips, Points, Pixels, Characters, Inches, Millimeters, and Centimeters?	234
Create One of Those Backgrounds That Fade from Blue to Black?	235
Create a Rubber Band Selection Rectangle?	236
Create Graphics Hot Spots?	239
Draw a Polygon Quickly?	241
Draw an Ellipse?	244
Fill an Irregularly Shaped Area with a Color?	247
Rotate a Bitmap?	250
Scroll a Graphics Image?	252
Use BitBlt to Create Animation?	254
Use Picture Objects for Animation?	257
Use the Animation Control?	259
Position Text at an Exact Location in a Picture Box?	260
Scale a Font Infinitely?	262
Rotate Text to Any Angle?	263
Use Multiple Fonts in a Picture Box?	267
Chapter Fifteen -- File I/O	269
Rename, Copy, or Delete a File Efficiently?	270
Work with Directories and Paths?	271
Perform Fast File I/O?	273
Work with Binary Files?	274
Use the Visual Basic File System Objects?	279
Chapter Sixteen -- The Registry	283
Read and Write to the Registry?	284
Remember the State of an Application?	287
Associate a File Type with an Application?	288
Chapter Seventeen -- User Assistance	291
Add ToolTips?	292
Add a Status Display to My Application?	293
Display a Tip of the Day at Startup?	294
Walk Users Through Tasks Using Wizards?	295
Create a WinHelp File?	297
Use the WinHelp API Function to Add Help Files to My Projects?	303
Add Context-Sensitive F1 Help to My Projects?	305
Use the CommonDialog Control to Add Help Files to My Projects?	306
Add WhatsThisHelp to a Form?	307

Create HTML Help?	308
Chapter Eighteen -- Security	312
Add a Hidden Credits Screen?	313
Create a Password Dialog Box?	315
Encrypt a Password or Other Text?	316
Work with Internet Security Features?	322
Chapter Nineteen -- The Mouse	323
Change the Mouse Pointer?	324
Create a Custom Mouse Pointer?	326
Display an Animated Mouse Pointer?	327
Determine Where the Mouse Pointer Is?	328
Chapter Twenty -- The Keyboard	329
Change the Behavior of the Enter Key?	330
Determine the State of the Shift Keys?	331
Create Hot Keys?	332
Chapter Twenty-One -- Text Box and Rich Text Box Tricks	333
Display a File?	334
Create a Simple Text Editor?	336
Detect Changed Text?	338
Fit More than 64 KB of Text into a Text Box?	339
Allow the User to Select a Font for a Text Box or a Rich Text Box?	341
Chapter Twenty-Two -- Multiple Document Interface	343
Create an MDI Application?	344
Add a Logo (Splash Screen) to an MDI Form?	346
Chapter Twenty-Three -- Database Access	347
Use Wizards in My Database Development Cycle?	348
Use the Data Control to Connect An Application to a Database?	349
Use Data Access Objects to Connect An Application to a Database?	352
Create a Report?	357
Chapter Twenty-Four -- ActiveX Objects in Other Applications	358
Use ActiveX to Perform Spell Checking?	359
Use ActiveX to Count Words?	364
Use Microsoft Excel's Advanced Math Functions?	365
Chapter Twenty-Five -- Screen Savers	367
Create a Screen Saver?	368
Prevent Two Instances of a Screen Saver from Running at the Same Time?	371
Hide the Mouse Pointer in a Screen Saver?	373
Detect Mouse Movement or a Mouse Click to Terminate a Screen Saver?	375
Detect a Keypress to Terminate a Screen Saver?	376
Use an Image of the Screen as a Screen Saver?	377
Add Password and Setup Capabilities to a Screen Saver?	381
Chapter Twenty-Six -- Project Development	382
Grab a Running Form and Save It as a Bitmap?	383
Use Resource Files?	385
Use a String Database for Internationalization?	388
Chapter Twenty-Seven -- Advanced Programming Techniques	391
Use Visual Basic to Create an ActiveX DLL?	392
Use C to Create a DLL?	396
Create an Application That Runs Remotely?	400
Create an Add-In for the Visual Basic Development Environment?	406
Make My Application Scriptable?	411
Pass a User-Defined Type to My Object?	419
Chapter Twenty-Eight -- Miscellaneous Techniques	421
Create a Linked List?	422

Respond to O/S Version Differences?	424
Exit and Restart Windows?	426
Dial a Phone from My Application?	427
Use Inline Error Trapping?	428
Part III: Sample Applications	430
Chapter Twenty-Nine -- Graphics	430
The HSVHSL Application	431
The Animate Application	445
The Lottery Application	453
The MySaver Application	460
Chapter Thirty -- Development Tools	472
The ColorBar Application	473
The APIAddin Application	476
The Metric Application	486
Chapter Thirty-One -- Date and Time	499
The VBCal Application	500
The VBClock Application	509
The NISTTime Application	521
Chapter Thirty-Two -- Databases	527
The AreaCode Application	528
The DataDump Application	535
The Jot Application	539
Chapter Thirty-Three -- Utilities	550
The MousePtr Application	551
The ShowTell Application	557
The WindChil Application	562
Chapter Thirty-Four -- Advanced Applications	569
The Messages Application	570
The Secret Application	578
The BitPack Application	590
The Dialogs Application	597
About This Electronic Book	608
About Microsoft Press	609



Copyright© 1998 by John Clark Craig and Jeff Webb.

To my folks way, way down south. Dad, who is also my friend and chat companion, and Mom, whose gentle spirit is still full of love in trying times.

—John Clark Craig

To my dharma-partner, Trish, who gave me two great kids, a chandelier, Stinky Bear, and lots of special treats.

—Jeff Webb

Acknowledgments

Microsoft Visual Basic 6.0 is another step forward in Microsoft's tradition of improving the Basic programming language. A few short years ago, the Basic programming language was widely considered, by those in the know—and by those who really didn't know—to be a "toy" language, suitable for quick-and-dirty, short programs of no great commercial or industrial significance. Well, don't let anyone tell you today's Visual Basic falls into this category!

Visual Basic 6.0, I'm very pleased to report, satisfies the needs, requirements, and goals of today's most exacting and demanding commercial and industrial applications requirements. It's a great tool for creating ActiveX components and full-blown, stand-alone 32-bit Windows applications. With its short learning curve, environment of proven, highly productive development, and ability to create fast and powerful applications and components, it's safe to say Visual Basic 6.0 simply does it all!

I want to thank all the great people behind the scenes at Microsoft Press for helping create this book. Several people played major roles and deserve special thanks. Marc Young's skill as Technical Editor was superb and precise. Pamela Hafey did an excellent job as Project Editor. Eric Stroo's insight and project guidance were, once again, right on track.

This is the second edition of the book with Jeff Webb as co-author. As before, it's been a real pleasure working with Jeff, and I know this book contains lots of improved content and coverage because of his talent, skills, and knowledge. Jeff's insight and ability to quickly understand and explain many of the complex new features of Visual Basic have made this book much better than it would have been with just one author.

Two readers deserve special credit, as they each helped immensely with improvements to specific programs in this edition. Dennis Borg provided source code for the common HSL (Hue, Saturation, Luminosity) color model conversion, and helped me debug my programs to get it right. Thanks, Dennis! Dylan Wallace spent many e-mail hours explaining to me how he figured out the preview mode enhancements, and a few other nice changes, for the screen saver programs. Thanks, Dylan! Both these guys are true experts, and it was fun to learn from them.

About the Authors

Jeff Webb

Jeff Webb is a former senior member of the Microsoft Visual Basic product team. While at Microsoft, he worked on all things Basic, including Basic PDS, QuickBasic, Visual Basic, Visual Basic for Applications, and OLE Automation. He lives in Sanibel, Florida, and when he's not fly fishing, he can be reached at JeffWebb@msn.com.

John Clark Craig

John Clark Craig is the author of 16 books on personal computing, including titles from Microsoft Press covering all versions of Microsoft Visual Basic for Windows, Microsoft Visual Basic for MS-DOS, Microsoft QuickBasic, and Microsoft QuickC. John was the technical editor for the first and second editions of *Microsoft Mouse Programmer's Reference* (Microsoft Press, 1989, 1991), and he authored parts of *Microsoft Word Developer's Kit* (Microsoft Press, 1993) and *Microsoft Windows 3.1 Developer's Workshop* (Microsoft Press, 1994). He lives with his family in Castle Rock, Colorado. His software and electronics design company is Craig Software, and he can be reached at JohnCraig@msn.com.

Introduction

In this edition of *Microsoft Visual Basic Developer's Workshop*, we've added a lot of new material covering the Internet, database programming, and Microsoft ActiveX development. We've also improved much of the material from the previous edition by incorporating suggestions from readers. As with the previous edition, you get two authors for the price of one—myself and Jeff Webb. Jeff is the author of several other fine books on topics related to Visual Basic. With two of us working on the book, we've been able to keep pace with the ever-changing world of Visual Basic development. Microsoft Visual Basic has become such a diverse and wide-ranging product that a team effort is much more effective in providing proper coverage and perspective. We've learned a lot in our research, and we hope you'll find many useful tidbits of Visual Basic programming information in these pages as a result!

Right up front, you'll find a concise description of the new features in Visual Basic 6. This will help you sift through all the hype to see how Visual Basic can help you become more efficient in the development of your projects. There have been many major changes to Visual Basic in this version, so be sure to at least skim Chapter 1, "[What's New in Visual Basic 6?](#)" to become familiar with the new capabilities.

This book refers throughout to Microsoft Windows 95 and Microsoft Windows NT, but also applies to Microsoft Windows 98. As a rule, any Visual Basic program written for Windows 95 will run on Windows 98 as well.

Programming Style

We've also included information about standard programming style, as gleaned from experts at Microsoft and elsewhere. These suggestions are not intended to be too rigorous—indeed, software development teams that are too strict tend to be uncreative, boring, and unproductive—but rather are meant to provide guidelines to help groups of developers more easily understand and share code. All of the source code presented in this book pretty much follows these guidelines.

What's Been Left Out

Introductory material on the history of Basic has been left out of this book, along with some beginner-level instructions for using Visual Basic. This leaves more room for the good stuff! There's plenty of introductory material for beginners in several popular introductory books on the Basic programming language. We've found that most readers are already somewhat familiar with Visual Basic and are looking for new information to improve their skills and to add to their personal toolboxes of techniques.

How-To

In our presentations on Visual Basic, we've noticed that the majority of questions asked by the audience are of a "How do I?Dear John, How Do I... ?" nature. Visual Basic programmers have discovered a wealth of powerful, but sometimes not too well documented, tricks and techniques for getting things done. We've added new, up-to-date material to Part II, the "Dear John, How Do I?Dear John, How Do I... ?" section of this book. Often Visual Basic 6 provides better solutions to these questions than did earlier versions of the language, so the information in Part II is more current than in many other books.

By the way, after much debate it was decided for the sake of simplicity to leave these questions in their "Dear JohnDear John, How Do I... " format, but please note that in many cases "Dear JeffDear John, How Do I... " would have been more accurate. "Dear J & JDear John, How Do I... , " "Dear Jeff and JohnDear John, How Do I... , " and "Hey, you guys!Dear John, How Do I... " were all ruled out, so "Dear JohnDear John, How Do I... " it remains. For the same reason, we use *I* instead of *we* in the text, even though there are two authors speaking.

Sample Programs

The CD-ROM that accompanies this book contains more—and more comprehensive—material than the previous edition. You'll now find that the CD-ROM contains the code from all parts of this book, including the code snippets in the "Dear John, How Do I?Dear John, How Do I... ?" section and the full-blown sample applications found in Part III. Visual Basic is a rich, diversified development environment, and we've tried to provide useful, enlightening examples that cover all the major subject areas. This does impose limits on the depth of coverage possible in a single book, however. For example, we've provided a real, working example of a complete but relatively simple database program, but there's no way that one book can cover all aspects of the rich set of database programming features now built into Visual Basic. This book provides a quick start in the fundamentals of database programming with Visual Basic, but you might want to supplement it with a book devoted entirely to the subject if this is your main area of interest.

Windows 32-Bit Programming

Visual Basic 4 was provided in both 16-bit and 32-bit flavors, designed to run on the Windows version 3.1 operating system and on Windows 95 and Windows NT. Visual Basic 6, however, does not provide a version for 16-bit programming. As a result, this book focuses primarily on Visual Basic 6 programming for 32-bit Windows.

If you are developing code for Windows NT, this book will still suit your needs well. Some application programming interface (API) system-level calls will need slight adjustments, but most of the programs and examples here will work just fine under Windows NT.

Using the Companion CD-ROM

Bound into the back of this book is a companion CD-ROM that contains all the source code, projects, forms, and associated files described in Part II and Part III of this book. This code is ready to be compiled into executable files, integrated into your own applications, or torn apart to see how it works. You can copy and paste the code or load the project files directly into your Visual Basic environment from the CD-ROM. If you would like to copy the project files from the CD-ROM to your hard drive, we've arranged them in a chapter-based directory structure to help you locate them. Also available on the CD-ROM is a complete online version of this book that you can view onscreen as you work through the exercises. It is a powerful, searchable HTML version that enables you to quickly locate specific information, such as a procedure or a definition, with just a click of the mouse. For additional information about using the companion CD-ROM, see the README.TXT file on the CD-ROM.

Chapter One

What's New in Visual Basic 6?

When you launch Visual Basic 6, you are greeted by the New Project dialog box, shown in Figure 1-1. As you can see, in addition to a "Standard EXE" application, Visual Basic lets you create a wide variety of component types, including Microsoft ActiveX dynamic link libraries (DLLs), ActiveX controls, and ActiveX documents for deployment on intranets or the Internet.

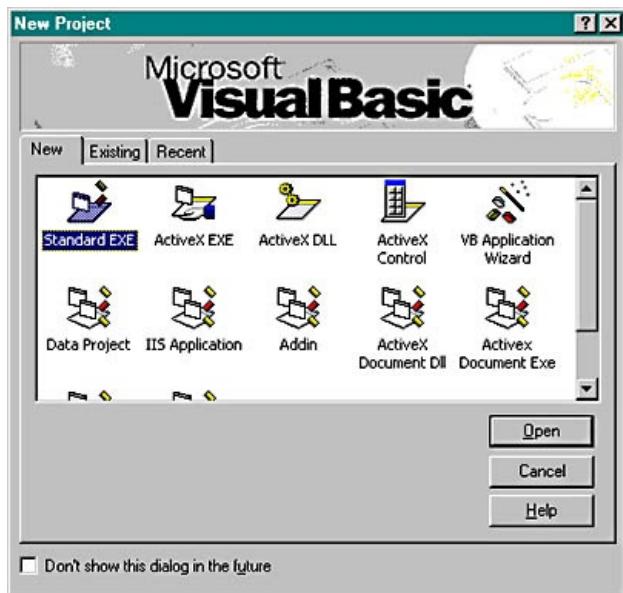


Figure 1-1. The Visual Basic 6 New Project dialog box.

In this chapter, I list many of the major new features and improvements found in Visual Basic 6, as well as some of the little enhancements that make Visual Basic a powerful and easy-to-use tool. Many of these new features are demonstrated throughout this book. Although I cover a lot of territory in this chapter and in the rest of this book, there may be a particular topic for which you require more detailed information. In this case, I recommend that you consult the Visual Basic documentation, Visual Basic's Books Online, and Microsoft's Web site (<http://microsoft.com>).

Edition Enhancements

Visual Basic now comes in three 32-bit flavors, or editions; each edition is an enhancement of its equivalent predecessor. All editions include Visual Basic Books Online, which is the complete documentation for Visual Basic in a CD-ROM multimedia format, based on Hypertext Markup Language (HTML), that is easy to navigate.

The Visual Basic Learning Edition allows programmers to easily create powerful 32-bit applications for Microsoft Windows 95, Microsoft Windows 98, and Microsoft Windows NT. It includes all the intrinsic controls, plus grid, tab, and data-bound controls.

The Professional Edition has all the features of the Learning Edition, plus additional ActiveX controls, including Internet and data access controls. Additional documentation includes *Component Tools Guide* and *Data Access Guide*.

The Enterprise Edition includes everything in the Professional Edition, plus the Automation Manager, the Component Manager, database management tools, the Microsoft Visual SourceSafe project-oriented version control system, and more.

Integrated Development Environment

Visual Basic's integrated development environment (IDE) has been getting better and better with each new version. I really like the option that lets you run the environment in Single Document Interface (SDI) mode or in Multiple Document Interface (MDI) mode and the capability to simultaneously load multiple projects into the environment as a single project group. SDI mode is what you're familiar with if you've used versions of Visual Basic prior to Visual Basic 5. In SDI mode, shown in Figure 1-2, the various windows open independently on your display, with the desktop or previously opened applications visible behind and between the Visual Basic windows. The MDI mode (now the default mode), shown in Figure 1-3, behaves more like Microsoft Word, Microsoft Excel, or Microsoft Access, with one large window for the overall application and multiple child windows displayed in and limited to the area of the main window. At first, especially if you're used to the SDI mode, this new MDI mode feels strange, but give it a try—you'll probably like it much better than the old SDI mode once you get used to it.

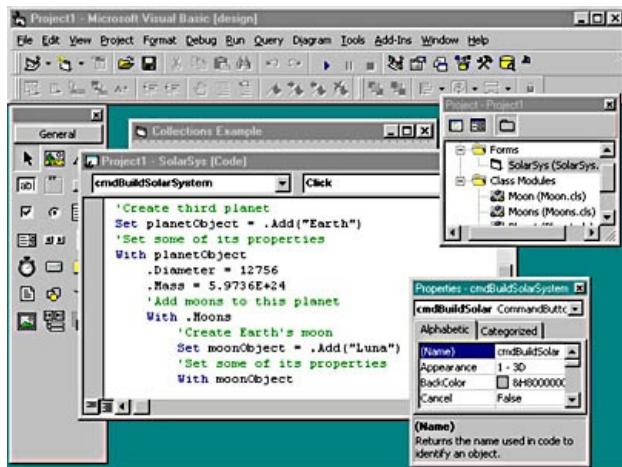


Figure 1-2. The integrated development environment in SDI mode.

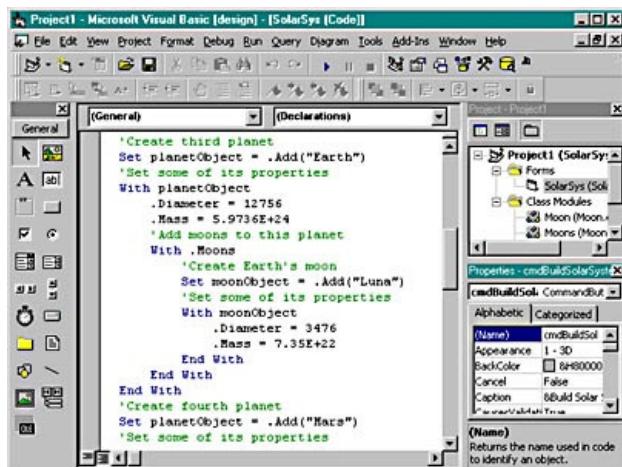


Figure 1-3. The integrated development environment in MDI mode.

The Object Browser is a powerful feature in the Visual Basic development environment. From the Object Browser, shown in Figure 1-4, you can jump quickly to modules and procedures in your project. With built-in search capabilities, a description pane, and other versatile features, the Object Browser makes it much easier to locate and understand all the component elements that are in your project or available to your project.

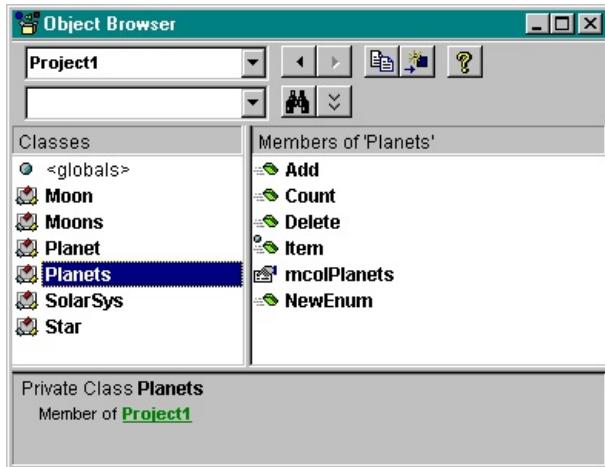


Figure 1-4. The enhanced Object Browser.

The Code Editor has some interesting features that are really slick. As you type, drop-down lists appear that help you complete the spelling of keywords, help you determine the properties or methods available for a control or an object, and provide other appropriate options. Figure 1-5 following shows an example of the Auto List Members feature. Another helpful feature is Auto Quick Info. The Auto Quick Info feature displays the syntax for statements and functions, as shown in Figure 1-6.

Numerous other recent enhancements include improved debugging windows, improved toolbars, dockable windows, an improved color palette, margin indicators that assist in code editing (such as creating breakpoints), a quick menu link to Microsoft's help information on the Internet, and the capability to add comment characters to and remove comment characters from all lines in a selected block of code. I find the block commenting capability especially useful during the development process. (Look on the Edit toolbar for the Comment Block and Uncomment Block buttons.)

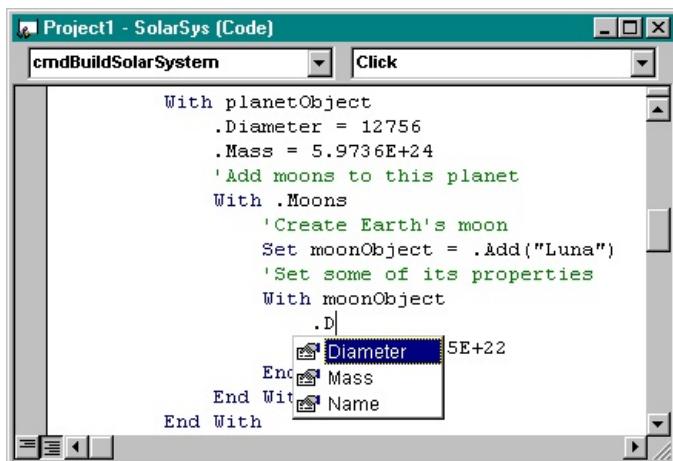


Figure 1-5. Example of the Auto List Members feature.

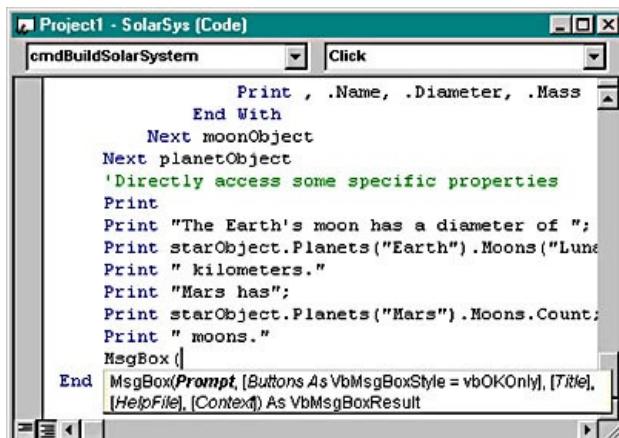


Figure 1-6. Example of the Auto Quick Info feature.

Visual Basic includes several handy utilities to assist you during the development process. For example, the Application Performance Explorer (APE), a software utility written in Visual Basic, is included in the Enterprise Edition of Visual Basic. The APE aids you in the design, deployment planning, and performance tuning of distributed client/server applications. The APE lets you easily run automated "what-if" tests to profile the performance of a multitier application in different network topologies, taking into consideration such factors as network bandwidth, request frequency, data transfer requirements, server capacity, and so on. All the source code for this application is provided with your Visual Basic installation for your study and modification.

Native Code Compiler

In my opinion, the native code compiler is one of the most important features of Visual Basic, partly because it helps Visual Basic gain respectability in the eyes of the many developers who've resisted trying Visual Basic simply because of its "toy language" image. Mostly, however, I like the native code compiler because it lets Visual Basic do new things that simply weren't appropriate in the past. For example, until recently, Visual Basic was a poor choice for tasks such as high-speed animated games, three-dimensional graphics transformations, ray-tracing algorithms, and so on. Figure 17 shows the compiler options on the Compile tab of the Project Properties dialog box, which is displayed when you choose Project Properties from the Project menu.

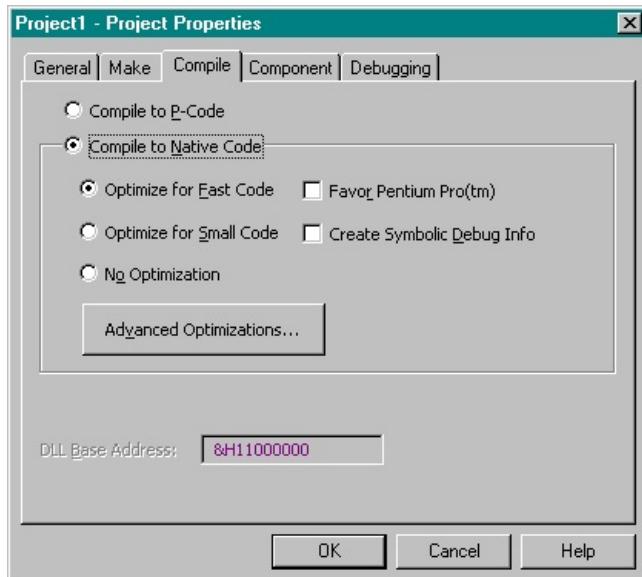


Figure 1-7. The Compile tab of the Project Properties dialog box.

High-speed number crunching, three-dimensional graphics creation, and similar computations will benefit the most from the compiler, but all Visual Basic applications will now run faster when compiled. Visual Basic uses the same compiler technology as found in Visual C++, and you can even debug your compiled code using the Visual C++ environment if you want.

ActiveX

ActiveX is a buzzword that refers to technologies that previously may have been associated with the term object linking and embedding (OLE). ActiveX is Microsoft's name for technologies that are based on the Component Object Model (COM). Visual Basic lets you create ActiveX controls, ActiveX documents, ActiveX DLLs, applications that expose ActiveX objects for other applications to use, and on and on. Even Visual Basic itself, and its IDE, are composed of ActiveX components. You'll hear a lot about ActiveX technology in this book and in just about everything else you read about Visual Basic. Think of ActiveX as the technology behind all the components you use and all the components you can create in Visual Basic.

Even More New Internet Features

It's obvious that the Internet and intranet technologies now constitute an extremely important part of Microsoft's vision and long-term goals. To jump-start this very important and significant use of Visual Basic for Internet and intranet applications, a special Control Creation Edition of Visual Basic became available as a free download from Microsoft's World Wide Web site before Microsoft released the full-blown version of Visual Basic 5. This limited edition of Visual Basic provides the capability to create ActiveX controls designed to work hand in hand with Microsoft Internet Explorer. Visual Basic 6 includes all the features of the special Control Creation Edition and much more. Visual Basic's Internet capabilities let you create powerful applications hosted by standard browsers, going far beyond the limitations of standard, relatively flat-looking HTML documents.

Visual Basic lets you build your own ActiveX documents. ActiveX documents are created in much the same way as any other Visual Basic form, but they are designed to be hosted by ActiveX document containers such as Internet Explorer. These documents support hyperlink navigation, menu negotiation with the hosting browser, help menu merging, and other powerful features. Perhaps the most significant feature is simply the capability to create complete, fully developed applications that run over the Internet or your intranet.

New and Enhanced Controls

Controls are an integral part of the Visual Basic experience. As with the earlier versions of Visual Basic, new and enhanced controls are an important incremental improvement to the language.

Generally speaking, the ActiveX controls in Visual Basic are one of its hottest concepts. You can combine one or more existing controls or develop your own controls from scratch to create ActiveX controls deployable locally or over the Internet. You'll continue to hear a lot about ActiveX controls and their use on the Internet over the next few years. There are several new ActiveX controls in the Visual Basic package. First, I'll list the ones common to all editions of Visual Basic 6:

- The new ActiveX Data Objects (ADO) Data control provides an enhanced way to connect any data source to any data consumer and requires minimal coding. This control also allows on-the-fly changing of the data sources.
- The DataGrid control, which is similar to the previous DBGrid control, is Unicode-enabled and permits easy viewing and editing of recordsets.
- The DataList and DataCombo controls—enhanced versions of the older DBList and DBCombo controls—allow dynamic switching of the data sources.
- The new Hierarchical FlexGrid control is an updated version of the FlexGrid control and permits display of hierarchical recordsets created from several different tables.
- The new ImageCombo control is similar to the previous ComboBox control, with the added ability to add images to the list of items.
- The ImageList control now supports graphics files in the GIF format.

Next I'll list several new controls and enhancements to previous controls available in the Professional and Enterprise Editions of Visual Basic 6:

- The new CoolBar control lets you create the stylish new user-configurable toolbars you see in the Visual Basic IDE.
- The DTPicker control provides a drop-down calendar for foolproof user selection and entry of dates and times.
- A completely new way of organizing and grouping the display of data from data sources is provided by the new DataRepeater control. This control hosts TextBox, CheckBox, and other bound controls to provide views of a database similar to Access forms.
- The FlatScrollBar control provides a new type of flat scrollbar for your applications.
- The ListView control has been enhanced to allow the addition of subitems in its new ListSubItems collection.
- The new MonthView control provides a one-month calendar sheet to let the user pick dates and select ranges of dates.
- The MSChart control has been enhanced to allow binding directly to a data source.
- The enhanced ProgressBar control now supports smooth scrolling, and vertical and horizontal placement.
- The Script control is a powerful new feature that allows your applications to give the end user a "script" capability. This lets users interactively create Visual Basic, Scripting Edition (VBScript), Microsoft JScript, or other scripting functions that you define.
- The Slider control has been enhanced to support ToolTips and buddy windows.

- The TabStrip control has been enhanced to support several new features, such as control placement, hot tracking, flat buttons, multiselect, and highlighting.
- The TreeView control enhancements now provide row selection, hover selection, and check boxes.

Visual Basic 6 continues the pattern of enhancements found in previous versions. For example, you'll find that almost all visual controls now have a property for displaying ToolTips. (See Figure 1-8.) Also, more and more controls are enhanced for common features such as data-access binding, many kinds of Internet capabilities, drag-and-drop functions, and so on.

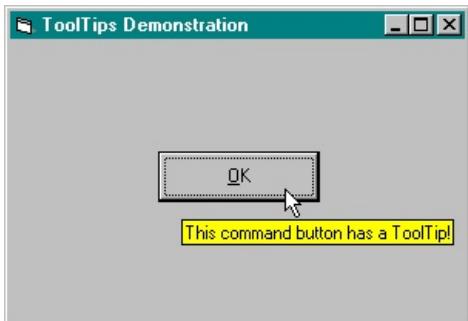


Figure 1-8. The *ToolTipText* property is now standard on most controls.

One of my favorite features is the capability of the controls containing graphics images to support the display of standard Internet graphics image types, such as JPEG and GIF, which are found all over the Internet, as shown in Figure 1-9.

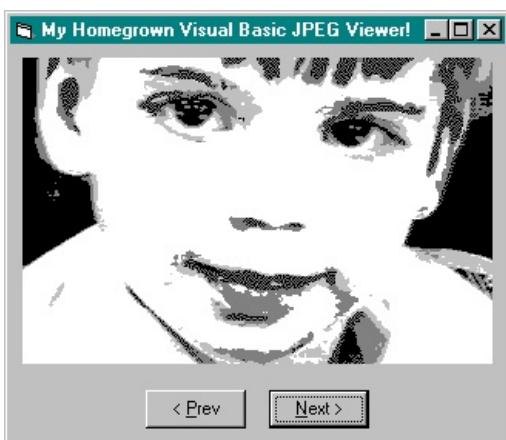


Figure 1-9. JPEG and GIF images are supported in controls that display graphics.

Be sure to carefully check through the lists of properties, methods, and events for the controls as you get ready to implement them—you'll probably discover many other exciting and powerful new features!

Object-Oriented Features

Class modules in Visual Basic 4 gave you the ability to create and use objects for the first time. Visual Basic 5 enhanced the features of class modules and added many new object-oriented programming (OOP) capabilities to the language. Surprisingly, Visual Basic 6 does not add a lot of new object-oriented features to the language. However, it's clear that the solid object-oriented foundation laid in previous versions of Visual Basic will play an increasing role in the use of Visual Basic 6 to create today's, and tomorrow's, advanced applications.

If you aren't yet familiar with these object-oriented programming features, now is a great time to jump on the bandwagon! Throughout this book I've attempted to provide working examples of object-oriented code and modules, in such a way that you aren't overwhelmed by this new way of organizing and structuring your applications. A whole new set of terminology comes with object-oriented programming. If the terms *polymorphism*, *friend functions*, *encapsulation*, *interfaces*, *collections*, and so on, are not all that clear to you, don't panic! I explain these concepts in more detail later and provide working examples to help you get a firm grasp on this important new part of the language.

Language Enhancements

The Visual Basic language has several new functions and enhancements that can greatly simplify some types of common programming tasks. For example, there are five new string functions that are a welcome addition to the bag of tricks available for working with strings. Each of these new functions is demonstrated throughout this book, but I'll list them here for quick reference: Filter, InStrRev, Join, Replace, and Split. Keep your eyes open in later chapters and you'll see how these little miracles can save you lots in your programming efforts!

In addition to the new string functions, there's one important new mathematical function I want to mention right up front. The Round function lets you round off a number to a specified number of digits. This could be accomplished before, using string conversions or tricky math, but the Round function simplifies the process and provides much faster operation. The Round function carries out the very common programming task of rounding money amounts to the nearest cent, as I've demonstrated in the Loan object in Chapter 5, "[Object-Oriented Programming](#)."

A whole range of clever new programming possibilities is opened up by the new CallByName function, which lets an application access a property or a method of an object by referring to it by its name as stored in a string.

The new FileSystem objects provide an entirely new way for your Visual Basic applications to interact with files. This object-oriented approach to traversing files and working with drives, folders, files, and all their properties, is a carry-over from the C++ world. It has some distinct advantages and provides easy solutions to some notably tricky but common programming tasks involving the file system.

There are several enhancements to Visual Basic that free you from some restrictions of earlier versions. User-defined types (UDTs) are now allowed as arguments and return types of public properties and methods. Functions and property procedures can now return entire arrays. Variable-sized arrays, also referred to as SAFEARRAYs, can now appear on the left side of an assignment.

There are quite a few other changes and enhancements in Visual Basic 6, such as many new functions and properties in Visual Basic for Applications. Throughout this book I've tried to use the very latest functions, features, and ways of doing things, wherever possible.

Data Access

The new ADO data access object model is simpler and more robust than earlier versions and is better integrated with other Microsoft data access tools and applications. The interface is the same now, whether you are accessing a local or a remote database, and hierarchical recordsets are supported.

In support of the new ADO object model, Visual Basic now ships with a new ADO Data control. This control allows you to create database applications with a minimum of coding and works hand in hand with the new ADO object model.

Data providers and data consumers can be linked in more ways than ever before. Practically any object can be set up for binding to any database. The Binding Manager facilitates binding between visual and nonvisual objects.

The Data Report Designer controls are a versatile set of tools that allow you to create and print banded reports efficiently. There are six controls in this set, and they are specially designed for calculating and formatting data for printed reports.

User controls can be created as data sources, allowing other controls to be bound to them.

The new Data View window lets you browse all the database connections currently available to your project.

A new Data Form Wizard is designed to automatically generate Visual Basic forms containing individual bound controls and procedures that are used to manage information derived from database tables and queries.

Several new and enhanced controls provide even better ways to view and edit recordsets from databases. These include the DataGrid, DataRepeater, and Hierarchical FlexGrid controls.

Internet

The Internet is playing an increasingly important role in today's applications development, and Visual Basic 6 provides many new and enhanced features to aid with the development of Internet enabled applications.

You can create Web page content using the new DHTMLPageDesigner object, which lets you drag and drop controls onto your page and write code to support them.

The Enterprise Edition of Visual Basic lets you create server-side applications that generate Dynamic HTML (DHTML).

The Package and Deployment Wizard (formerly known as the Setup Wizard) packages your application files for distribution on floppy disks, CD-ROMs, a local or network drive, or the Web.

Many Visual Basic 6 enhancements support the downloading, deployment, and capabilities of ActiveX documents using Internet Explorer 4 technology.

Chapter Two

Programming Style Guidelines

Visual Basic's ease of use can mask the need for a consistent programming style, but that need grows with the size of the programs you create. It's easy to debug a 100-line program, harder at 1000 lines, and almost impossible at 10,000 lines unless you've used some system of organization. The need for rules grows further when more than one person is writing the code or when the project needs to be maintained over long periods of time.

The goal of programming style is consistency and clarity. Good programming style will reduce the possibility of bugs and shorten the time it takes to read and understand code. Programming style guidelines must address naming, type checking, scope, and comments. Remember the following guidelines:

- Name items descriptively.
- Always use the Option Explicit statement.
- Use the most appropriate data type for each variable and parameter.
- Limit scope by using property procedures instead of global data.
- Write comments while you create code.

These rules bear repeating until they become so automatic that you don't have to think about them. This chapter expands on these guidelines and provides specific details.

Descriptive Naming

Is the Basic programming language easy to read or hard to read? It depends on who you talk to, what your background is, and how clean your programming habits are. In the old days, line numbers and GOTO statements made it easy to create obscure spaghetti code. Fortunately, today's versions of Basic are much more structured and object oriented, although you can still mangle the readability of your code if you really want to!

A lot of suggestions for standard coding style have been floated over the past few years, some good and some not so useful. In my opinion, it's not worth getting uptight about. If you will get into the habit of using a few relatively simple and standard techniques, you can gain a reputation for creating easy-to-read code. If you're part of a team programming effort, which often seems to be the case for Visual Basic programmers today, efforts at making code easier to read can greatly enhance productivity.

Control Prefixes

One of the easiest techniques to master is naming each control with a standard prefix that identifies references to the control in the source code. This really does improve the readability of your code. Consider, for example, an event-driven procedure named *Buffalo_Click*. Does this refer to a command button? A picture? An option button? Until you figure out just what object on the form the name refers to, you are pretty well buffalood. However, *cmdBuffalo_Click* is easily recognized as belonging to a command button, *picBuffalo_Click* belongs to a picture, and so on. A widely accepted list of standard prefixes has already been published in several places, and I'll repeat the list here with some additions to help propagate the standard.

Component-Naming Prefix Conventions for Visual Basic

Prefix	Component
<i>ado</i>	ADO Data
<i>ani</i>	Animation
<i>cal</i>	Calendar
<i>cbo</i>	Combo box
<i>ch</i>	Chart
<i>ch3</i>	3D check box
<i>chk</i>	Check box
<i>clp</i>	Picture clip
<i>cm3</i>	3D command button
<i>cmd</i>	Command button
<i>com</i>	Comm
<i>con</i>	Container (DAO)
<i>ctr</i>	Control (specific type unknown)
<i>dat</i>	Data
<i>db</i>	Database (DAO)
<i>dbc</i>	DataCombo
<i>dbd</i>	DataGrid
<i>dbe</i>	Database engine (DAO)
<i>dbgrd</i>	Data Bound Grid

<i>dbl</i>	ListBox
<i>dir</i>	Directory list box
<i>dlg</i>	Common dialog box
<i>doc</i>	Document (DAO)
<i>drv</i>	Drive list box
<i>drp</i>	DataRepeater
<i>dtp</i>	DTPicker
<i>fil</i>	File list box
<i>fld</i>	Field (DAO)
<i>flex</i>	Hierarchical FlexGrid
<i>fr3</i>	3D frame
<i>fra</i>	Frame
<i>frm</i>	Form
<i>fsb</i>	FlatScrollBar
<i>gau</i>	Gauge
<i>gpb</i>	Group push button
<i>gra</i>	Graph
<i>grd</i>	Grid
<i>grp</i>	Group (DAO)
<i>hdr</i>	Header
<i>hsb</i>	HScrollBar
<i>ils</i>	ImageList
<i>img</i>	Image
<i>imgcbo</i>	ImageCombo
<i>ix</i>	Index (DAO)
<i>key</i>	Key status
<i>lbl</i>	Label
<i>lin</i>	Line
<i>lst</i>	List box
<i>lsw</i>	ListView
<i>lwchk</i>	Lightweight check box
<i>lw.cbo</i>	Lightweight combo box
<i>lw.cmd</i>	Lightweight command button
<i>lw.fra</i>	Lightweight frame
<i>lw.hsb</i>	Lightweight horizontal scrollbar
<i>lw.lst</i>	Lightweight list box
<i>lw.opt</i>	Lightweight option button

<i>lwtxt</i>	Lightweight text box
<i>lwvsb</i>	Lightweight vertical scrollbar
<i>mci</i>	Multimedia MCI
<i>med</i>	Masked Edit
<i>mnu</i>	Menu
<i>mpm</i>	MAPIMessages
<i>mps</i>	MAPISession
<i>mst</i>	Tabbed Dialog
<i>mvw</i>	MonthView
<i>ole</i>	OLE
<i>op3</i>	3D option button
<i>opt</i>	Option button
<i>out</i>	Outline
<i>pic</i>	Picture box
<i>pnl</i>	3D panel
<i>prg</i>	ProgressBar
<i>prm</i>	Parameter (DAO)
<i>qry</i>	QueryDef (DAO)
<i>rd</i>	RemoteData
<i>rec</i>	RecordSet (DAO)
<i>rel</i>	Relation (DAO)
<i>rtf</i>	RichTextBox
<i>shp</i>	Shape
<i>sld</i>	Slider
<i>spn</i>	Spin button
<i>sta</i>	StatusBar
<i>ssys</i>	SysInfo
<i>tbd</i>	TableDef (DAO)
<i>tbs</i>	TabStrip
<i>tlb</i>	Toolbar
<i>tmr</i>	Timer
<i>tre</i>	TreeView
<i>txt</i>	Text box
<i>usr</i>	User (DAO)
<i>vsb</i>	VScrollBar
<i>wsp</i>	Workspace (DAO)

Variable Names

Some people, particularly those coming from the world of C programming, suggest naming all variables using Hungarian Notation prefixes, which are similar to the control prefixes listed above. I have mixed feelings about this technique. On the one hand, it's nice to know what type of variable you're dealing with as you read it in the code, but on the other hand, this can sometimes make for cluttered, less readable syntax.

Since day one, Basic programmers have had the option of naming variables using a suffix to identify the data type. For example, `X%` is an integer, `X!` is a single-precision floating-point number, and `X$` is a string. It's a matter of opinion whether these are better names than those using Hungarian Notation prefixes. I've even seen schemes for keeping track of the scope of variables using part of the prefix, which is an advantage of Hungarian Notation, but this much detail in the name of a variable might be overkill.

The following table lists the suffixes for various data types. It also lists the newer data types, which have no suffix, supporting my contention that Microsoft is heading away from the use of these suffixes.

Standard Suffixes for Data Types

Suffix	Data Type
<code>%</code>	2-byte signed integer
<code>&</code>	4-byte signed integer
<code>@</code>	8-byte currency
<code>!</code>	4-byte single-precision floating-point
<code>#</code>	8-byte double-precision floating-point
<code>\$</code>	String
None	Boolean
None	Byte
None	Collection
None	Date
None	Decimal (a 12-byte Variant)
None	Object
None	Variant

You might think that Hungarian Notation prefixes for all variable names are an absolute must, so I'll list the suggested prefixes here. Notice that some data types are indicated by more than one prefix—this lets you keep track of the intended use of the variable. For example, the standard 16-bit signed integer data type can be called Boolean, Handle, Index, Integer, or Word, depending on how you want to look at a given variable.

Hungarian Notation Prefixes for Data Types

Prefix	Data Type
<code>b/ln</code>	Boolean
<code>byt</code>	Byte
<code>cur</code>	Currency
<code>dbl</code>	Double
<code>dec</code>	Decimal
<code>d/tm</code>	Date (Time)

sng	Float/Single
h	Handle
i	Index
lng	Long
int	Integer
obj	Object (generic)
str	String
u	Unsigned quantity
ulng	Unsigned Long
vnt	Variant
wrd	Word

The Object data type refers only to generic objects, not to specific objects that you know the name of—for example, a procedure might receive an object as a parameter but might not know what type of object it is. When you do know an object's type, use a prefix based on the object's class name. (See the section "[Class Names](#)" below for more information.)

Variable Declarations

As mentioned at the beginning of this chapter, one very important technique for keeping track of variable names is the use of the Option Explicit statement. You can choose to automatically add the Option Explicit statement whenever a new module is created, and I'd strongly recommend using this option. From the Tools menu, choose Options, select the Editor tab, and check the Require Variable Declaration check box.

The Option Explicit statement forces you to declare all variables before they are referenced. The Dim statement for each variable is an excellent place to explicitly declare the data type of the variable. This way, a variable name can be easy to read and easy to differentiate from the names of controls and other objects that do have prefixes, and its type can be easily determined by a quick glance at the block of Dim statements at the head of each module or procedure.

WARNING

A very common and dangerous mistake, especially for those familiar with the C programming language, is to incorrectly combine the declarations of several variables of one data type into one statement. This won't cut the mustard! You must explicitly define the data type for each variable in a Dim statement. Read on!

The following explicit variable declarations are all OK, and you'll get what you expect:

```
Dim intA As Integer, sngB As Single, dblC As Double
Dim D%, E!, F#
```

However, the following declaration will create two Variant variables (*intI* and *intJ*) and one Integer variable (*intK*) instead of the expected three Integer variables:

```
Dim intI, intJ, intK As Integer
```

Your program might or might not work as expected, and the reason for any unexpected behavior can be very hard to track down.

Menus

There are several schemes for naming menu items. I've chosen to name all menu items with the now-standard *mnu* prefix. You might want to add letters to identify which drop-down menu a given menu item is associated with. For example, a menu item named *mnuHelpAbout* refers to the About item selected from the Help menu. I've chosen to keep things slightly simpler—I usually shorten the name to something like *mnuAbout*. When you see *mnu* as a prefix in the source code in this book, you'll know immediately that it's a menu item. It's usually quite simple to figure out where in the menu structure the item is positioned, so I don't confuse the issue with extra designators.

Class Names

When creating new classes, forgo a prefix and name the class descriptively; when declaring object variables, use a prefix that designates its class. Typically, a unique three-character prefix is used for object variables, but if necessary, use a longer prefix to be more descriptive. In other words, create your own object-naming scheme. Microsoft suggests using whole words and standard capitalization when naming classes, as shown in the following table.

Examples of Class-Naming and Object Variable-Naming Conventions

Class Name	Object Variable Name Prefix	Sample Object Variable Name
Loan	lon	lonBoat
Planet	plnt	plntEarth
EmployeeRecord	erc	ercSales
DailySpecial	dspec	dspecTuesday

Checking Data Types

You should strive to use the data type that is most appropriate for each variable. This is a little different from using the "smallest possible" data type. For example, the Boolean data type should consume only 1 bit—*True* or *False*, right? But a Boolean variable is actually the same size as an Integer variable, 2 bytes.

So why carry around extra baggage? Because these days it is better to be safe than small. Even if you might be developing code that will be used on the Internet, where size is an important consideration, it is better to use appropriate data types to make your code safer and more understandable. A Boolean variable will always contain one of two values: *True* or *False*. The smallest type, Byte, also always evaluates to *True* or *False*. The difference is that a Byte variable can contain any value from 0 to 255, so there's no guarantee that its value represents only *True* or *False*. Using a Boolean variable to represent *True* or *False* values is a clearer and therefore safer way to program.

Using the most appropriate data type is especially important when you declare object variables. Declaring an object variable as a specific class name rather than using the generic Object data type allows the compiler to optimize the use of that object. (The specifics of this are a little too complicated to discuss here; for more information, see Chapter 24, "[ActiveX Objects in Other Applications](#)."

Always specify data types for each parameter in a procedure. This helps detect the inappropriate use of a procedure. Remember, optional parameters can now have data types other than Variant. This feature was missing in Visual Basic 4, so it's a good idea to go back over procedures you created with that version to see whether you can tighten things up a bit.

Scoping Things Out

Scope refers to the visibility of a variable, a procedure, or an object to other procedures in a program. In the old days, all data was global—that is, you could get or change the value of any variable anywhere in a program. Later, when subroutines and function procedures were added to Basic, data could be global or local to a procedure. Today there are four levels of scope, listed here:

- **Universal** Visible to other running applications through Microsoft ActiveX
- **Global** Visible to all procedures in all modules of a project
- **Module** Visible to all procedures in the same module
- **Local** Visible only to the procedure that contains the variable

Limiting the scope of an item provides control over how the item can be changed. The wider the scope, the more careful you have to be. It makes sense to limit the scope of items as much as possible. In fact, global variables can be eliminated entirely using property procedures. These procedures provide an additional level of control over global data by checking whether a value is valid and whether the calling procedure has permission to change the data.

Commenting While You Code

Good comments are a pleasant surprise for anyone who's ever tried to revise someone else's program. Not too long ago, I landed a contract writing a guide for a new, company-internal programming language. Was there a spec? No. Were there sample programs? Not really. Comments in the source code? No—strike three. The lead developer and I spent a lot more time with each other than with our families for a number of weeks—good thing we got along. But why take that chance? Better to write your comments as you go. Here are some ways to structure comments within a procedure:

- In procedure headers, define what the procedure does and list the other procedures or global data the procedure uses.
- Before Case statements and other decision structures, briefly summarize the choices and the possible actions taken.
- Before loops, describe the processing that is performed and the exit conditions.
- Warn about assumptions by using the word **ASSUMPTION:** followed by a description of what is assumed. Use **UNDONE** to mark anything you need to get back to, and then use the Find command to return to these lines and resolve problems before releasing your code.

For More Information

You can find more information about standard naming schemes and programming style in the Visual Basic online help under the topic Visual Basic Coding Conventions. If you're part of a large programming team and this is an important subject for you, I'd suggest this topic as a good reference for a standard set of techniques. Many corporations and programmers use this same set of standards, so there's no sense in setting out alone on a dead-end road with your own "standards."

Chapter Three

Variables

Visual Basic's new string functions reduce the need for the bag of string-handling routines most of us have been carrying around since BASICA. Common string tasks such as removing spaces, replacing tabs, and breaking strings into tokens are now handled by the Visual Basic language.

Visual Basic 6 also provides controls for selecting and displaying dates in calendar and list box formats. The Calendar and DTPicker controls make conveying date information to users easier and saves you from a lot of handwork creating attractive ways to display dates.

This chapter covers the uses of these new features and discusses some tricks for working with variables that are not new to this version of Visual Basic, such as using Variants and simulating unsigned integers when working with the Windows application programming interface (API). Also covered in this chapter are working with predefined constants and creating user-defined type (UDT) structures.

Dear John, How Do I... Simulate Unsigned Integers?

Unfortunately, Visual Basic does not support unsigned 16-bit integers, a data type often encountered in API calls. But don't despair—there are ways to compensate for this lack.

Visual Basic's integer data types come in three distinct flavors: Long, Integer, and Byte. Long variables are 32-bit signed values in the range -2,147,483,648 through +2,147,483,647. The most frequently used of these three data types is Integer, which stores 16-bit signed integer values in the range -32,768 through +32,767. Byte variables hold unsigned 8-bit numeric values in the range 0 through 255. Notice that the only unsigned integer data type is Byte.

Unsigned 16-bit integers are useful in many API function calls. You can go ahead and pass signed integer variables to and from these functions instead, but you must develop a mechanism to deal with negative values when they show up. There are several approaches to simulating unsigned 16-bit integers, and I'll cover two of the best.

NOTE

If you are compiling your program using the native code compiler, be aware that you can turn off integer overflow detection. This allows incorrect integer computation in some cases, so you should carefully consider your application before turning off this option. On the positive side, turning off integer overflow detection allows faster integer math calculations and, for certain types of calculations, effectively allows unsigned integer results.

Transferring to and from Long Variables

In some cases, you can manipulate unsigned integers in the range 0 through 65,535 by storing them in Long integers. When it comes time to assign the 16bit values to a signed Integer variable that will simulate an unsigned 16-bit variable, use the following calculation:

```
intShort = (lngLong And &H7FFF&) - (lngLong And &H8000&)
```

Here *intShort* is a signed Integer variable to be passed to the API, and *lngLong* is the Long variable containing the stored value to be passed. The calculation uses the logical And operator and hexadecimal-based bit masks to execute bitwise operations to convert the 32-bit value to a 16-bit unsigned value.

To store the value of a signed integer that has been used to simulate an unsigned 16-bit integer as a Long integer (the inverse of the calculation just shown), use this calculation:

```
lngLong = intShort And &HFFFF&
```

Be aware that the 16-bit unsigned value (*intShort*), although stored in a 16-bit signed Integer variable, might be interpreted as a negative value if you print or calculate with it. Use these two calculations to convert the values just before and just after calling API functions that expect a 16-bit unsigned integer. Work with the Long integer version (*lngLong*) of the unsigned integers in your code, and pass the Integer version (*intShort*) of the value to API functions.

Packing Unsigned Byte Values Using Data Structures

Visual Basic doesn't have a Union construct as C does, but you can simulate the construct's functionality by copying bytes between UDT structures using the LSet statement. This makes it easy to pack and unpack unsigned Byte values into and out of a signed integer. The following code fragments demonstrate this technique:

```
Option Explicit
```

```
Private Type UnsignedIntType
    lo As Byte
    hi As Byte
End Type
```

```
Private Type SignedIntType
    n As Integer
End Type
```

These two UDT structures define storage for 2 bytes each. Although the memory allocation is not overlapping, as would be true in a C union, the binary contents can be shuffled between variables of these types, as the next block of code demonstrates:

```
Private Sub Form_Click()
    'Create variables of user-defined types
    Dim intU As UnsignedIntType
    Dim intS As SignedIntType
    'Assign high and low bytes to create integer
    intU.hi = 231
    intU.lo = 123
    'Copy binary data into the other structure
    LSet intS = intU
    Print intS.n, intU.hi; intU.lo
    'Assign integer and extract high and low bytes
    intS.n = intS.n - 1  'Decrement integer for new value
    'Copy back into the other data structure
    LSet intU = intS
    Print intS.n, intU.hi; intU.lo
End Sub
```

If you put these two code fragments in a form and click on it, the signed integer value (-6277) and the two byte values (231 and 123) on which it is based are printed on the form. After conducting an operation on the integer—in this case, subtracting 1—the result of the reverse operation is also printed: the new signed integer value (-6278) and the high and low bytes (231 and 122) of the integer. This is accomplished by first declaring two variables, *intU* and *intS*, using the declared UDT structure definitions. The *intU.hi* and *intU.lo* bytes are used to assign values to the high and low bytes of the *intU* variable. The binary contents of *intU* are then copied into *intS* using LSet, and the resulting signed integer is printed out. Finally, the integer value in *intS* is decremented to provide a value different from the original, and *intS* is copied back into *intU* to show how a signed integer can be split into two unsigned byte values.

By adding text boxes and labels to your form, it's easy to transform the code fragments into a working integer-byte calculator, as shown in Figure 3-1.

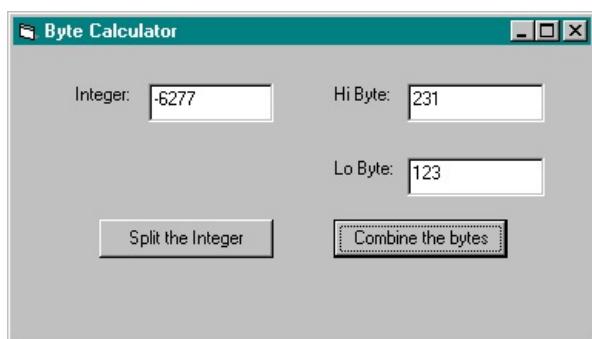


Figure 3-1. A signed integer comprising two unsigned bytes.

Note that LSet can be used to copy any binary contents of one UDT structure to another. This provides a simple way to treat the binary contents of memory as different types of data.

WARNING

In Microsoft Windows 95 and Microsoft Windows NT, the elements in your UDT structures might not line up in memory exactly as you'd expect. This is because each element is aligned on a 4-byte boundary, with extra padding bytes inserted to accomplish the

alignment. When using LSet to move data from one UDT structure to another, experiment to verify that the bytes end up where you want them to go. Be careful!

SEE ALSO

- "Dear John, How Do I... Use C to Create a DLL?" in Chapter 27, "[Advanced Programming Techniques](#)," for information about creating a DLL using Microsoft Visual C++. The C language is ideally suited to the packing and unpacking of bytes within integers.

Dear John, How Do I... Work with *True/False* Data?

The Boolean data type stores either the value *True* or the value *False* and nothing else. Typically, Boolean variables are used to store the results of comparisons or other logical tests. For example, the following procedure assigns the result of a logical comparison of two values to a Boolean variable named *blnTestResult* and then prints this variable to the display:

```
Private Sub Form_Click()
    Dim blnTestResult As Boolean
    blnTestResult = 123 < 246
    Print blnTestResult
End Sub
```

Notice that the displayed result appears as *True* in this case. When you print or display a Boolean variable, you print or display either *True* or *False*.

WARNING

Watch out for strange variable type coercions! Read onDear John, How Do I...

Visual Basic supports many types of automatic variable type coercions, which means you can assign just about anything to just about anything. For example, even though Boolean variables hold only the values *True* and *False*, you can assign a number or even a string to a Boolean variable. The results might not be what you'd expect, though, so be careful!

Here's one short example of this rather strange behavior. The following code will display *True*, *True*, and *False*, which implies that *bytA* equals *True* and *intB* equals *True* but that *bytA* does not equal *intB*! The reason for the final result of *False* might not be obvious; it results from the internal conversions of the unlike variables *bytA* and *intB* to like data types during the test to see whether they equal each other.

```
Private Sub Form_Click()
    Dim bytA As Byte
    Dim intB As Integer
    bytA = True
    intB = True
    Print bytA = True
    Print intB = True
    Print bytA = intB
End Sub
```

Use the operators *And*, *Or*, and *Not* to perform logical operations with Booleans. Logical operations combine several conditions in a single statement, creating a result that is either *True* or *False*. For example:

```
If blnExit And Not blnChanged Then End
```

This line of code ends a program if *blnExit* is *True* and *blnChanged* is *False*. You can use mathematical operators for the same task, but the logical operators make the code shorter and more readable. By using Boolean variables exclusively in these tests, you also avoid unexpected type coercion.

Dear John, How Do I... Use Byte Arrays?

One of the main reasons Byte arrays were created was to allow the passing of binary buffers to and from the 32-bit API functions. One of the differences to be aware of between 16-bit and 32-bit Visual Basic applications is that 32-bit version strings are assumed to contain Unicode characters, which require 2 bytes for each character. The system automatically converts 2-byte Unicode sequences to 1-byte ANSI characters, but if the string contains binary data the contents can become unintelligible. To prevent problems, you should get into the habit of passing only character data in strings and passing binary data in Byte arrays.

Passing Byte Arrays Instead of Strings

Byte arrays contain only unsigned Byte values in the range 0 through 255. Unlike the contents of strings, Byte array contents are guaranteed not to be preprocessed by the system. You can pass Byte arrays in place of strings in many API functions.

For example, the following code, which uses the GetWindowsDirectory Windows API function to find the path to the Windows directory, demonstrates the changes you'll need to make to the function declarations and function parameters. The code is shown in two versions: the first passes a string in which the API function can return the Windows directory, and the second passes a Byte array instead. When you place either of these code examples in a form, run it, and click on the form, you'll see the path to your Windows directory, as shown in Figure 3-2.

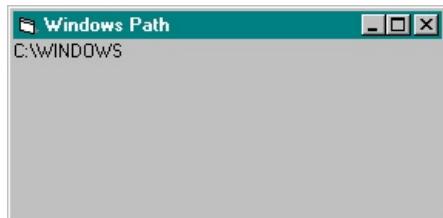


Figure 3-2. The API function that returns the path to your Windows directory.

By carefully noting the differences in these two examples, you'll gain insight into the differences between string and Byte array parameters. The string example is shown here:

```
Option Explicit
```

```
Private Declare Function GetWindowsDirectory _  
Lib "kernel32"  
Alias "GetWindowsDirectoryA" ( _  
    ByVal lpBuffer As String, _  
    ByVal nSize As Long _  
) As Long

Private Sub Form_Click()  
    Dim intN As Integer  
    Dim strA As String  
    'Size the string variable  
    strA = Space$(256)  
    intN = GetWindowsDirectory(strA, 256)  
    'Strip off extra characters  
    strA = Left$(strA, intN)  
    Print strA  
End Sub
```

In this first code example, the string parameter *lpBuffer* returns the path to the Windows directory. I presized the variable *strA* to 256 characters before calling the function, and I stripped off the extra characters upon returning from the function.

WARNING

Before making the function call, always presize a string or a Byte array that an API function fills with data. Your program will probably crash if you forget to do this!

Here's the Byte array example:

```
Option Explicit

Private Declare Function GetWindowsDirectory _
Lib "kernel32" _
Alias "GetWindowsDirectoryA" ( _
    ByRef lpBuffer As Byte, _
    ByVal nSize As Long _
) As Long

Private Sub Form_Click()
    Dim intN As Integer
    Dim bytBuffer() As Byte
    Dim strA As String
    `Size the Byte array
    bytBuffer = Space$(256)
    intN = GetWindowsDirectory(bytBuffer(0), 256)
    strA = StrConv(bytBuffer, vbUnicode)
    `Strip off extra characters
    strA = Left$(strA, intN)
    Print strA
End Sub
```

Take a close look at the function declaration for the GetWindowsDirectory API function in this second example. I changed the declaration for the first parameter from a ByVal string to a ByRef Byte array. The original parameter declaration was:

```
ByVal lpBuffer As String
```

I changed the ByVal keyword to ByRef and changed String to Byte in the new form of this declaration:

```
ByRef lpBuffer As Byte
```

A ByVal modifier is required for passing string buffers to API functions because the string variable actually identifies the place in memory where the address of the string contents is stored—in C terminology, a pointer to a pointer. ByVal causes the contents of the memory identified by the string name to be passed, which means that the value passed is a memory address to the actual string contents. Notice that in my function call I pass *bytBuffer(0)*, which, when passed using ByRef, is passed as the memory address for the contents of the first byte of the array. The result is the same. In case of either byte array or string, the GetWindowsDirectory API function will blindly write the Windows directory path into the addressed buffer memory.

Further along in the code is this important line which requires some explanation:

```
strA = StrConv(bytBuffer, vbUnicode)
```

This command converts the Byte array's binary data to a valid Visual Basic string. Dynamic Byte arrays allow direct assignment to and from strings, which is accomplished like this:

```
bytBuffer = strA
strB = bytBuffer
```

When you assign a string to a dynamic Byte array, the number of bytes in the array will be twice the number of characters in the string. This is because Visual Basic strings use Unicode, and each Unicode character is actually 2 bytes in size. When ASCII characters are converted to a Byte array, every other byte in the array will be a 0. (The second byte is used to support other character sets, such as for European or Asian languages, and becomes important when internationalization of your applications is necessary.) In the case of the GetWindowsDirectory API function, however, the returned buffer is not in

Unicode format, and we must convert the Byte buffer's binary contents to a Unicode string ourselves, using the `StrConv` function as shown previously. In the first code example, we let Visual Basic perform this housekeeping chore automatically as it filled our string parameter.

The conversion to Unicode converts each character in the buffer to 2 bytes and actually doubles the number of bytes stored in the resulting string. This isn't readily apparent when you consider that the function `Len(strA)` reports the same size as would `UBound(bytBuffer)` after the Unicode conversion in the above case. However, the function `LenB(strA)` does report a number twice the size of the number reported by `UBound(bytBuffer)`. This is because the `Len` function returns the number of characters in the string, whereas the `LenB` function returns the number of bytes in the string. The character length of a Unicode string (remember, this includes all 32-bit Visual Basic strings) is only half the number of actual bytes in the string because each Unicode character is 2 bytes.

To summarize, when converting an API function parameter from a string buffer to the Byte array type, change the `ByVal` keyword to `ByRef`, pass the first byte of the array instead of the string's name, and if the binary data is to be converted to a string, remember to use `StrConv` with the `vbUnicode` constant.

Copying Between Byte Arrays and Strings

To simplify the transfer of data between Byte arrays and strings, the designers of Visual Basic decided to allow a special case assignment between any dynamic Byte array and any string.

NOTE

You can assign a string to a Byte array only if the array is dynamic, not if it is fixed in size.

The easiest way to declare a dynamic Byte array is to use empty parentheses in the `Dim` statement, like this:

```
Dim bytBuffer() As Byte
```

The following `Dim` statement creates a fixed-size Byte array, which is useful for a lot of things but not for string assignment. It will, in fact, generate an error if you try to assign a string to it.

```
Dim bytBuffer(80) As Byte
```

Dear John, How Do I... Work with Dates and Times?

Internally, a Date variable is allocated 8 bytes of memory that contain packed bit patterns for not only a date but also an exact time. Magically, when you print a Date variable, you'll see a string designation for the year, month, day, hour, minute, and second that the 8-byte internal data represents. The exact format for the displayed or printed date and time is dependent on your system's regional settings. For all the sample calculations that follow, I've assumed that date and time values are always stored in Date variables.

Using the Date Controls

The DTPicker and Calendar controls provide convenient ways to get and present date information. The DTPicker control is part of the Microsoft Windows Common Controls part 2 (MSCOMCT2.OCX). The Microsoft Calendar control is in its own file (MSCAL.OCX).

Use the DTPicker control to get or display date information in a list box. Clicking the DTPicker control displays a small calendar from which you can choose a date, as shown in Figure 3-3.



Figure 3-3. The DTPicker also allows users to type dates directly in the text box portion of the control.

Use the Calendar control to get or display date information as a calendar page. The Calendar control typically takes up more space than the DTPicker and provides you with more display options, as shown in Figure 3-4.

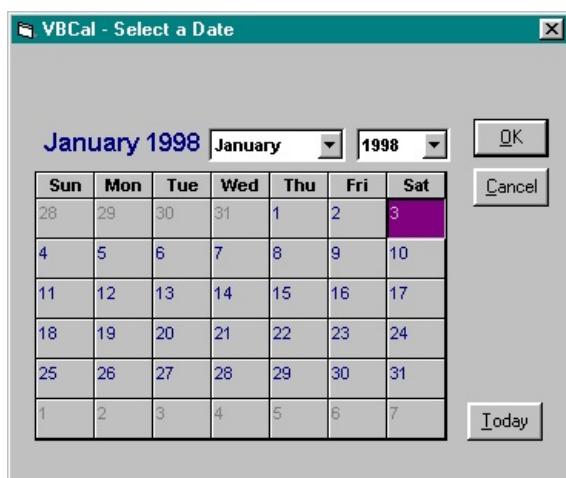


Figure 3-4. The Calendar control lets you choose different fonts, selected date highlighting, and other display options.

By default, both date controls display the current date when initialized. To change the displayed date in code, simply assign a date to the date controls' Value property. Be careful to strip out time information before assigning a date to the Calendar control, however. Including time information will cause an error. For example, the following two lines show the right way and the wrong way to reset the Calendar control to the current date:

```
calDate.Value = Date      'The right way
calDate.Value = Now       'The wrong way; causes an error
```

This problem does not occur with the DTPicker control, a fact that can cause some subtle problems when you use the controls together. The following four lines show how time information retained in a DTPicker control named *dtpDate* can cause an error when passed to a Calendar control named *calDate*:

```
dtpDate.Value = Date  
calDate.Value =.dtpDate.Value `This line runs fine,  
dtpDate.Value = Now  
calDate.Value =.dtpDate.Value `but causes an error here!
```

Loading a Date Variable

To load date and time values directly into a variable, enclose the information between two # characters. As you enter a program line with date values in this format, Visual Basic checks your syntax. If the date or time is illegal or nonexistent, you'll immediately get an error. Here's an example in which a Date variable, *dtmD*, is loaded with a specific date and time:

```
Dim dtmD As Date  
dtmD = #11/17/96 6:19:20 PM#
```

We'll use this value of *dtmD* throughout the following examples.

Several functions convert date and time numbers to Date type variables. DateSerial combines year, month, and day numbers in a Date value. In a similar way, TimeSerial combines hour, minute, and second numbers in a Date value:

```
dtmD = DateSerial(1996, 11, 17)  
dtmD = TimeSerial(18, 19, 20)
```

To combine both a date and a time in a Date variable, simply add the results of the two functions:

```
dtmD = DateSerial(1996, 11, 17) + TimeSerial(18, 19, 20)
```

To convert a string representation of a date or time to a Date value, use the DateValue and TimeValue functions:

```
dtmD = DateValue("11/17/96")  
dtmD = TimeValue("18:19:20")
```

Again, to convert both a date and a time at the same time, simply add the results of the two functions:

```
dtmD = DateValue("Nov-17-1996") + TimeValue("6:19:20 PM")
```

A wide variety of legal formats are recognized as valid date and time strings, but you do need to be aware of the expected format for the regional setting on a given system. In some countries, for example, the month and day numbers might be expected in reverse order.

Displaying a Date or a Time

The Format function provides great flexibility for converting a Date variable to a printable or displayable string. The following block of code shows the predefined named formats for these conversions:

```
Print Format(dtmD, "General Date") `11/17/96 6:19:20 PM  
Print Format(dtmD, "Long Date") `Sunday, November 17, 1996  
Print Format(dtmD, "Medium Date") `17-Nov-96  
Print Format(dtmD, "Short Date") `11/17/96  
Print Format(dtmD, "Long Time") `6:19:20 PM  
Print Format(dtmD, "Medium Time") `06:19 PM  
Print Format(dtmD, "Short Time") `18:19
```

Be aware that the results depend on your system's regional settings. You should run this code to see whether your results differ from mine.

In addition to the named formats, you can create your own user-defined formats for outputting date and

time data. For example, the following line of code formats each part of the date and time in a unique way and stores the result in a string variable.

```
strA = Format(dtmD, "m/d/yyyy hh:mm AM/PM")    `11/17/1996 06:19 PM
```

These user-defined formats are extremely flexible. For example, here's how you can generate the textual name of the month for a date:

```
strMonthName = Format(dtmD, "mmmm")      `November
```

See the online help for the Format function to get a detailed description of the many combinations of user-defined date and time formats you can use.

Extracting the Details

Several functions are available to extract parts of the Date variable. The following lines of code provide a quick reference to this group of related functions:

```
Print Month(dtmD)      `11
Print Day(dtmD)        `17
Print Year(dtmD)       `1996
Print Hour(dtmD)       `18
Print Minute(dtmD)     `19
Print Second(dtmD)     `20
Print WeekDay(dtmD)    `1
```

A set of built-in constants is provided by Visual Basic for the WeekDay result; *vbSunday* is 1, *vbMonday* is 2, and so on through *vbSaturday*, which is 7.

Date and Time Calculations

Date variables can be directly manipulated in a mathematical sense if you keep in mind that the unit value is a day. For example, you can easily create an application to calculate your age in days, as shown in Figure 3-5. To do so, simply subtract your date of birth (stored in a Date variable) from the Date function (which returns today's date).

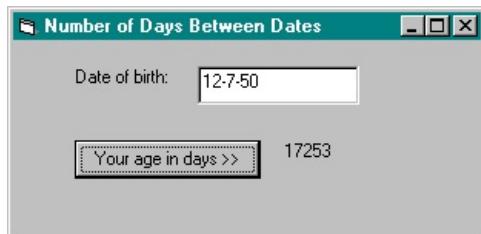


Figure 3-5. Using Date variables to calculate the number of days between dates.

The TimeSerial function provides a convenient way to convert a time value to a date value. Let's say, for instance, that you want to calculate the exact date and time 10,000 minutes from the current moment. Here's how to do this without getting tangled up in a lot of math:

```
dtmD = Now + TimeSerial(0, 10000, 0)
```

The TimeSerial function returns a value representing no hours, 10,000 minutes, and no seconds. This value is added to Now to calculate a Date variable containing the desired date and time. If you print the value of *dtmD*, you'll see a date and time that's roughly one hour short of exactly a week from now.

Date and Time Validity Checking

You can use an error trap when assigning a user-entered date or time string to a Date variable. If the date or time is not recognizable as a valid date or time, a "Type mismatch" error is generated by Visual Basic. Another approach to getting a valid date from the user is to use a Calendar control. This prevents errors by letting the user interactively select only a valid date from a one-month calendar page.

SEE ALSO

- The VBCal application in Chapter 31, "[Date and Time](#)," for a date selection dialog box that you can plug into your own applications

Dear John, How Do I... Work with Variants?

Variants are extremely flexible (some say too flexible), but they do allow for some clever new ways to structure and organize your data.

If not declared as something else, all variables default to Variant. You can store just about anything in a Variant variable, including arrays, objects, UDT structures, and other Variants. Arrays always contain like elements, which means you normally can't mix strings and numbers in the same array, for instance. But an array of Variants gets around this limitation. Consider the code below, which creates a Variant array and loads it with an integer, a string, and another Variant. To hint at the flexibility here, I've even stored a second Variant array in the third element of the primary Variant array. Remember, you can store just about anything in a Variant!

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim i          `Note that this defaults to Variant
    Dim vntMain(1 To 3) As Variant
    Dim intX As Integer
    Dim strA As String
    Dim vntArray(1 To 20) As Variant
    `Fill primary variables
    strA = "This is a test."
    For i = 1 To 20
        vntArray(i) = i ^ 2
    Next i
    `Store everything in main Variant array
    vntMain(1) = intX
    vntMain(2) = strA
    vntMain(3) = vntArray()
    `Display sampling of main Variant's contents
    Print vntMain(1)           `0
    Print vntMain(2)           `This is a test.
    Print vntMain(3)(17)       `289
End Sub
```

Notice, in the last executable line of the procedure, how the Variant array element within another Variant array is accessed. The `vntMain(3)(17)` element looks, and indeed acts, somewhat like an element from a two-dimensional array, but the subscripting syntax is quite different. This technique effectively lets you create multidimensional arrays with differing dimensions and data types for all elements.

For Each Loops

Variants serve an important role in For Each...Next loops. You can loop through each member of a collection, or even through a normal array, using a For Each statement. The only type of variable you can use for referencing each element is an Object or a Variant. When looping through arrays you *must* use a Variant. When looping through a collection you *may* use an Object variable, but you always have the option of using a Variant.

Flexible Parameter Type

The Variant type is useful as a parameter type, especially for object properties for which you want to set any of several types of data in a single property. You can pass just about anything by assigning it to a Variant variable and then passing this data as an argument.

Variant-Related Functions

You should be aware of several useful functions for working with Variants. TypeName returns a string describing the current contents of a Variant. The family of Is functions, such as IsNumeric and IsObject, provides fast logical checks on a Variant's contents. Search the online help for more information about these and other related functions.

Empty and Null

Be careful of the difference between an *Empty* Variant and a *Null* one: a Variant is Empty until it's been assigned a value of any type; Null is a special indication that the Variant contains no valid data. A Null value can be assigned explicitly to a Variant, and a Null value propagates through all calculations. The Null value most often appears in database applications, where it indicates unknown or missing data.

Data Type Coercion

Variants are very flexible, but you need to be careful when dealing with the complexities of automatic data conversions. For example, some of the following program steps may surprise you:

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim vntA, vntB
    vntA = "123"
    vntB = True
    Print vntA + vntB           `122
    Print vntA & vntB          `123True
    Print vntA And vntB = 0     `0
    Print vntB And vntA = 0     `False
End Sub
```

The first Print statement treats the contents of the two Variants as numeric values, and the second Print statement treats them as strings. The last two Print statements produce considerably different results based on the operational hierarchy, which isn't at all obvious. The best advice I can provide is this: Be cautious and carefully check out the results of your coding to be sure that the results you get are what you expect.

Dear John, How Do I... Work with Strings?

There are five new string functions built into the Visual Basic language:

- The Replace function replaces one set of characters within a string with another set. This makes it very easy to replace all occurrences of a string within a text box or other text source.
- The Split function breaks a string into an array of smaller strings, or *tokens*, based on some delimiter character, such as a space or tab.
- The Join function does the opposite of Split. It converts an array of tokens into a single string with each token separated from the adjacent ones by a specified delimiter character.
- The Filter function can be used with Split and Join. You use Filter to search an array of strings for some substring. Filter returns an array of strings that contain the specified substring.
- The InStrRev function—the opposite of the InStr function—lets you search from the end of a string toward the beginning. This is useful when you want to let users do selective search-and-replace tasks either forward (InStr) or backward (InStrRev).

Figure 3-6 shows a sample application, StringFun.VBP, which demonstrates each of the new functions. Each of the tasks performed by StringFun.VBP is described in the sections that follow.

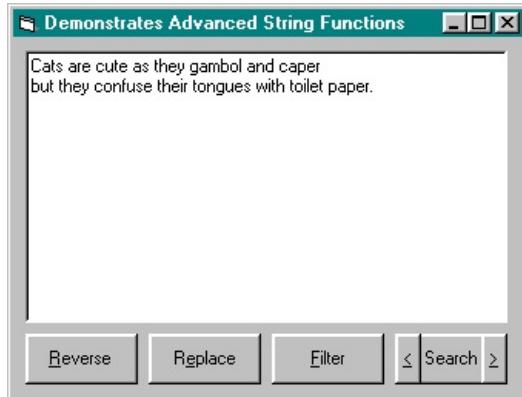


Figure 3-6. Click on each of the command buttons in StringFun.VBP to see a demonstration of each of the new string functions.

Replacing Characters in a String

One of the most common programming tasks is converting text from one format to another. Most conversions require replacing one set of delimiting characters—such as tabs, spaces, or brackets—with another. The Replace function handles those types of tasks in a single step. The following procedure shows how to use Replace to perform a very simple conversion by replacing spaces with carriage returns and vice versa.

```
'cmdReplace_Click uses Replace to
`do a global search and replace
Private Sub cmdReplace_Click()
    Static blnReplaced As Boolean
    `Replace spaces with carriage returns
    If Not blnReplaced Then
        txtString = Replace(txtString, " ", vbCrLf)
    `Replace carriage returns with spaces
    Else
        txtString = Replace(txtString, vbCrLf, " ")
    End If
    blnReplaced = Not blnReplaced
End Sub
```

Breaking Up and Rejoining Strings

Breaking and rejoining strings is commonly used when gathering and reordering information in a text box. The Split and Join functions can be used together to break text into words, reverse their order, and display them, as shown by the procedure below.

```
'cmdReverse uses Split and Join to
'reverse the order of words in a text box
Private Sub cmdReverse_Click()
    Dim strForward() As String
    Dim strReverse() As String
    Dim intCount As Integer
    Dim intUpper As Integer
    'Tokenize the text
    strForward = Split(txtString, " ")
    'Initialize the other array
    intUpper = UBound(strForward)
    ReDim strReverse(intUpper)
    'Reverse the order of words in strReverse
    For intCount = 0 To intUpper
        strReverse(intUpper - intCount) = strForward(intCount)
    Next
    'Detokenize the text
    txtString = Join(strReverse, " ")
End Sub
```

The key to understanding Split and Join is to think of them as conversion functions. Split converts a string into an array of strings and Join converts an array of strings into a single, continuous string. Once you've converted a string into an array of items, you can sort the list using standard array sorting procedures, or you can select items from the list by applying a filter, as described in the next section.

Applying Filters

The Filter function takes an array of strings and returns an array of elements that contain a specific substring. The Filter function is commonly used with the Split and Join functions to process lists of items in a text box or other text source. The procedure below shows how to apply a filter to words in a text box.

```
'cmdFilter_Click uses Filter to display only
'the words containing a specific set of characters
'This is useful when narrowing a list of items based
'on what a user types
Private Sub cmdFilterClick()
    Dim strFilter As String
    Dim strWords() As String
    'Get a substring to check for
    strFilter = InputBox("Show only words containing:")
    'If cancelled, then exit
    If strFilter = "" Then Exit Sub
    'Get rid of carriage returns
    txtString = Replace(txtString, vbCrLf, " ")
    'Split string into an array of single words
    strWords = Split(txtString, " ")
    'Get a list of the words containing strFilter
    strWords = Filter(strWords, strFilter)
    'Display the list in the text box
    txtString = Join(strWords)
End Sub
```

Searching for Strings

Finding a word or phrase in a block of text is a common text-editing task that most users expect from word processors or when presented with a large amount of text online. The InStr function provides the ability to search forward within a block of text and now the InStrRev function lets you reverse that search direction. The two procedures below demonstrate these types of searching within a text box.

```
'cmdBack_Click and cmdForward_Click
'use InStr and InStrRev to perform forward and
`backward searches through a text box.
`These techniques can be used to selectively
`replace text
Private Sub cmdBack_Click()
    Dim strFind As String
    On Error GoTo errNotFoundRev
        `Get a string to find
        strFind = InputBox("String to find:", "Search Backward")
        txtString.SelStart =
            InStrRev(txtString, strFind, txtString.SelStart) - 1
        txtString.SelLength = Len(strFind)
        Exit Sub
errNotFoundRev:
    MsgBox strFind & " not found.", vbInformation
End Sub

Private Sub cmdForward_Click()
    Dim strFind As String
    On Error GoTo errNotFoundFor

        `Get a string to find
        strFind = InputBox("String to find:", "Search Forward")
        txtString.SelStart =
            InStr(txtString.SelStart + 1, txtString, strFind) - 1
        txtString.SelLength = Len(strFind)
        Exit Sub
errNotFoundFor:
    MsgBox strFind & " not found.", vbInformation
End Sub
```

Strangely, the InStr and InStrRev functions order their arguments differently. In InStr, the start position of the search is the first argument and in InStrRev the start position is the third argument. Because of this inconsistency, it is easy to confuse the arguments when using these functions together.

Dear John, How Do I... Work with Objects?

Object variables not only store information but also perform actions. This makes them a very special sort of variable, and there are special considerations you should take into account when working with them.

New Objects

Objects are declared slightly differently than other variables. Visual Basic initializes most variables when they are first used. For instance, you declare a new Integer variable with Dim, and then when you first use the variable Visual Basic initializes its value to 0 automatically. Objects aren't initialized this way. If you declare an object variable and then check its type name, you'll get the answer "Nothing":

```
Dim frmX As Form1
Debug.Print TypeName(frmX)      `Displays "Nothing"
```

You can't do anything with an object variable until you create an instance of it. Use the New keyword to create a new object:

```
Dim frmX As Form1
Debug.Print TypeName(frmX)      `Displays "Nothing"
Set frmX = New Form1
Debug.Print TypeName(frmX)      `Displays "Form1"
```

The New keyword is important here. Just dimensioning an object variable creates a variable with that object type, but the variable is set to Nothing until you assign an object to it.

You also can use the New keyword when you declare an object variable, as follows:

```
Dim frmX As New Form1
Debug.Print TypeName(frmX)      `Displays "Form1"
```

Declaring an object variable with the New keyword tells Visual Basic to create a new object whenever it encounters the object variable in executable code and the object variable is set to Nothing. Since Dim statements aren't executable, Visual Basic waits to create the object till the Debug.Print statement above. This might seem like an unimportant detail, but it can lead to some unexpected results. For instance, the following code does not display "Nothing" as you might expect:

```
Dim frmX As New Form1
Set frmX = Nothing
Debug.Print TypeName(frmX)      `Displays "Form1"
```

The preceding code actually creates an instance of Form1 in the Set statement and immediately destroys it. Then it creates a new instance on the very next line. You can avoid this behavior by not declaring object variables As New. For instance, the following code yields the expected result:

```
Dim frmX As Form1
Set frmX = New Form1          `Create an instance of the form
Debug.Print TypeName(frmX)    `Displays "Form1"
Set frmX = Nothing           `Destroy the form
Debug.Print TypeName(frmX)    `Displays "Nothing"
```

In general, you should use

```
Dim Dear John, How Do I... As New
```

to create objects that will exist throughout the scope where they are declared and you should use

```
Dim Dear John, How Do I... As Dear John, How Do I...
Set Dear John, How Do I... New Dear John, How Do I...
```

to create objects which can be created and destroyed within their scope.

Existing Objects

An Object variable only *refers* to an object—this isn't the same thing as *being* the object. You can reassigned which object an Object variable refers to by using the Set statement, as shown here:

```
Dim frmX As New Form1      'Create a new object
Dim frmY As New Form1      'Create another object
Set frmX = frmY            'frmX and frmY both refer to the same object
```

You must use Set rather than simple assignment to assign an object reference to a variable. Most objects have a default property. Set tells Visual Basic to perform an object assignment rather than a property assignment.

Object Operations

You compare objects using the Is operator. Is tells you whether two Object variables refer to the same object. The following code shows how this works:

```
Dim frmX As New Form1      'Create a new object
Dim frmY As New Form1      'Create another object

Debug.Print frmX Is frmY   'Displays "False"
Set frmX = frmY            'frmX and frmY both refer to the same object
Debug.Print frmX Is frmY   'Displays "True"
```

Dead Objects

How long do objects live? Until they become Nothing!

Objects stick around as long as they have a variable that refers to them or, as is the case with Form objects, as long as they are visible. You can usually control the visibility of an object by using its Visible property. You can get rid of an object by setting its variable to another object or to Nothing:

```
Dim frmX As New Form1      'Create a new object
Dim frmY As Object          'Declare a variable of type Object
Set frmY = frmX              'frmX and frmY both refer to same object
Set frmX = Nothing           'frmY is still valid; object persists
Set frmY = Nothing           'Object goes away
```

At least, that's what's supposed to happen. Some objects aren't so careful. Early versions of some objects (including some from Microsoft) weren't so good about going away. Most objects are better behaved today, but it's still a good idea to be careful when working with objects you can't see. In general, objects you create in Visual Basic are well behaved, and objects in other products are usually, but not always, well behaved.

SEE ALSO

- Chapter 5, "[Object-Oriented Programming](#)," for more information about objects

Dear John, How Do I... Work with Predefined Constants?

There are actually several types of constants in Visual Basic, including some predefined constants provided by the system.

Compiler Constants

If you need to develop your applications in both 16-bit and 32-bit versions, you must use Visual Basic 4. This is the only version that provides a development environment for Windows 3.1 (16-bit), Windows 95 (32-bit), and Windows NT 3.51 (32-bit). To create applications that will run on both 16-bit and 32-bit operating systems, you'll need to use the Win16 and Win32 compiler constants. These constants indicate the system currently in use during development of your Visual Basic applications and let you select appropriate blocks of code to suit the conditions.

These constants are used with the #IfDear John, How Do I... ThenDear John, How Do I... #Else directives to select or skip over sections of code during compilation. Visual Basic versions 5 and 6 still support these constants, but Win32 is always *True* and Win16 is always *False* because these versions of Visual Basic run only on 32-bit versions of Windows.

Visual Basic Constants

A long list of predefined constants is provided automatically by Visual Basic for virtually every need. To see what's available, click the Object Browser button, select VBA or VBRUN from the top drop-down list, and click on the appropriate group of constants in the Classes list. A list of constants, each identifiable by the Constant icon and the *vb* prefix, is shown on the right side in the Members list, as you can see in Figure 3-7.

Notice that many components provide constants, and this is the place to find them when you need them.

You'll find that there are predefined constants for almost every control and function parameter imaginable—for example, *vbModal* and *vbModeless* for the form's Show method; *vbRed*, *vbBlue*, and *vbGreen* for color constants used in most graphics methods; and even *vbCr*, *vblf*, *vbCrLf*, and *vbTab* for common text characters. The following code shows how to insert a carriage return/linefeed character into a string:

```
strA = "Line one" & vbCrLf & "Line two"
```

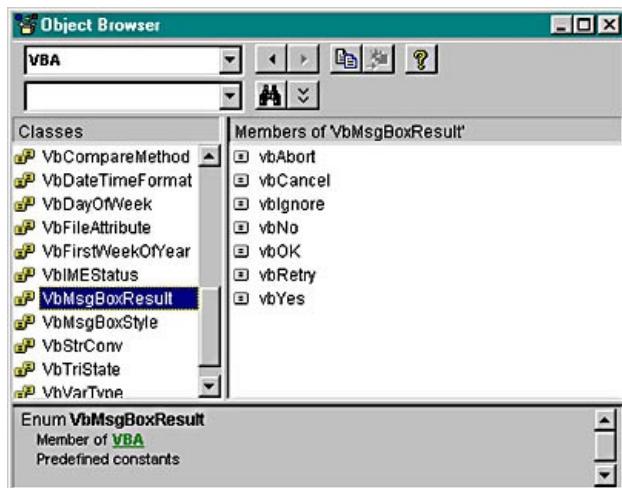


Figure 3-7. Built-in constants, as shown in the Object Browser.

Anytime you find yourself starting to type a numeric value for a property or a method argument, you should stop yourself and check to see whether there is already a constant defined by the system that you could use instead. This can help make your programs self-documenting, easier to read, and easier to maintain.

User-Defined Constants

Like earlier versions of Visual Basic, Visual Basic 6 lets you define your own constants in your programs by using the *Const* keyword. You can also create conditional compiler constants using the *#Const*

directive, which looks similar but is actually quite different. Constants created by #Const are to be used only with the #IfDear John, How Do I... ThenDear John, How Do I... #Else directives described earlier. Conversely, user-defined constants created with the Const keyword are not to be used as conditional compiler constants.

Some of the predefined constants are extremely helpful for building complex string constants. For example, the following code prints one constant to display digits in a column on a form. This same formatting was impossible without extra coding in earlier versions of Visual Basic. Now you can do all this formatting at design time. Note that only one executable statement is executed in the following procedure at runtime.

```
Option Explicit
```

```
Private Sub Form_Click()
    Const DIGITS =
        "1" & vbCrLf & _
        "2" & vbCrLf & _
        "3" & vbCrLf & _
        "4" & vbCrLf & _
        "5" & vbCrLf & _
        "6" & vbCrLf & _
        "7" & vbCrLf & _
        "8" & vbCrLf & _
        "9" & vbCrLf
    Print DIGITS
End Sub
```

Use the Private and Public keywords to define the scope of constants declared at the module level. If declared Private, the constants will be local to the module only, whereas Public constants are available project-wide. These keywords cannot be used for defining constants inside procedures. A constant declared within a procedure is always local to that procedure.

Enumerations

An enumeration (Enum for short) is a way that you can associate integer values with names. This can be helpful with property procedures. For example, a SpellOption property might have these valid settings:

```
'MailDialog class module-level declarations
Public Enum CheckedState
    Unchecked           'Enums start at 0 by default
    Checked             '1
    Grayed              '2, and so on
    'Can'tDeselect = 255 (You can set specific values too)
End Enum
```

The properties that set or return the state of check boxes in a dialog box would then be defined using Enum in the property's parameter declaration:

```
Public Property Let SpellOption(Setting As CheckedState)
    'Set the state of the Check Spelling check box
    frmSendMail.chkSpelling.Value = Setting
End Property

Public Property Get SpellOption() As CheckedState
    'Return the state of the Check Spelling check box
    SpellOption = frmSendMail.chkSpelling.Value
End Property
```

You can use Enums in any type of module. They appear in the Object Browser in the Classes list box; their possible settings are shown in the Members list box. Only the public Enums in public class modules are visible across projects, however.

Flags and Bit Masks

The operators And, Or, and Not are most often used for logical comparisons in decision structures. For example, an If statement can test several conditions using the And operator, as shown here:

```
If blnExit And Not blnChanged Then End
```

Logic is great, but to be a really smooth operator you'd better be bitwise too. *Bitwise* operations are bit-by-bit comparisons of two numbers to create a result. Bitwise operations are used when you want to stuff more than one meaning into a single numeric value. These values are referred to as *flags*. A good example is the Print dialog box's Flags property:

```
dlgPrint.Flags = cdlPDCollate Or cdlPDNoSelection
```

In most cases, using Or to combine flags is equivalent to using the addition operator (+). However, when flags conflict you'll get a different result using addition—for example, $1 + 1 = 2$, but $1 \text{ Or } 1 = 1$. This shouldn't happen if flag values are well thought out, but it's safer to use Or.

To reset flags to 0 or to check the value of a flag, use And. This code checks whether Print To File is selected in the Print dialog box:

```
'Check whether Print To File is selected  
If dlgPrint.Flags And cdlPDPrintToFile Then
```

The And operation above checks whether bit 5 is on (&H20 is the value of cdlPDPrintToFile); if it is, the statement is *True*. Checking flags this way is known as using a *bit mask*. A bit mask is the collection of bits you are checking—for example, 01101101 (in binary; &H6D in hex). If the value in question has 1s in all the same bit positions as the mask, the mask "fits."

Masking is how a signed integer was converted to an unsigned number earlier in this chapter. Signed integers have values from &H8000 (-32,768 in decimal) to &H7FFF (+32,767 in decimal). By masking out the proper bits, you can convert a signed integer to an unsigned value:

```
intShort = (lngLong And &H7FFF&) - (lngLong And &H8000&)
```

Dear John, How Do I... Create User-Defined Type (UDT) Structures?

The Type keyword is used to declare the framework of a UDT data structure. Notice that the Type command doesn't actually create a data structure; it only provides a detailed definition of the structure. You use a Dim statement to actually create variables of the new type.

You can declare a Public UDT structure in a standard module or a public class module. Within a form or a private class module, you must declare the UDT structure as Private. When combined with Variants, amazingly dynamic and adjustable data structures can result. The following code demonstrates a few of these details:

```
Option Explicit

Private Type typX
    a() As Integer
    b As String
    c As Variant
End Type
Private Sub Form_Click()
    'Create a variable of type typX
    Dim X As typX
    'Resize dynamic array within the structure
    ReDim X.a(22 To 33)
    'Assign values into the structure
    X.a(27) = 29
    X.b = "abc"
    'Insert entire array into the structure
    Dim y(100) As Double
    y(33) = 4 * Atn(1)
    X.c = y()
    'Verify a few elements of the structure
    Print X.a(27)      '29
    Print X.b          'abc
    Print X.c(33)      '3.14159265358979
End Sub
```

Notice the third element of the *typX* data structure, which is declared as a Variant. Recall that we can store just about anything in a Variant, which means that at runtime we can create an array and insert it into the UDT data structure by assigning the array to the Variant.

Memory Alignment

32-bit Visual Basic, in its attempt to mesh well with 32-bit operating system standards, performs alignment of UDT structure elements on 4-byte boundaries in memory (known as *DWORD alignment*). This means that the amount of memory your UDT structures actually use is probably more than you'd expect. More important, it means that the old trick of using LSet to transfer binary data from one UDT structure to another might not work as you expect it to.

Again, your best bet is to experiment carefully and remain aware of this potential trouble spot. It could cause the kind of bugs that are hard to track down unless you're fully aware of the potential for trouble.

Dear John, How Do I... Create New Data Types with Classes?

First, a little terminology. *Classes* or *class modules* are where objects are defined, just as Type statements are where UDT structures are defined. The actual object is an *instance* of the class. Classes define what kind of data the object can contain in its *properties* and how the object behaves through its *methods*.

Classes can range from very simple to vastly complex. Of course, it's best to start with the simple. This section shows you how to create an unsigned Integer data type using a class module. Chapter 5, "[Object-Oriented Programming](#)," covers more complex aspects of object-oriented programming.

Creating a New Data Type

The first section of this chapter showed you how to simulate unsigned integers in Visual Basic code. You can put that code into a UInt class to create a new, "fundamental" data type for working with unsigned integers.

The module level of the UInt class, shown below, defines the UDT structures and private variables used to return the high and low bytes from the integer. It's a good idea to specify the uses of a class, as well as to include comments within the class module. This makes the class more easily understood and more likely to be reused.

```
'Class UInt module level.
'Provides an unsigned integer type.
'

`Methods:
`    None

`Properties:
`    Value (default)
`    HiByte
`    LoByte

`Type structures used for returning high/low bytes
Private Type UnsignedIntType
    lo As Byte
    hi As Byte
End Type

Private Type SignedIntType
    n As Integer
End Type

`Internal variables for returning high/low bytes
Private mnValue As SignedIntType
Private muValue As UnsignedIntType
```

The class uses the private module-level variable *mnValue* to store the actual value of the property, but access to this variable is controlled through the Let Value and Get Value property procedures, as shown in the following example. These procedures perform the conversions needed to simulate an unsigned integer.

```
Property Let Value(lngIn As Long)
    mnValue.n = (lngIn And &H7FFF&) - (lngIn And &H8000&)
    LSet muValue = mnValue
End Property

Property Get Value() As Long
    Value = mnValue.n And &HFFFF&
End Property
```

Each property of the UInt class has a Let procedure in which the value of the property is assigned and a Get procedure in which the property value is returned. The HiByte and LoByte properties, for example,

are implemented using Let and Get procedures, as shown here:

```
Property Let HiByte(bytIn As Byte)
    muValue.hi = bytIn
    LSet mnValue = muValue
End Property

Property Get HiByte() As Byte
    HiByte = muValue.hi
End Property

Property Let LoByte(bytIn As Byte)
    muValue.lo = bytIn
    LSet mnValue = muValue
End Property

Property Get LoByte() As Byte
    LoByte = muValue.lo
End Property
```

Finally, the UInt class has a public function that returns the signed integer value:

```
Function Signed() As Integer
    Signed = mnValue.n
End Function
```

Using the New Data Type

To use the UInt class, be sure to declare the object variable as New, as shown below. This creates a new instance of the class.

```
Dim uNewVar As New UInt
```

The Value property of the UInt class is the *default* property. (For information about how to set the default property of a class, see Chapter 5, "[Object-Oriented Programming](#).") You can omit the property name Value when using the property, as shown here:

```
`Set the default property
uNewVar = 64552
```

The HiByte and LoByte properties can be called just like any other Visual Basic property, as shown here:

```
`Set the high byte
uNewVar.HiByte = &HFF

`Return the low byte
Print Hex(uNewVar.LoByte)
```

Chapter Four

Parameters

Visual Basic parameters may be optional or mandatory. They may require one data type or may accept any kind, including user-defined types, arrays, and enumerated constants (Enums). With such flexibility comes the responsibility to define parameters wisely so that error-causing values are not passed to procedures willy-nilly.

This chapter covers how and when to use named parameters, optional parameters, and data typing in your procedure definitions. Most of these features have been added or refined over the past several versions of Visual Basic. So, while they are not new to most of us, they may not be widely used in the code most of us work on day to day.

NOTE

In this chapter, an *argument* is a constant, a variable, or an expression that is passed to a procedure. A *parameter* is the variable that receives the argument when it is passed to the procedure.

Dear John, How Do I... Use Named Arguments?

The most common way to pass arguments in Visual Basic, and indeed in almost every programming language, is by position in a comma-delimited list. So, for instance, if you create a procedure that expects the parameters (*sngRed*, *sngGreen*, *sngBlue*), you always pass three values to the procedure and it's understood that the first argument will always represent *sngRed*, the second *sngGreen*, and the last *sngBlue*.

Now take a look at the following procedure and the code that calls it. Here I've used explicit parameter names to pass the three arguments, but in reverse order!

```
Option Explicit
```

```
Private Sub FormColor(sngRed As Single, sngGreen As Single, _
    sngBlue As Single)
    BackColor = RGB(sngRed * 256, sngGreen * 256, sngBlue * 256)
End Sub
```

```
Private Sub Form_Click()
    FormColor sngBlue:=0, sngGreen:=0.5, sngRed:=1      `Brown
End Sub
```

The real value of named arguments lies not so much in their ability to be passed in any order as in their ability to clearly pass a subset of parameters to a procedure that expects any of a large number of parameters. To take full advantage of named arguments in your applications, you need to make some or all of your parameters optional, which takes us right into the next question.

Dear John, How Do I... Use Optional Parameters?

Optional parameters are especially useful when you have a relatively long list of parameters for a procedure. Here's a simple modification of the previous example to show you how this works:

```
Option Explicit
```

```
Private Sub FormColor( _  
    Optional sngRed = 0, _  
    Optional sngGreen = 0, _  
    Optional sngBlue = 0 _  
)  
    BackColor = RGB(sngRed * 256, sngGreen * 256, sngBlue * 256)  
End Sub
```

```
Private Sub Form_Click()  
    FormColor sngGreen:=0.5      `Medium green  
End Sub
```

By adding the keyword **Optional** to the parameter declarations in the **FormColor** procedure, you can pass any or all of the parameters as named arguments to clearly indicate which parameters are passed. In this example, I've passed only the *sngGreen* parameter, knowing that the procedure will default to the value 0 for the optional parameters I didn't pass.

Notice that the default values are set in the parameter list. In Visual Basic 4, you had to check each optional parameter by using an **IsMissing** function and then setting a default value. Now, optional parameters can accept specific data types other than Variant. However, you still have to be sure that once a parameter is defined with the **Optional** modifier, all remaining parameters in the list are also declared as optional.

SEE ALSO

- The Metric application in Chapter 30, "[Development Tools](#)," for a demonstration of the use of optional parameters

Dear John, How Do I... Pass Parameter Arrays?

If you declare the last parameter in the parameter list of a procedure as a Variant array with the `ParamArray` keyword, you can pass a flexible number of arguments. The following code demonstrates this by converting any number of arguments to a vertically formatted list of strings:

```
Option Explicit

Private Function MakeVerticalList(ParamArray vntN()) As String
    Dim strA As String, i As Integer
    For i = LBound(vntN) To UBound(vntN)
        strA = strA + vbCrLf + CStr(vntN(i))
    Next i
    MakeVerticalList = strA
End Function

Private Sub Form_Click()
    Dim intA As Integer
    Dim sngB As Single
    Dim strC As String
    intA = 123
    sngB = 3.1416
    strC = "This is a test."
    Print MakeVerticalList(intA, sngB, strC)
    Print
    Print MakeVerticalList("This", "time", "we'll", "pass five", _
        "string arguments.")
End Sub
```

Notice that I called `MakeVerticalList` twice, the first time with a variety of types of arguments—three in all—and the second time with five string arguments.

Figure 4-1 shows the results displayed when you click on the form.

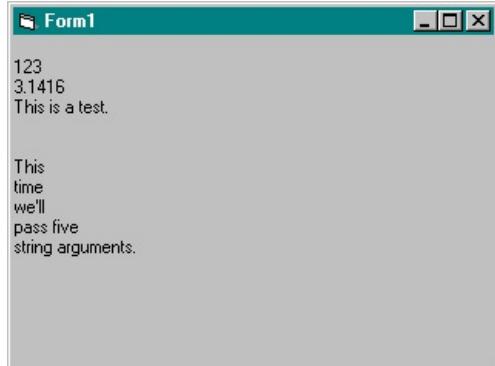


Figure 4-1. Results of passing first three and then five arguments to the `MakeVerticalList` function.

Dear John, How Do I... Pass Any Type of Data in a Parameter?

Remember that Variant variables can contain just about any kind of data imaginable. This opens the door to passing any type of data you want to a parameter as long as the parameter has been declared as Variant. For example, in this code sample I've passed an integer, a string, and an array to a small procedure:

```
Option Explicit

Private Sub VariantTest(vntV)
    Print TypeName(vntV)
End Sub

Private Sub Form_Click()
    Dim dblA(3, 4, 5, 6, 7) As Double
    VariantTest 123
    VariantTest "This is a test string."
    VariantTest dblA
End Sub
```

The Variant parameter *vntV*, defined as the only parameter in the VariantTest procedure, accepts whatever you want to pass it. The TypeName function is then used to display the type of data that was passed.

Figure 4-2 shows the results of running this sample code.

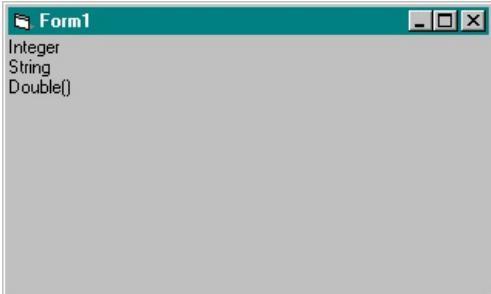


Figure 4-2. Results of a procedure that uses a Variant parameter.

Dear John, How Do I... Use Enums in Parameters?

You use enumerations (Enums) to restrict parameters to a predefined set of values. An Enum defines a set of symbolic values, much the same as Const, but the name of the Enum can be included in a procedure's parameter type declaration. When you use the procedure, Visual Basic's Auto List Members feature displays a list of the possible values for the argument. To see how this works, type in the following code:

```
Enum Number
    Zero
    One
    Two
    Three
End Enum

Sub ShowNumber(Value As Number)
    MsgBox Value
End Sub

Sub Form_Load()
    ShowNumber One
End Sub
```

After you type *ShowNumber* in the *Form_Load* event procedure, Visual Basic displays the list of Enums to choose from. A similar thing happens when you use an Enum in the parameter list of a user control property. The following code shows a simple *Value* property in a user control:

```
Enum Number
    Zero
    One
    Two
End Enum

Dim mValue As Number

Property Get Value() As Number
    Value = mValue
End Property

Property Let Value(Setting As Number)
    mValue = Setting
End Property

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    mValue = PropBag.ReadProperty("Value", One)
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "Value", mValue, One
End Sub
```

The *ReadProperties* and *WriteProperties* event procedures maintain the *Value* property setting in the Properties window. When you select the *Value* property in the Properties window, you will see a list of possible values from the *Number* Enum: *Zero*, *One*, and *Two*, as shown in Figure 4-3 following.

Another bit of helpful information is that Enums automatically coerce floating-point values to long integer values. For example, if you pass the value *0.9* to the *ShowNumber* procedure shown earlier, it will display *1*. Also, Enums don't provide any special type checking—invalid settings are gleefully accepted. You'll have to write code to check whether a passed-in value is valid.

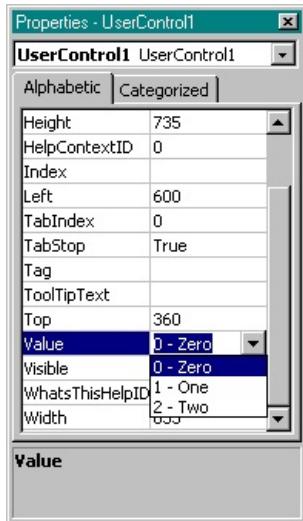


Figure 4-3. List of possible values for the *Value* property from the *Number* Enum in the Properties window.

Chapter Five

Object-Oriented Programming

Perhaps the most important advanced features in Visual Basic are those that allow you to do true object-oriented programming (OOP). This is a huge topic, and indeed many pages of Microsoft's manuals deal with this important subject in great detail. Instead of repeating all these details, in this chapter I'll highlight the main concepts, provide simple code examples, and give you a foundation from which to further your studies.

Object-oriented programming in Visual Basic is a fascinating subject. Class modules let you structure your applications in ways that you never could before, collections provide a way to flexibly organize and structure the objects you create, and ActiveX technology provides the mechanism to share and use objects across applications. You'll find yourself able to grasp the organization of much bigger projects and to keep this structure clearly in mind as you work with objects. Team programming and single programmer projects alike are made easier by this object-oriented approach.

There's a lot of new information to be absorbed when you first get into the object-oriented programming aspects of using Visual Basic. With a little persistence, and by experimenting with the examples provided in this book or elsewhere, you'll soon start to get the hang of it. For me, the moment of enlightenment came when I suddenly realized that creating objects is actually a lot of fun!

In this chapter, we'll create a sample object and a sample ActiveX EXE, and we'll take a look at one way to work with collections of objects. We will also discuss polymorphism and friend methods.

Dear John, How Do I... Choose Between an ActiveX EXE and an ActiveX DLL?

ActiveX EXEs and ActiveX dynamic link libraries (DLLs) let you combine your objects into components that can provide these objects to other applications (clients) through Automation. Both types of ActiveX components expose some or all of their objects to external applications.

A client application that creates and uses instances of an ActiveX EXE's exposed objects uses them *out-of-process*, which means that the code in the ActiveX EXE runs in its own thread and in its own workspace, separate from the code space of the client application.

On the other hand, an ActiveX DLL can't run as a stand-alone application but instead provides a dynamic link library of objects for applications to use *in-process*. This means that the code from the ActiveX DLL runs in the same code space as the calling application, resulting in faster, more efficient program execution. Both types of ActiveX components are useful, and Visual Basic lets you build both types without resorting to C or other languages.

Dear John, How Do I... Create All My Objects in External ActiveX Components?

You don't have to! There's a lot of hype and interest and great reasons to create and use objects in ActiveX components, but an excellent way to get started creating your own objects is to simply add class modules to your Standard EXE Visual Basic projects.

Objects defined by class modules within your project are automatically private to your application. (If you want to let other applications create instances of your objects, well then, we're back to talking about ActiveX components.) Your application can create one or multiple instances of each class module-defined object, and each copy has its own set of data. One of the advantages of these objects is that the memory used by each instance is recovered immediately when the object is destroyed. Probably the biggest advantage, however, is the increased structure, understandability, and organization that OOP techniques bring to your programming efforts. Getting rid of Basic's line numbers a few years back was a huge step in the right direction, and the addition of objects is another quantum leap.

The next two sections provide a relatively simple example of an object created in a Standard EXE project. This is the simplest way to start working with objects.

Dear John, How Do I... Create a New Object?

The short answer: create a class module. Class modules open the door to some powerful new ways of structuring programs. The printed manuals and online help are the best resources to study for all the intricacies of class modules and the objects you can create with them. Here I'll present a simple example, just enough to whet your appetite and provide a framework for learning. Throughout this book, you'll find numerous working examples of class modules—they provide a great way to create structured code, and I like using them.

Loan—A Class Module Example

To demonstrate many of the most important features of objects, I've built a relatively simple class module that lets you create Loan objects. The Loan object is used to calculate the payment schedule over the life of a loan. You could easily add more methods and properties to this class or restructure the way it works, but I kept it fairly simple on purpose, to provide a working model for you to study.

To create a class module, start a new Standard EXE project, choose Add Class Module from the Project menu, and double-click the Class Module icon in the Add Class Module dialog box. Add the following code, change the Name property to *Loan*, and save the module as LOAN.CLS. The code for the Loan class is followed by explanations of the different parts of the code.

```
'LOAN.CLS - This is a class module that provides a
`blueprint for creating Loan objects
```

```
Option Explicit
```

```
'This variable is known only within this class module
Private mintMonths As Integer      `Number of months of loan

`~~~Property (R/W): Principal
Public Principal As Currency

`~~~Property (R/W): AnnualInterestRate
Public AnnualInterestRate As Single

`~~~Property (R/W): Months
`Lets user assign a value to Months property
Property Let Months(intMonths As Integer)
    mintMonths = intMonths
End Property

`Gets current value of Months property for user
Property Get Months() As Integer
    Months = mintMonths
End Property

`~~~Property (R/W): Years
`Lets user assign a value to Years property
Property Let Years(intYears As Integer)
    mintMonths = intYears * 12
End Property
`Gets current value of Years property for user
Property Get Years() As Integer
    Years = Round(mintMonths / 12#, 0)
End Property

`~~~Property (R/O): Payment
`Gets calculated Payment for user
Property Get Payment() As Currency
    Dim sngMonthlyInterestRate As Single
    `Verify that all properties are loaded
    If PropertiesAreLoaded() Then
        sngMonthlyInterestRate = AnnualInterestRate / 1200
```

```

        Payment = (-sngMonthlyInterestRate * Principal) / 
            ((sngMonthlyInterestRate + 1) ^ (-mintMonths) - 1)
    Else
        Payment = 0
    End If
End Property

`~~~Property (R/O): Balance()
`Gets array of loan balances
Property Get Balance() As Currency
    Dim intCount As Integer
    Dim curPayment As Currency
    ReDim curBalance(0) As Currency
    If PropertiesAreLoaded() Then
        ReDim curBalance(mintMonths)
        curBalance(0) = Principal
        curPayment = Round(Payment, 2)      `Rounds to nearest penny
        For intCount = 1 To mintMonths
            curBalance(intCount) = curBalance(intCount - 1) * _
                (1 + AnnualInterestRate / 1200)
            curBalance(intCount) = curBalance(intCount) - curPayment
            curBalance(intCount) = Round(curBalance(intCount), 2)
        Next intCount
    End If
    Balance = curBalance
End Property

`~~~Method: Reset
`Initializes all properties to start over
Public Sub Reset()
    Principal = 0
    AnnualInterestRate = 0
    mintMonths = 0
End Sub

`Private function to check if all properties are properly loaded
Private Function PropertiesAreLoaded() As Boolean
    If Principal > 0 And AnnualInterestRate > 0 And mintMonths > 0 Then
        PropertiesAreLoaded = True
    End If
End Function

```

The Loan objects created in the main program will have properties and methods, just like other objects. The simplest way to create readable and writable properties is to declare public variables. Principal and AnnualInterestRate, declared at the top of the LOAN.CLS listing, are two such properties:

```

Public Principal As Currency
Public AnnualInterestRate As Single

```

One variable, *mintMonths*, is declared Private, as shown below, so that the outside world will be unaware of its existence. Within the Loan object, you can be assured that the value contained in this variable is always under direct control of the object's internal code.

```

Private mintMonths As Integer      `Number of months of loan

```

NOTE

You might wonder about the *mint* prefix to this variable's name. The *m* part indicates this variable is declared at the module level, and the *int* part indicates the variable is of type Integer. You also might be wondering why the public variables don't have any Hungarian

Notation prefixes. This is because public variables behave like properties, and the accepted standard for properties exposed to the outside world is avoidance of prefixes. This makes sense when you realize the purpose of variable prefixes is to improve the readability of the internal code of a class or other code module. Properties, on the other hand, are exposed to the external world, where clarity of naming matters more than understanding the underlying code.

The variable *mintMonths* stores the number of months of the loan. I could have made this a simple public variable, but I've designed the Loan object with two related public properties, Years and Months, either of which can be set by the user to define the length of the loan. These properties then internally set the value of *mintMonths*. Read on to see how these two properties are set up to do this.

The Property Let statement provides a way to create a property in your object that can take action when the user assigns a value to it. Simple properties, such as Principal, just sit there and accept whatever value the user assigns. The Months property, on the other hand, is defined in such a way that you can add code that will be executed whenever a value is assigned to the property. In this case, the code is a simple assignment of the number of months to the local *mintMonths* variable for safekeeping:

```
Property Let Months(intMonths As Integer)
    mintMonths = intMonths
End Property
```

The Property Let and Property Get statements work hand in hand to build one writable and readable property of an object. Here the Property Get Months statement provides a way for the user to access the current contents of the Months property:

```
Property Get Months() As Integer
    Months = mintMonths
End Property
```

Stop and think about what I've just done here. From the outside world, the Months property appears to be a simple variable that can store values assigned to it, and it supplies the same value when the property is accessed. But when you look at the implementation of the Months property from a viewpoint inside the Loan class module, you see a lot more going on. Months is not just a simple variable. Code is activated when values are written to or read from the property, and it's possible for this code to take just about any action imaginable. It's a powerful concept!

If you don't add a corresponding Property Get statement to go along with a Property Let statement, the defined property will be write-only. Conversely, a Property Get without a corresponding Property Let results in a read-only property. For some properties, this can be a good thing. This is true for the Payment property, which is described later in this chapter.

The Years property, shown below, provides a second, alternative, property that the user can set to define the length of the loan. Notice that the value set into the Years property is multiplied by 12 internally to convert it to months. This detail is hidden from the user.

```
Property Let Years(intYears As Integer)
    mintMonths = intYears * 12
End Property
```

```
Property Get Years() As Integer
    Years = Round(mintMonths / 12#, 0)
End Property
```

NOTE

The Round function is new in Visual Basic 6. I've used it in the Property Get Years procedure to round off the calculated number of years to zero digits after the decimal point, or to the nearest whole year.

Compare the Years and Months property procedures to see why I created the local variable *mintMonths* to store the actual length of the loan separately from the properties used to set this value. Again, the implementation of the Years and Months properties is encapsulated within the Loan object, and the details of how these properties do their thing is hidden from the outside world.

When the user accesses the value of the Loan object's Payment property, shown below, a complicated calculation is triggered, and the payment amount is computed from the current value of the other property settings. This is a clear example that shows why Property Let and Property Get statements add useful capabilities beyond those provided by properties that are created by simply declaring a public variable in a class module.

```
Property Get Payment() As Currency
    Dim sngMonthlyInterestRate As Single
    'Verify that all properties are loaded
    If PropertiesAreLoaded() Then
        sngMonthlyInterestRate = AnnualInterestRate / 1200
        Payment = (-sngMonthlyInterestRate * Principal) /
            ((sngMonthlyInterestRate + 1) ^ (-mintMonths) - 1)
    Else
        Payment = 0
    End If
End Property
```

The Balance property, which is set up as a read-only property because I define it only in a Property Get statement with no corresponding Property Let, is shown next.

```
Property Get Balance() As Currency()
    Dim intCount As Integer
    Dim curPayment As Currency
    ReDim curBalance(0) As Currency
    If PropertiesAreLoaded() Then
        ReDim curBalance(mintMonths)
        curBalance(0) = Principal
        curPayment = Round(Payment, 2)      `Rounds to nearest penny
        For intCount = 1 To mintMonths
            curBalance(intCount) = curBalance(intCount - 1) *
                (1 + AnnualInterestRate / 1200)
            curBalance(intCount) = curBalance(intCount) - curPayment
            curBalance(intCount) = Round(curBalance(intCount), 2)
        Next intCount
    End If
    Balance = curBalance
End Property
```

The Balance property demonstrates two new features of Visual Basic. In addition to the new Round function, which is used several places throughout the Loan class, the Balance property returns an entire array of numbers rather than just a single value. Take a close look at the Property Get declaration for this property. The return type is stated *As Currency()*. The parentheses indicate that an array is to be returned. In the Balance property's code, I declared a local array of type Currency, called curBalance, and after expanding and filling this dynamic array with values, assigned it to Balance, the name of the property. Voila! The whole array of monthly loan balances is returned to the calling application.

If anything goes wrong here—if the user hasn't set all the pertinent properties first, for example—the property will return an array dimensioned to 0, with the value 0 in it. This is a simplified way to handle potential errors. My intention is not to cover the best ways to handle all potential errors here but simply to show you an example class object in a simplified, straightforward manner.

Methods, like properties, are an important aspect of the interface an object presents to the outside world. I've added one simple method to the Loan object so that you can get a feel for how methods work. The Reset method is simply a Sub procedure designed to erase the loan's property values. You don't need to call this method unless multiple loan calculations are to be made using the same Loan object.

```
Public Sub Reset()
    Principal = 0
    AnnualInterestRate = 0
    mintMonths = 0
End Sub
```

NOTE

The Class_Initialize and Class_Terminate event procedures provide a way to automatically initialize variables and perform housekeeping at the time an object is created and when it is destroyed.

Finally, I added the PropertiesAreLoaded private function to show how procedures that are not known to the outside world can be added to an object. These private procedures are called only from other code within a class. In this case, since I needed to check in more than one place to see whether the object's properties were loaded and ready to go, all the code required to check all the properties is contained in one utility procedure.

```
Private Function PropertiesAreLoaded() As Boolean
    If Principal > 0 And AnnualInterestRate > 0 And mintMonths > 0 Then
        PropertiesAreLoaded = True
    End If
End Function
```

NOTE

If you don't explicitly assign a return value to a Function procedure, the procedure will return the default value for its type. For example, by default, Variant functions return Empty, Integer functions return the value 0, and Boolean functions return False. That's why we allow the PropertiesAreLoaded function to appear to return nothing when the test is False.

SEE ALSO

- The next section, "[Dear John, How Do I... Use My New Object?](#)" for a demonstration of how to use objects

Dear John, How Do I... Use My New Object?

The Loan class module doesn't actually create an object; it just provides a blueprint, or a definition, that lets your program stamp out one or more real Loan objects as needed. Let's complete the example by adding the LOAN.CLS module to a simple project and doing something with it.

Start with a new Standard EXE project containing just a single form. I'll keep this form very simple as its only purpose is to create and demonstrate a Loan object. Change the form's Caption property to *Please Click on This Form*. Add the LOAN.CLS class module created in the previous section to your project, and add the following lines of code to your form. Following the code listing are explanations of the different parts of this code.

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim loanTest As New Loan
    Dim intCount As Integer
    Dim curBalance() As Currency

    'Clear the face of the form
    Cls
    'Set loan parameters
    loanTest.Reset
    loanTest.Principal = 1000
    loanTest.Months = 12
    loanTest.AnnualInterestRate = 8.5
    'Display parameters used
    Print "Principal: " , , Format(loanTest.Principal, "Currency")
    Print "No. Months: " , , loanTest.Months
    Print "(Years:) " , , loanTest.Years
    Print "Interest Rate:" , , Format(loanTest.AnnualInterestRate _
        / 100, "Percent")
    Print "Monthly Payment: " , Format(loanTest.Payment, "Currency")
    Print
    'Get array of monthly balances
    curBalance() = loanTest.Balance()
    'Display payment schedule
    For intCount = LBound(curBalance) To UBound(curBalance)
        Print "Month: " ; intCount,
        Print "Balance: " ; Format(curBalance(intCount), "Currency")
    Next intCount
End Sub
```

Figure 5-1 shows the form during development.

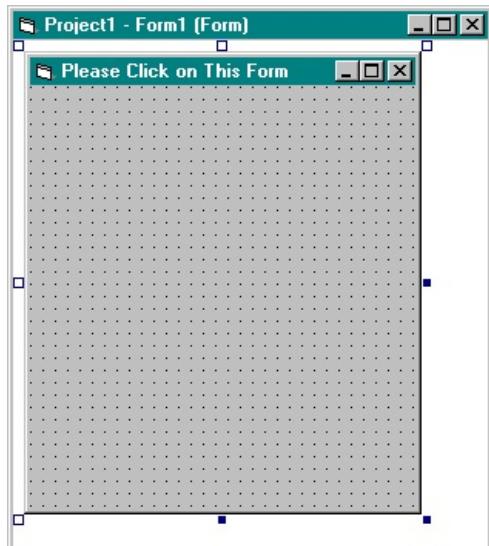


Figure 5-1. The form used to demonstrate the Loan object.

A new Loan object named loanTest is created when you click on the form, as shown in the code below. The Loan object will be destroyed automatically, just like any other local variable, when it goes out of scope.

```
Dim loanTest As New Loan
```

The following three lines assign values to three of the properties we've created for the Loan object. Review the Loan class module to see how the Months property works differently from the Principal and AnnualInterestRate properties.

```
loanTest.Principal = 1000
loanTest.Months = 12
loanTest.AnnualInterestRate = 8.5
```

As you output results, various properties of the Loan object are accessed as required. This is shown in the following code. In most cases, the loanTest object simply hands back each value as it is currently stored in the object. In the case of the loanTest.Payment property, however, a complex calculation is triggered. The calling code is blissfully unaware of when, where, and how the calculations are performed.

```
Print "Principal: ", , Format(loanTest.Principal, "Currency")
Print "No. Months: ", , loanTest.Months
Print "(Years:) ", , loanTest.Years
Print "Interest Rate:", , Format(loanTest.AnnualInterestRate _
    / 100, "Percent")
Print "Monthly Payment: ", Format(loanTest.Payment, "Currency")
Print
```

Calling the Balance property, shown here, tells the loanTest object to prepare and return an array of loan balances, in preparation for outputting each month's balance in the next few lines of the program:

```
`Get array of monthly balances
curBalance() = loanTest.Balance()
```

The last few lines of our sample program display an amortized table of the loan's balance at the end of each month of the loan:

```
`Display payment schedule
For intCount = LBound(curBalance) To UBound(curBalance)
    Print "Month: "; intCount,
    Print "Balance: "; Format(curBalance(intCount), "Currency")
Next intCount
```

When the end of the Form_Click event procedure is reached, all variables declared in the procedure go out of scope and are automatically destroyed by the system. This includes the instance of the Loan object, with all its code and data.

Figure 5-2 shows the results of running this program. Notice that you could greatly improve and expand on this simple program without ever having to touch the contents of the Loan class module.

A carefully designed object, in the form of a class module, lets you extend Visual Basic by adding some new programmable objects to your bag of tricks. Once you get an object the way you want it, all you need to document for future reference and use is the object's *interface*, which is its set of public properties and methods. The internal workings and implementation of the object can be mentally "black-boxed" and ignored. Imagine the object encased in a hard shell, with just the defined interface visible from the outside world. This is what is meant by the object-oriented programming term *encapsulation*. The object itself is responsible for the correct implementation of its features and behavior, and the calling application is responsible for understanding and correctly using the object's interface only. This encapsulation makes it much easier to create and debug larger applications. That demonstrates the true power and beauty of object-oriented programming!

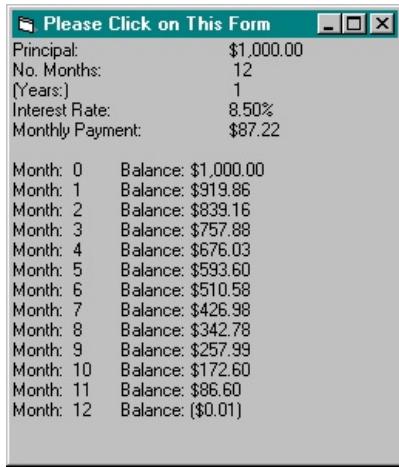


Figure 5-2. Output generated by using a Loan object.

The Loan class was simply added as another module to the sample application. This is a great way to use class modules, but you can also combine one or more class modules into an ActiveX component.

SEE ALSO

- The section "[Dear John, How Do I... Create and Use an ActiveX EXE?](#)" later in this chapter, and Chapter 27, "[Advanced Programming Techniques](#)," for more information about ActiveX components
- The Lottery application in Chapter 29, "[Graphics](#)," for a demonstration of the use of objects

Dear John, How Do I... Set a Default Property for My Object?

I prefer to always set all properties explicitly rather than by using the shortcut for an object's default property, but some people prefer to do it the other way. For example, the Text property is the default property for the TextBox control, which means that you can set the Text property explicitly or by default. Both of these lines have exactly the same effect:

```
Text1.Text = "This string is assigned to the Text property."  
Text1 = "This string is assigned to the Text property."
```

It's not well documented, but Visual Basic now lets you assign a default property to your own objects, if you like to use this technique. Let's see how this is done by setting the Loan object's Principal property as its default property. Bring up the code for the Loan class, and from the Tools menu choose Procedure Attributes to display the Procedure Attributes dialog box. From the Name drop-down list, select the Principal property. Click the Advanced button, select the (Default) option from the Procedure ID drop-down list, and click OK. The Principal property is now the default for the Loan object.

To verify the correct operation of this default property, bring up the code for the form you created to test the Loan object. Find the line on which the Principal property is set to 1000, and edit the line to remove the .Principal part, as shown here:

```
loanTest = 1000
```

Run the test application, and verify that it still calculates the loan values correctly.

Dear John, How Do I... Create and Use an ActiveX EXE?

ActiveX EXEs and ActiveX DLLs are similar creatures. In Chapter 27, "[Advanced Programming Techniques](#)," I'll walk you through the creation of an ActiveX DLL; here I'll walk you through the creation of an ActiveX EXE. An ActiveX EXE can run by itself, yet it also provides objects for use by external applications. As mentioned, these objects run *out-of-process*, which means that they run in their own code space, separate from a client application's code space.

Chance—An ActiveX EXE Example

In this example, we'll build a very simple ActiveX EXE containing the definition for one type of object. Or, to use the correct terminology, we'll create one *class*. The class can then function as a template to produce copies of the object. Once the ActiveX EXE is up and running and automatically registered with the system, we'll build a second, more conventional Visual Basic application that will create a couple of these objects on the fly and manipulate properties and methods of these objects to get the desired results.

We'll create an object named Dice in an ActiveX EXE component named Chance. Even though I've named the object Dice, I've provided properties that let the calling application set the number of sides so that each die can act more like a coin (if it has two sides), like a dodecahedron-shaped die (if the number of sides is set to 12), and so on.

DICE.CLS

Let's jump right in and start building the ActiveX EXE. Start a new project and double-click the ActiveX EXE icon in the New Project dialog box. Add the following lines of code to the class module, change its Name property to *Dice*, save this class module as DICE.CLS, and save the project as CHANCE.VBP:

```
Option Explicit
`Declare properties that define the range of possible values
Public Smallest As Integer
Public Largest As Integer

`Declare a read-only property and define its value
`as a roll of the die
Public Property Get Value() As Integer
    Value = Int(Rnd * (Largest - Smallest + 1)) + Smallest
End Property

`Use a method to shuffle the random sequence
Public Sub Shuffle()
    Randomize
End Sub
```

This class module defines three properties and one method for the Dice object. Smallest and Largest are two properties that I've declared as public variables so that the client application (which is created a little later on) can set the range of returned values—in this case, the number of sides of the die. Properties of objects are often simply declared as public variables in this way.

Value is another property of the Dice object, but in this case we want to perform a little calculation before handing a number back to the client application. The Property Get statement provides the mechanism for performing any extra steps we specify when the property's value is requested. In this case, a random number is generated to simulate a roll of the die, and this number is returned as the value of the property. (For the mathematical purists among us, the number is not truly random, but it's close enough for our needs.) The Value property is read-only for the client application because the Property Get statement was used but there is no corresponding Property Let statement. Adding a Property Let statement would enable the client application to write a new value to the Value property. In this example, we just want to return a random roll of the die, so we have no need to let the client application set a value for this property.

Shuffle is a simple method I've added so that the object will have at least one method to show for itself. When this method is called, the object will randomize the random number generator to effectively shuffle the outcome of rolls of the die.

Let's add a little code that runs when the Chance ActiveX EXE starts up. This will make it easier to follow the action later on. From the Project menu, choose Add Module and then double-click the Module icon in the Add Module dialog box. Add the following code to this module, and save it as CHANCE.BAS:

```
Option Explicit
Sub Main()
    Dim diceTest As New Dice
    diceTest.Smallest = 1
    diceTest.Largest = 6
    diceTest.Shuffle
    MsgBox "A roll of two dice: " &_
        diceTest.Value & "," & diceTest.Value,_
        0, "Message from the Chance ActiveX EXE"
End Sub
```

From the Project menu, choose Project1 Properties to open the Project Properties dialog box. From the Startup Object drop-down list, select Sub Main, enter *Chance* in the Project Name field, enter *Chance - ActiveX EXE Example* in the Project Description field, and then click OK. Finally, from the File menu, choose Make Chance.exe. In the displayed Make Project dialog box, select a location and click OK. This will compile and automatically register the Chance ActiveX EXE.

Testing the ActiveX EXE Component

Now that the Chance ActiveX EXE is compiled and registered, we can reference it and use its objects from any application that supports ActiveX programming. Let's demonstrate this by creating a second Visual Basic application that uses the ActiveX EXE we just created.

Start a new Standard EXE project, save its default form as DICEDEMO.FRM, and save the project itself as DICEDEMO.VBP. Change the form's Caption property to *Taking a Chance with Our ActiveX EXE*, and add a single command button. I changed the button's Caption property to *Roll 'em* and moved it off to the upper right side of the form. Add the following code to the form to complete it, and then save your work:

```
Option Explicit
Dim diceTest As New Dice
Private Sub Command1_Click()
    diceTest.Shuffle

    diceTest.Smallest = 1
    diceTest.Largest = 2
    Print "Coin:" & diceTest.Value,
    diceTest.Largest = 6
    Print "Dice:" & diceTest.Value,
    diceTest.Largest = 12
    Print "Dodecahedron:" & diceTest.Value
End Sub
```

Figure 5-3 shows the form during development.

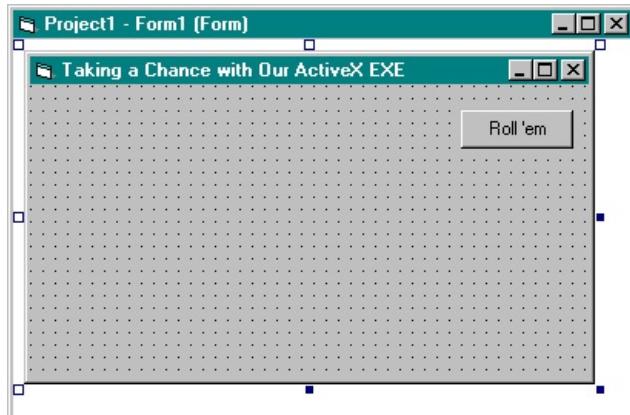


Figure 5-3. The DiceDemo form.

This form creates an instance of our new Dice object, so we need to add a reference to the Chance ActiveX EXE in order for the current project to know where the Dice object is defined. Choose References from the Project menu, and locate the check box labeled Chance - ActiveX EXE Example in the References dialog box. (Remember that this is the project description we gave the Chance project.) Check this box, and then click OK to add this reference to the DiceDemo project.

That should do it! Run the program and notice what happens. First our new test form opens, ready for us to click the Roll 'em button. When you click this button, the Dice object is created, and rolls of 2-sided, 6-sided, and 12-sided dice are tallied and displayed on the form. Each click creates another line of pseudorandom values. Figure 5-4 shows the results of clicking this button several times.

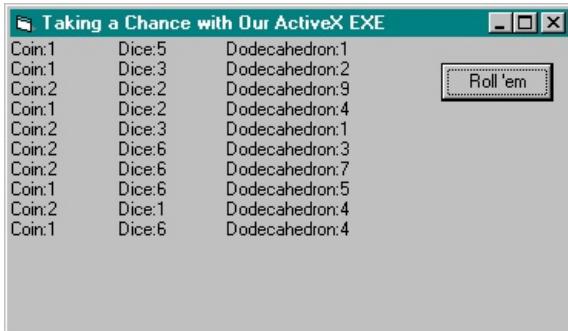


Figure 5-4. The DiceDemo application after the Roll 'em button has been clicked several times.

One other significant message appears the first time you click the Roll 'em button. As the Chance ActiveX EXE loads into memory, its Sub Main startup code displays a message box, and it also rolls a pair of dice using the Dice object defined within Chance's interior. The result is shown in Figure 5-5.

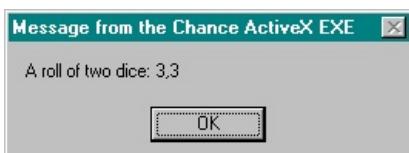


Figure 5-5. The Chance ActiveX EXE displays this message when it loads.

This message box appears only once, when the Chance component first loads. In case you're wondering, the Chance ActiveX component will unload from memory when the last reference to any of its objects goes away—which happens when we terminate the DiceDemo application.

Dear John, How Do I... Create an Object That Displays Forms?

The Loan and Dice sample objects shown earlier in this chapter don't display visible interfaces, such as forms, to the user. Many objects don't present a visible interface, but there will be cases in which you'll want to create objects that use forms to collect data from the user.

It's very easy to add a form to an ActiveX EXE or ActiveX DLL project, but you need to be aware of one major "gotcha"—objects that display forms need to carefully manage the creation and destruction of those forms. A client application will never interact directly with a form in an ActiveX component (these forms are private to the component), so it's important that all housekeeping of the form be correctly implemented by the object within the component that is responsible for the form. Otherwise, instances of the object can remain hidden in memory, consuming resources and occasionally causing unexpected results.

The following code shows the class module definition for an object named User. This object is used to gather name and password information and displays a modal form with two text boxes, named *txtName* and *txtPassword*, and an OK button, named *cmdOK*.

```
'USER.CLS
`~~~.Name
Property Get Name()
    Name = frmPass.txtName
End Property

`~~~.Password
Property Get Password()
    Password = frmPass.txtPassword
End Property

`~~~.Show
Sub Show()
    frmPass.Show vbModal
End Sub

Private Sub Class_Terminate()
    `Unload form when object is destroyed
    Unload frmPass
End Sub
```

The form, named *frmPass*, contains only one event procedure to hide the form when the user clicks OK. This lets the object's creator validate the user name and password and quickly redisplay the form if the entered data is incorrect. The data on the form isn't cleared, making it easier for the user to make corrections.

```
`FRMPASS.FRM
Private Sub cmdOK_Click()
    Hide
End Sub
```

To use this object, another application simply creates an instance of the object and then calls the *Show* method. The code below shows an example of how this can be done on a form containing a command button named *cmdSignIn*. Note the syntax for referencing the User class in the project named *PasswordSample*.

```
Private Sub cmdSignIn_Click()
    `Create an instance of the object
    Dim usrSignIn As New PasswordSample.User
    `Loop until user enters Guest password
    Do While usrSignIn.Password <> "Guest"
        `Display dialog box
        usrSignIn.Show
    Loop
End Sub
```

Figure 5-6 shows an example of the User object form.

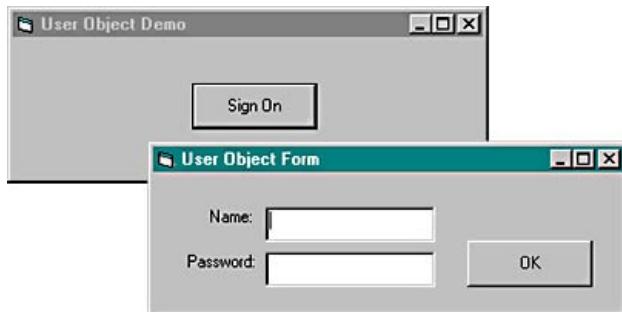


Figure 5-6. Example of an object that displays a form.

To see the importance of managing form creation and destruction, remove the User object's Class_Terminate event procedure, and then create the object from another application, as shown above. When the cmdSignOn_Click event procedure ends, the *usrSignIn* variable loses scope and is destroyed, but the *frmPass* form remains loaded, running in a hidden instance of the object's application.

If the object is defined in an EXE component, you can view this hidden instance in the Windows Task Manager by pressing Ctrl-Alt-Delete. You'll have to end the task from the Task Manager so that you can correct this problem and recompile a corrected version. Don't leave a "bad" version of this object around to cause problems later—hidden instances of object applications can be difficult to detect.

In general, you use the Class_Initialize and Class_Terminate event procedures to manage the creation and deletion of forms. Because this sample was so short, I let the form be created implicitly by whichever property or method was called first. I could just as easily have been explicit by including this Class_Initialize event procedure:

```
'Add to USER.CLS to be explicit
Private Sub Class_Initialize()
    'Create form when object is created
    'by calling Show method
    Show
End Sub
```

NOTE

It's a good idea to *always* be explicit when you create and destroy forms in objects. This will save time down the road when you are debugging.

Event, WithEvents, and RaiseEvent

There are several related Visual Basic statements that let you add your own user-defined events to your objects. I mention this here because one of the best uses for this technique is to provide a way for private forms owned by objects in ActiveX components to immediately react with client applications. An event raised in a private form can trigger an immediate response in the ActiveX object responsible for that form, and that object can in turn raise an event back in the client application. The Books Online documentation topic Adding a Form to the ThingDemo Project provides a good example of this scenario.

Dear John, How Do I... Work with Collections of Objects?

Visual Basic's Collection object lets you combine objects—or items of any data type for that matter—into a set that can be referred to as a single unit. You might think of a Collection object as an extremely flexible array that contains just about anything you want to put in it, including objects and other collections. The most common use for Collection objects, however, is to maintain a collection of an unspecified number of instances of a specific type of object. That's what the following example will demonstrate.

There are many ways to use collections in the Visual Basic objects you create. Some techniques are more "dangerous" than others, and depending on how much you want to protect yourself or other programmers, you can implement collections of objects in simple ways or in somewhat intricate but safe configurations. I recommend that you thoroughly read the Books Online topics covering this subject to gain a solid understanding of Microsoft's suggested approach. I've used an only slightly modified version of Microsoft's suggested techniques, and I think it would be helpful for you to follow their guidelines, too. (Search Books Online for *House of Straw*, *House of Sticks*, and *House of Bricks* to locate these topics.)

The following example presents a simple, straightforward method for implementing collections of user-defined objects within other objects. This method is fairly robust, and the technique can be easily duplicated or modified as desired. We'll create a solar system structure comprising a top-level Star object that contains a Planets collection. Each Planet object within this collection contains a Moons collection, and each Moon object is a very simple object consisting of just a few properties.

SolarSys—A Collections Example

Start up Visual Basic, and open a new Standard EXE project. Set the form's Name property to *SolarSys*, and save this form as SOLARSYS.FRM. Save the project using the name SOLARSYS.VBP. Change the form's Caption property to *Collections Example*, and add a command button named *cmdBuildSolarSystem*. Edit the button's Caption property to read *Build Solar System*. Add the following code to the form, and then save your work:

```
Option Explicit
Public starObject As New Star
Private Sub cmdBuildSolarSystem_Click()
    'Prepare working object references
    Dim planetObject As Planet
    Dim moonObject As Moon
    'Add planets and moons to the solar system
    With starObject.Planets
        'Create first planet
        Set planetObject = .Add("Mercury")
        'Set some of its properties
        With planetObject
            .Diameter = 4880
            .Mass = 3.3E+23
        End With
        'Create second planet
        Set planetObject = .Add("Venus")
        'Set some of its properties
        With planetObject
            .Diameter = 12104
            .Mass = 4.869E+24
        End With
        'Create third planet
        Set planetObject = .Add("Earth")
        'Set some of its properties
        With planetObject
            .Diameter = 12756
            .Mass = 5.9736E+24
        End With
        'Add moons to this planet
        With .Moons
            'Create Earth's moon
            Set moonObject = .Add("Luna")
        End With
    End With
End Sub
```

```
'Set some of its properties
With moonObject
    .Diameter = 3476
    .Mass = 7.35E+22
End With
End With
End With
`Create fourth planet
Set planetObject = .Add("Mars")
`Set some of its properties
With planetObject
    .Diameter = 6794
    .Mass = 6.4219E+23
`Add moons to this planet
With .Moons
    `Create Mar's first moon
    Set moonObject = .Add("Phobos")
    `Set some of its properties
    With moonObject
        .Diameter = 22
        .Mass = 1.08E+16
    End With
    `Create Mar's second moon
    Set moonObject = .Add("Deimos")
    `Set some of its properties
    With moonObject
        .Diameter = 13
        .Mass = 1.8E+15
    End With
End With
End With
End With
`Disable the command button
cmdBuildSolarSystem.Enabled = False
`Display the results
Print "Planet", "Moon", "Diameter (km)", "Mass (kg)"
Print String(100, "-")
For Each planetObject In starObject.Planets
    With planetObject
        Print .Name, , .Diameter, .Mass
    End With
    For Each moonObject In planetObject.Moons
        With moonObject
            Print , .Name, .Diameter, .Mass
        End With
    Next moonObject
Next planetObject
`Directly access some specific properties
Print
Print "The Earth's moon has a diameter of ";
Print starObject.Planets("Earth").Moons("Luna").Diameter;
Print " kilometers."
Print "Mars has";
Print starObject.Planets("Mars").Moons.Count;
Print " moons."
End Sub
```

Figure 5-7 shows the SolarSys form during development.

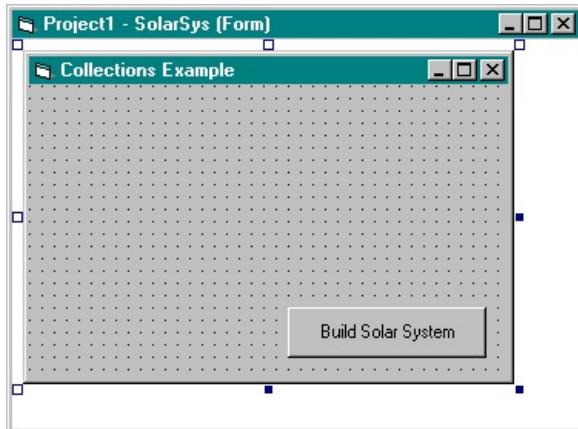


Figure 5-7. The SolarSys form, used to demonstrate nested collections of objects.

We'll refer back to this code in a minute, but first let's define the objects and collections of objects that this code creates and manipulates.

The Star Class

From the Project menu, choose Add Class Module, and double-click the Class Module icon in the Add Class Module dialog box. Set the class module's Name property to *Star*, add the following code , and then save the module as STAR.CLS.

```
Option Explicit

Public Name As String

Private mPlanets As New Planets

Public Property Get Planets() As Planets
    Set Planets = mPlanets
End Property
```

The Star object declares and contains a collection of planets. We'll take a look at the Planets class in a minute, but for now be aware that the Planets class defines a collection of Planet objects, which are in turn defined in their own class.

Notice that the Planets collection is declared private, as *mPlanets*, so it's not directly accessible by the external world. Instead, the Property Get Planets procedure provides read-only access to the collection. Since there's no corresponding Property Let procedure, external procedures are prevented from reassigning unexpected objects to this reference. We need the read-only access to the Star's Planets collection in order to access the individual Planet objects it contains.

The Planets Class

All the planets in the solar system will be maintained as individual Planet objects in a Planets collection. Notice that I named this collection the same as the objects in it, except for the addition of the natural-sounding "s" to make it plural. This is the standard technique used throughout all object collections. Add a new class module to your project, name it Planets, add the following code to it, and save it as PLANETS.CLS.

```
Option Explicit
Private mcolPlanets As New Collection
Public Function Add(strName As String) As Planet
    Dim PlanetNew As New Planet
    PlanetNew.Name = strName
    mcolPlanets.Add PlanetNew, strName
    Set Add = PlanetNew
End Function

Public Sub Delete(strName As String)
```

```

mcolPlanets.Remove strName
End Sub

Public Function Count() As Long
    Count = mcolPlanets.Count
End Function

Public Function Item(strName As String) As Planet
    Set Item = mcolPlanets.Item(strName)
End Function

Public Function NewEnum() As IUnknown
    Set NewEnum = mcolPlanets._NewEnum
End Function

```

When I first started creating collections of objects, I was tempted to combine each object and its collection into one class module. This works, but there are lots of hidden problems and unforeseen gotchas that are avoided by making the collection an object separate from the objects it contains. That's why this example application includes both a Planet and a Planets class. They work hand in hand to create a fairly robust collection of planets. Likewise, as you'll see in a minute, the Moon and Moons classes work together to create a collection of moons.

The Planets collection class encapsulates and handles the creation and manipulation of individual Planet objects. The actual Collection object, *mcolPlanets*, is declared private to this class module to encapsulate and protect it from the outside world. The Add, Delete, Count, Item, and NewEnum collection methods provide the interface used to interact with the collection of planets. I've used some special tricks you should know about, so let's go over these procedures to explain them.

The Add method is used to create a new planet in the collection. For this example I've used the name of each planet, as passed to the Add method, both to set the planet's Name property and as the key by which the planet is to be stored and retrieved. The key can be something different from the name, as it is in the collections example described in Visual Basic's online documentation. Each element stored in the collection must have a unique string key, and letting the collection class create and automatically maintain its own set of keys is one way to guarantee each key's uniqueness. The technique I've chosen is slightly simpler to implement and provides a natural way to retrieve each planet from the collection by name. Note, however, that an error will be generated if an attempt is made to add two planets of the same name. You may want to add code to the Add method to trap errors caused by repeated planet names. To keep things simple I chose to leave this step up to the reader.

The Count procedure returns the number of planets in the Planets collection, and the Delete procedure removes a planet from the Planets collection. Again, you may wish to add error-trapping code to prevent errors resulting from attempts to delete nonexistent planets from the collection.

When given a specific planet name, the Item procedure returns that planet object from the collection. Here's the first important trick I want to mention. If you set the Item procedure as the default property of the class, then the syntax to access a specific planet is more natural. For example, you can access the planet Earth object using the Item property like this: *starObject.Planets.Item("Earth")*; or, if the Item procedure is set as the default procedure, the syntax is shortened to this: *starObject.Planets ("Earth")*. This shortened syntax is much more in keeping with the standard object syntax you'll see in many other Windows application objects.

NOTE

To set a procedure as the default, select Procedure Attributes from the Tools menu. Select the procedure in the Name drop-down list, click the Advanced button to see more options in the dialog box, and select (Default) from the Procedure ID drop-down list. Only one procedure in each class can be selected as the default.

The second important trick enables you to access objects in a collection class using the For Each looping

construct. This one's a little obscure, so follow along closely. The first step is to add the special NewEnum IUnknown procedure to your collection class, as shown in the code for the Planets class. The square brackets are required, as is the underscore for the hidden enumeration property of Visual Basic's Collection object. The second step is to hide and enable the NewEnum function in your class, using the steps detailed in the following note:

NOTE

To hide and enable the special NewEnum procedure in your class, choose Procedure Attributes from the Tools menu. Select the NewEnum procedure in the Name drop-down list, and click the Advanced button to see more options. Click to put a check mark in the Hide This Member check box, and type the special code number of minus four (-4) in the Procedure ID list. This enables your collection for access using the For Each looping construct.

The Planet Class

Add another class module to the SolarSys project, set its Name property to *Planet*, and save the module as PLANET.CLS. This class defines the individual Planet objects that are collected within the Planets collection. Add the following code to this class module.

```
Option Explicit

Public Name As String
Public Diameter As Long
Public Mass As Single

Private mMoons As New Moons

Public Property Get Moons() As Moons
    Set Moons = mMoons
End Property
```

Each Planet object has a few simple properties (Name, Diameter, and Mass) and a Moons collection containing anywhere from zero to several Moon objects. Once again, the actual collection is declared private so that external code won't be able to set unexpected references into the Moons object reference. The read-only Property Get Moons procedure allows you to reference the moons for this planet.

The Moons Class

The Moons class is a collection class very similar in structure to the Planets collection class. Add another class module to your SolarSys project, set its Name property to *Moons*, add the following code to it, and save it as MOONS.CLS.

```
Option Explicit

Private mcolMoons As New Collection

Public Function Add(strName As String) As Moon
    Dim MoonNew As New Moon
    MoonNew.Name = strName
    mcolMoons.Add MoonNew, strName
    Set Add = MoonNew
End Function

Public Sub Delete(strName As String)
    mcolMoons.Remove strName
End Sub

Public Function Count() As Long
```

```

    Count = mcolMoons.Count
End Function

Public Function Item(strName As String) As Moon
    Set Item = mcolMoons.Item(strName)
End Function

Public Function NewEnum() As IUnknown
    Set NewEnum = mcolMoons.[_NewEnum]
End Function

```

Notice the striking similarity to the Planets collection class. The procedures are nearly identical; only the referenced object names have been changed to protect the innocent.

The Moon Class

This class is ultrasimple. Each Moon object will have Name, Diameter, and Mass properties, and that's all. Add another class module to the project, set its Name property to *Moon*, save it as MOON.CLS, and add the following code to define each Moon object:

```

Option Explicit

Public Name As String
Public Diameter As Long
Public Mass As Single

```

How the Nested Collections Work

Let's take a closer look at the top-level code in the SolarSys form. When the Build Solar System command button is clicked, a new Star object, named *starObject*, is created. Since we aren't concerned with creating more than one Star object, we won't bother to set its Name property. (If we were creating a program to catalog all star systems in a collection of science fiction stories, we might want to create multiple Star objects, perhaps in a Stars collection, and unique names for each would probably be appropriate.)

To simplify the code, I added only the first four of our solar system's planets and their moons. Feel free to build up the whole solar system if you feel so inclined! The following code adds the first Planet object named Mercury. Mercury has no moons, so we won't add any of these objects to this planet:

```

`Add planets and moons to the solar system
With starObject.Planets
    `Create first planet
    Set planetObject = .Add("Mercury")
    `Set some of its properties
    With planetObject
        .Diameter = 4880
        .Mass = 3.3E+23
    End With

```

I've used the *With* statement to simplify the object references because we're going several layers deep here. However, we could directly access the planet Mercury object's properties by explicitly using dot notation to get to them. For example, here's what the line to set Mercury's Diameter property would look like using the complete dot notation:

```
starObject.Planets("Mercury").Diameter = 4880
```

Paraphrased, this says to set the value 4880 as the Diameter property of the planet Mercury object in the Planets collection maintained by the Star object named *starObject*. Whew! You can see why the *With* statement is useful. In addition to being easier for humans to comprehend, *With* blocks reduce the number of de-referencing steps required to access each object and its properties, thus speeding things up.

The next block of code repeats the process, adding the second Planet object, named Venus, which also lacks Moon objects. Let's skip ahead then to the third planet from the sun, named Earth, which does have

one moon. Here's the code to add this planet and its moon to the nested collections:

```
'Create third planet
Set planetObject = .Add("Earth")
' Set some of its properties
With planetObject
    .Diameter = 12756
    .Mass = 5.9736E+24
    'Add moons to this planet
    With .Moons
        'Create Earth's moon
        Set moonObject = .Add("Luna")
        ' Set some of its properties
        With moonObject
            .Diameter = 3476
            .Mass = 7.35E+22
        End With
    End With
End With
```

Once again, notice how the nested With constructs help in referencing the structure of nested object collections.

Once this simplified model of our solar system is built up as a set of collections of objects that are contained within other collections of objects, the next few lines of code access this structure to display the names of all the nested objects:

```
'Display the results
Print "Planet", "Moon", "Diameter (km)", "Mass (kg)"
Print String(100, "-")
For Each planetObject In starObject.Planets
    With planetObject
        Print .Name, , .Diameter, .Mass
    End With
    For Each moonObject In planetObject.Moons
        With moonObject
            Print , .Name, .Diameter, .Mass
        End With
    Next moonObject
Next planetObject
```

We've enabled the For Each looping construct, described earlier in this chapter, which greatly aids in the process of referencing the various properties of the planets and moons. The code above is fairly straightforward and easy to understand. For purposes of comparison, in the next few lines of the code I've used some direct property referencing using the full dot notation to access and print some properties:

```
'Directly access some specific properties
Print
Print "The Earth's moon has a diameter of ";
Print starObject.Planets("Earth").Moons("Luna").Diameter;
Print " kilometers."
Print "Mars has";
Print starObject.Planets("Mars").Moons.Count;
Print " moons."
```

Figure 5-8 shows the results as displayed on the SolarSys form after the command button is clicked.

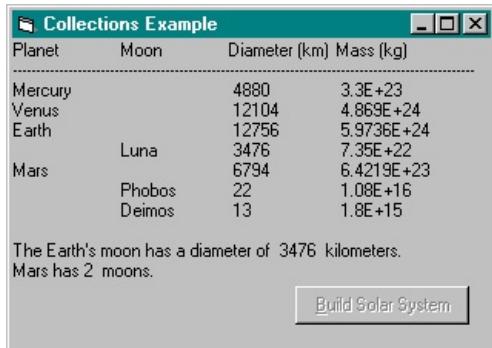


Figure 5-8. The simplified solar system's Planets and Moons collections.

There are a lot of ways you could modify the technique presented here to suit your needs. As mentioned, you might want to add some code to prevent accidental attempts to add multiple objects of the same name and to handle missing objects in a more robust fashion. This method serves as a strong conceptual foundation on which to build without getting too lost in all the possible complexities. For many programming situations, this method can provide everything you need.

Dear John, How Do I... Understand and Use Polymorphism?

This new OOP world is just full of new terminology and concepts! Let's take a look at *polymorphism*—what it means and how it can help you in your development efforts.

Simply put, polymorphism means that multiple objects will provide the same interface elements. For example, you might create a set of objects for your business that all provide a "standard" set of file functions. All of these objects would provide related methods and properties with predictably identical names, such as `FileName`, `Read`, and `Write`. This would simplify, standardize, and coordinate these objects, making them easier to use. It also would make them more predictable and consistent.

Visual Basic handles polymorphism a little differently than C++ does, for instance. Historically, in most OOP languages, the methods and properties of one object are inherited by a new object. The new object can then optionally modify the inherited interface elements or expose them "as is" to client applications.

Visual Basic, on the other hand, doesn't use inheritance to provide polymorphism. Instead, Visual Basic lets us create an interface, which as mentioned is a set of related property and method definitions, contained in an *abstract class*. An abstract class is simply a class module containing empty, ghost-like definition shells of methods and properties. This special class is then declared in class modules that agree to implement these interface items using the `Implements` keyword.

I like to think of the abstract class as a kind of contract. Multiple class modules can each agree to abide by this shared contract by using the `Implements` keyword. A client application can then declare instances of these objects, including an object variable defined by the special abstract class. By accessing each object's implemented interface elements through the abstract class-defined object, it's guaranteed that each object will handle the agreed-upon method or property in its own way.

The best way to master these concepts is to try them and work with them. The example provided in the Books Online documentation provides a great way to experiment and learn how they work. Search Books Online for *Tyrannosaurus*, or if you're in a hurry or don't feel like typing that much, search for *Flea*. The explanation of how a *Tyrannosaurus* object and a *Flea* object each agree to implement a *Bite* method through Visual Basic's form of polymorphism is a great way to learn—and to remember—how this all works.

Dear John, How Do I... Use Friend Methods?

Visual Basic 4 first introduced us to class module-defined objects and their methods and properties. Users soon noted a minor shortcoming in the way a component exposes properties and methods of its set of class modules to client applications. There was no way for a class to expose methods to other objects within the same component without also exposing these methods to the whole world.

Visual Basic solved this problem by providing the Friend keyword, which modifies a method so that it is exposed and visible to all classes within the same component but not to external, client applications.

Components often provide a related set of objects that have a lot in common with each other. The Friend methods let these objects communicate in a structured way among themselves, without requiring their intercommunication methods to be exposed to other applications.

Chapter Six

ActiveX Controls

Many analysts credit the early success of Visual Basic to custom controls. It's easy to see why—custom controls mean that you aren't limited to one vendor for all your programming tools. You can choose the best tools from a free and competitive marketplace.

With Visual Basic, you can create your own custom controls (called *ActiveX controls*). ActiveX controls written in Visual Basic look and behave just like controls written in C. In fact, you can distribute them to your C-programming friends for use in Microsoft Visual C++.

ActiveX controls can be used just about everywhere: on Web pages viewed by Internet Explorer; in Microsoft Excel and Microsoft Word documents; in Microsoft Access and Visual FoxPro database applications; and, of course, in Visual Basic, Microsoft Visual C++, and Microsoft Visual J++.

This chapter shows you how to create ActiveX controls and describes the programming issues that are unique to these controls. The sample code presented in this chapter is available on the companion CD-ROM.

Dear John, How Do I... Create an ActiveX Control?

The ActiveX Control Interface Wizard lets you base new controls on existing controls. The code generated by the wizard can be difficult to understand, however, so this section shows you how to create an ActiveX control manually. Once you understand the parts of an ActiveX control, it's easier to use the ActiveX Control Interface Wizard.

ActiveX Control Design Steps

To create an ActiveX control, follow these steps:

Create a new ActiveX control project.

1. In the UserControl window, draw the visual interface of the control using graphics methods and controls from the Toolbox, just as you would for a form.
2. In the code window, add an event procedure to resize the visible aspects of the control when the user resizes the control on a form.
3. Add the properties, methods, and events that your control will expose to applications using it. These elements define the interface of your control.
4. Write code that implements your control's functionality. This code determines the behavior of your control.
5. Add a new Standard EXE project within Visual Basic to provide a way to debug the control.
6. From the File menu, choose Make to compile the control into an OCX file.

The following text discusses the first six steps in greater detail while providing instructions on how to build a sample control named Blinker (BLINKER.VBP). The Blinker control is used to flash controls or windows on screen for emphasis. Debugging and compiling the control are discussed in the two sections that follow, "[Dear John, How Do I... Debug a Control?](#)" and "[Dear John, How Do I... Compile and Register a Control?](#)"

Creating the ActiveX Control Project

To create an ActiveX Control project, follow these steps:

1. From the File menu, choose New Project to display the New Project dialog box.
2. Double-click the ActiveX Control icon. Visual Basic creates a new project and displays the UserControl window.
3. From the Project menu, choose Project Properties to display the Project Properties dialog box.
4. In the Project Name text box, type *VB6WkSamp*. In the Project Description text box, type *VB6 Workshop Blinker Sample Control* and then click OK. The project name is used to name the OCX file; the project description is displayed in the Components dialog box.
5. In the UserControl Properties window, set the Name property to *Blinker*. This is the class name for the control. It will be used when naming instances of the control as they are drawn on a form: *Blinker1*, *Blinker2*, and so on.
6. Choose Save Project from the File menu, and save the Blinker control as *BLINKER.CTL* and the project as *BLINKER.VBP*.

Drawing the Interface

The Blinker control flashes other controls or windows in your application in order to draw the user's

attention. This isn't spectacularly hard to do—which is one reason it makes a good sample. The Blinker control uses an ActiveX UpDown control, an intrinsic TextBox control, and an intrinsic Timer control, as shown in Figure 6-1.

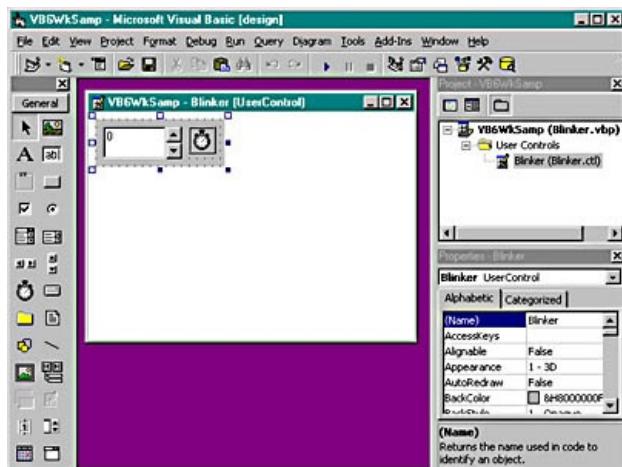


Figure 6-1. The visual interface of the Blinker control, which contains one ActiveX control and two intrinsic controls.

To create the visual interface of the Blinker control, follow these steps:

1. From the Project menu, choose Components to display the Components dialog box.
2. On the Controls tab, check the Microsoft Windows Common Controls-2 6.0 (MSCOMCT2.OCX) checkbox and click OK. Visual Basic adds the UpDown control and other common Windows controls to the Toolbox.
3. In the UserControl window, draw a text box. In the Properties window, set its Name property to *txtRate* and its Text property to *0*.
4. Draw an UpDown control next to the text box, and set its Name property to *updnRate*.
5. Draw a Timer control, and set its Name property to *tmrBlink*.

Resizing the Control

You can be sloppy when drawing the Blinker interface because the size and position of the visible controls must be handled in code anyway. When users draw the Blinker control on a form, they can click and drag the control to make it the size they want. Resizing triggers the *UserControl_Resize* event procedure, which positions and sizes the visible interface of the control appropriately.

The following code handles resizing the Blinker control:

```
'Code for control's visual interface
Private Sub UserControl_Resize()
    'Be sure visible controls are
    'positioned correctly within the
    'UserControl window
    updnRate.Top = 0
    txtRate.Top = 0
    txtRate.Left = 0
    'Resize visible controls
    'when user control is resized
    updnRate.Height = Height
    txtRate.Height = Height
    'Adjust UpDown control's width up
    'to a maximum of 240 twips
    If Width > 480 Then
        updnRate.Width = 240
    Else
```

```

        updnRate.Width = Width \ 2
End If
`Set width of text box
txtRate.Width = Width - updnRate.Width
`Move UpDown control to right edge of
`text box
updnRate.Left = txtRate.Width
End Sub

```

As you can see, I got a little fancy with the resize code for the UpDown control. Rather than fix it at 240 twips, I let it scale down if the user makes the Blinker control less than 480 twips. Your resize code can be as simple or as complicated as you want.

Adding Properties, Methods, and Events

In addition to the standard size, position, and visibility properties provided with all controls, the Blinker control has a TargetObject property, which specifies the object that will blink, and an Interval property, which specifies how many times per second the object should blink. The Blinker control also includes a Blinded event, which occurs after the object blinks, and a Blink method, which sets the TargetObject and Interval properties.

ActiveX control properties, methods, and events are defined in the same way as class properties, methods, and events. The following code shows the TargetObject, Interval, Blinded, and Blink members of the control:

```

Option Explicit

`Windows API to flash a window
Private Declare Function FlashWindow _
Lib "user32" (
    ByVal hwnd As Long,
    ByVal bInvert As Long
) As Long

`Blinded event definition
Public Event Blinded()

`Internal variables
Private mobjTarget As Object
Private mlngForeground As Long
Private mlngBackground As Long
Private mblnInitialized As Boolean

`Public error constants
Public Enum BlinkerErrors
    blkCantBlink = 4001
    blkObjNotFound = 4002
End Enum

`Code for control's properties and methods
`~~~.TargetObject
Public Property Set TargetObject(Setting As Object)
    If TypeName(Setting) = "Nothing" Then Exit Property
    `Set internal object variable
    Set mobjTarget = Setting
End Property

Public Property Get TargetObject() As Object
    Set TargetObject = mobjTarget
End Property

`~~~.Interval
Public Property Let Interval(Setting As Integer)

```

```

`Set UpDown control--updates TextBox and
`Timer controls as well
updnRate.Value = Setting
End Property

Public Property Get Interval() As Integer
    Interval = updnRate.Value
End Property

`~~~.Blink
Sub Blink(TargetObject As Object, Interval As Integer)
    `Delegate to TargetObject and Interval properties
    Set Me.TargetObject = TargetObject
    Me.Interval = Interval
End Sub

```

The TargetObject Property Set procedure sets the target object that should blink by assigning an internal object variable. The Interval property merely sets or returns the value of the UpDown control—this changes the value in the text box and thereby changes the Timer control's Interval property. The Blanked event is defined here but is triggered from the Timer event. Next I'll show you the event procedures for the UpDown, Timer, and TextBox controls—where the real work is done.

Programming the Control's Behavior

So far, the Blinker control looks nice, but doesn't do anything. The control's behavior—flashing a control or window—is determined by code in the UpDown control's Change event procedure, the timer's Timer event procedure, and the text box's Change event procedure, as shown here:

```

`Code for control's behavior
Private Sub updnRate_Change()
    `Update the text box control
    txtRate.Text = updnRate.Value
End Sub

Private Sub tmrBlink_Timer()
    On Error GoTo errTimer
    `Counter to alternate blink
    Static blnOdd As Boolean
    blnOdd = Not blnOdd
    `If the object is a form, use FlashWindow API
    If TypeOf mobjTarget Is Form Then
        FlashWindow mobjTarget.hwnd, CLng(blnOdd)
    `If it's a control, swap the colors
    ElseIf TypeOf mobjTarget Is Control Then
        If Not mblnInitialized Then
            mlngForeground = mobjTarget.ForeColor
            mlngBackground = mobjTarget.BackColor
            mblnInitialized = True
        End If
        If blnOdd Then
            mobjTarget.ForeColor = mlngBackground
            mobjTarget.BackColor = mlngForeground
        Else
            mobjTarget.ForeColor = mlngForeground
            mobjTarget.BackColor = mlngBackground
        End If
    Else
        Set mobjTarget = Nothing
        GoTo errTimer
    End If
    `Trigger the Blanked event
    RaiseEvent Blanked
    Exit Sub

```

```

errTimer:
    If TypeName(mobjTarget) = "Nothing" Then
        Err.Raise blkObjNotFound, "Blinker control",
            "Target object is not valid for use with this control."
    Else
        Err.Raise blkCantBlink, "Blinker control",
            "Object can't blink."
    End If
End Sub

Private Sub txtRate_Change()
    'Set Timer control's Interval property
    'to match value in text box
    If txtRate = 0 Then
        tmrBlink.Interval = 0
        tmrBlink.Enabled = False
        mblnInitialized = False
        'If blinking is turned off, be sure object
        'is returned to its original state
        If TypeOf mobjTarget Is Form Then
            FlashWindow mobjTarget.hwnd, CLng(False)
        ElseIf TypeOf mobjTarget Is Control Then
            mobjTarget.ForeColor = mlngForeground
            mobjTarget.BackColor = mlngBackground
        End If
    Else
        tmrBlink.Enabled = True
        tmrBlink.Interval = 1000 \ txtRate
    End If
End Sub

```

The UpDown control's Change event procedure simply copies the value of the UpDown control to the text box. The timer's Timer event procedure handles the flashing by using the FlashWindow API function for forms or by swapping the ForeColor and BackColor properties for visible controls. Using the FlashWindow API function is more effective than swapping the ForeColor and BackColor properties. The text box's Change event procedure handles the blinking rate.

SEE ALSO

- Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)," for more information about the FlashWindow API function

Dear John, How Do I... Debug a Control?

Once you've created an ActiveX control, you'll want to debug it. By default, Visual Basic debugs ActiveX controls by displaying them in Internet Explorer when you select Start from the Run menu, as shown in Figure 6-2. Though this lets you see what the control looks like at runtime, it isn't a very good way to debug a control since you can't easily test the control's properties and methods.

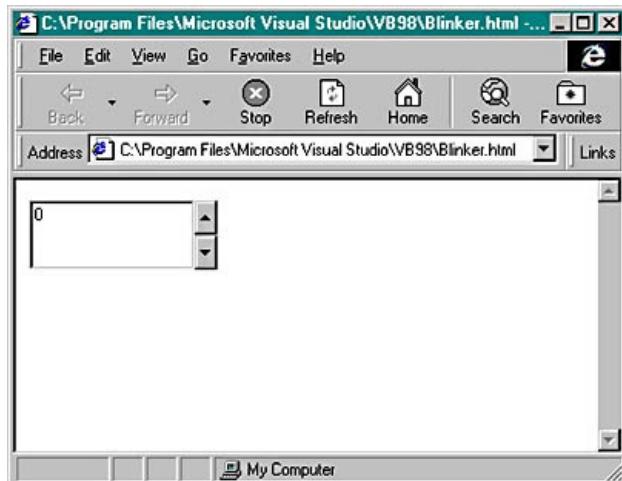


Figure 6-2. If you create an ActiveX control project and run it, Visual Basic will display the control in Internet Explorer.

To thoroughly test an ActiveX control, create a project group that contains a test project and the ActiveX control project, following these steps:

1. If the ActiveX control project is currently loaded, save it by choosing Save Project from the File menu.
2. From the File menu, choose New Project to display the New Project dialog box.
3. Double-click the Standard EXE icon. Visual Basic creates a new project with a single form that you can use to test your ActiveX control.
4. From the File menu, choose Add Project to display the Add Project dialog box.
5. Select the ActiveX control project from the Existing tab on the Add Project dialog box, and then click Open.
6. Close the UserControl window if it is open. Once the window is closed, Visual Basic activates the control's Toolbox icon.
7. Click the ActiveX control's Toolbox icon, and draw the control on your test project's form.
8. Write code in the test project to access the control's properties and methods.
9. Run the test project.

The following code tests the Blinker control named *blnkTest* on a form that also contains a text box named *txtStuff*:

```

Option Explicit

Private Sub Form_Load()
    'Set the object to blink
    blnkTest.Blink txtStuff, 1
End Sub

Private Sub Form_Click()

```

```

`Stop the blinker
blnkTest.Interval = 0
End Sub

Private Sub blnkTest_Blinked()
    Static intCount As Integer
    intCount = intCount + 1
    Caption = "Blinked " & intCount & " times"
End Sub

```

Figure 6-3 shows an example of the Blinker control being tested.



Figure 6-3. Testing the Blinker control.

While debugging a control in-process, you can trace execution from the test project into the ActiveX control's code. You can test the code in your ActiveX control by stepping through execution, setting breakpoints, and using watches, as shown in Figure 6-4.

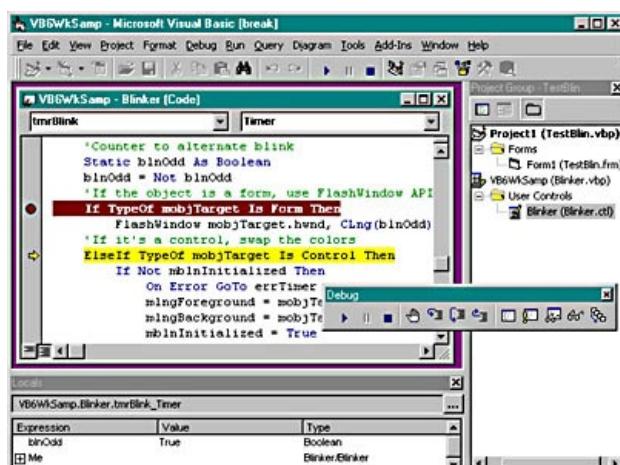


Figure 6-4. The Blinker control being debugged in-process.

Some of the code in the control runs before you run your test project. To see this, set a breakpoint in the `UserControl_Resize` event procedure before you draw the control. Visual Basic will jump to the breakpoint when you release the mouse button after drawing the control.

You can modify the code in your ActiveX control at any time, but if you open the `UserControl` window, the control is disabled on your test form and in the Toolbox. Close the `UserControl` window to reenable the control.

If you draw a control on your test form and then add design-time properties to the control, Visual Basic disables the control on the form. Terminating code running in the control has the same effect. You can reactivate the control by running the test project.

NOTE

When the ActiveX control is initialized at runtime, properties from other controls on the form might or might not be available to code in the ActiveX control. Referring to a control that was drawn on the form before the ActiveX control was drawn may cause your application to stop responding. This appears to be a bug in Visual Basic.

Dear John, How Do I... Compile and Register a Control?

Compiling the control is straightforward, but before you compile you should decide whether the control should be compiled to native code or pseudocode (p-code). Native code executes faster, but p-code results in a smaller OCX file.

In the case of the Blinker control, the differences are minimal. Because the control relies on Timer events, execution speed isn't much of an issue and the difference between a native code OCX file and a p-code OCX file is only 5 kilobytes (KB).

Of course, even 5 KB can make a difference when you are downloading a control across the Internet. It's generally a good idea to use p-code for any controls you plan to use in Internet applications.

To set the compiler options and compile your control, follow these steps:

1. Select the control project in the Project Explorer window. From the File menu, choose the Make option for your project—for example, Make Blinker.OCX—to display the Make Project dialog box.
2. Click Options to display the Project Properties dialog box.
3. Click on the Compile tab, and then select the compilation method and any optimization options. Click OK.
4. In the Make Project dialog box, select the location in which to save the control, and then click OK to compile the control.

Once the control is compiled, Visual Basic automatically registers it on your machine. Registering the control adds it to the Components dialog box, as shown in Figure 6-5.

If your control is part of an application that you are distributing, the Package and Deployment Wizard will handle registering your control when the application is installed on other machines. If you are distributing your control without a setup program, you'll need to install the Visual Basic runtime DLL, MSCOMCT2.OCX, and BLINKER.OCX and then use the REGOCX32.EXE utility to register MSCOMCT2.OCX and BLINKER.OCX. REGOCX32.EXE is found in the \COMMON\TOOLS\VB\REGUTILS directory on the Visual Studio CD-ROM. The following command line registers the Blinker ActiveX control:

```
REGOCX32 .EXE BLINKER .OCX
```

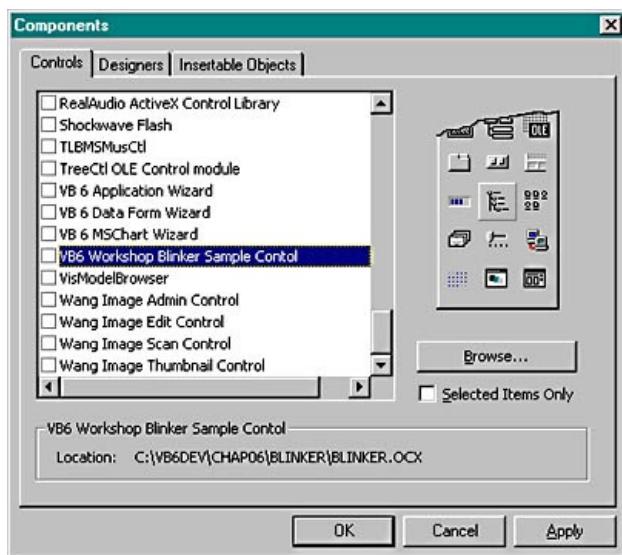


Figure 6-5. The Components dialog box, which lists all registered controls.

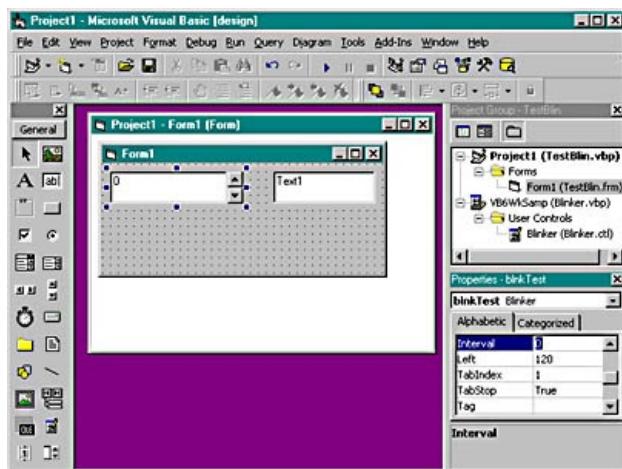
Dear John, How Do I... Create a Design-Time Property?

The TargetObject and Interval properties of the Blinker control can be set at runtime. To create a property that can be set from the Properties window at design time, you need to use the PropertyBag object in the ReadProperties and WriteProperties events, as shown below:

```
'Make Interval a design-time property
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    updnRate.Value = PropBag.ReadProperty("Interval", 0)
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "Interval", updnRate.Value, 0
End Sub
```

This code adds the Interval property to the Blinker control's list of properties in the Visual Basic Properties window, as shown in Figure 6-6.



You need to add a PropertyChanged statement to the control's Interval Property Let procedure so that Visual Basic will save changes to the property at design time. The following code shows the modified procedure:

```
'~~~.Interval
Public Property Let Interval(Setting As Integer)
    'Set UpDown control--updates TextBox and
    'Timer controls as well
    updnRate.Value = Setting
    'Update design-time setting
    PropertyChanged "Interval"
End Property
```

Finally, you need to ensure that design-time changes to the Interval property don't trigger the Timer event. You can tell whether a control is in design mode or user mode by checking the UserControl object's Ambient.UserMode property. UserMode is *False* during design time and *True* at runtime. The following changes to the TextBox's Change event procedure prevents errors from occurring at design time when Interval is set to a nonzero value:

```
Private Sub txtRate_Change()
    'Exit if in design mode
    If Not UserControl.Ambient.UserMode Then Exit Sub
    'Set Timer control's Interval property
    'to match value in text box
    If txtRate = 0 Then
        tmrBlink.Interval = 0
        tmrBlink.Enabled = False
        mblnInitialized = False
    End If
End Sub
```

```

`If blinking is turned off, be sure object
`is returned to its original state
If TypeOf mobjTarget Is Form Then
    FlashWindow mobjTarget.hwnd, CLng(False)
ElseIf TypeOf mobjTarget Is Control Then
    mobjTarget.ForeColor = mlngForeground
    mobjTarget.BackColor = mlngBackground
End If
Else
    tmrBlink.Enabled = True
    tmrBlink.Interval = 1000 \ txtRate
End If
End Sub

```

NOTE

The UserMode property is not available within the control's Initialize event.

Making the TargetObject property available at design time is tricky. Since Visual Basic Properties windows can't display objects, you need to create a new property that accepts a string that, in turn, sets the TargetObject property. The TargetString property shown here can appear in the Properties window:

```

`~~~.TargetString
Public Property Let TargetString(Setting As String)
    If UserControl.Parent.Name = Setting Then
        Set TargetObject = UserControl.Parent
    ElseIf Setting <> "" Then
        Set TargetObject = UserControl.Parent.Controls(Setting)
    End If
End Property

Public Property Get TargetString() As String
    If TypeName(mobjTarget) <> "Nothing" Then
        TargetString = mobjTarget.Name
    Else
        TargetString = ""
    End If
End Property

```

You also need to add a PropertyChanged statement to the control's TargetObject property, as shown in the following code:

```

`~~~.TargetObject
Public Property Set TargetObject(Setting As Object)
    If TypeName(Setting) = "Nothing" Then Exit Property
    `Set internal object variable
    Set mobjTarget = Setting
    `Property has changed
    PropertyChanged "TargetObject"
End Property

```

To add the TargetString property to the Properties window, edit the control's ReadProperties and WriteProperties event procedures as follows:

```

`Get design-time settings
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    updnRate.Value = PropBag.ReadProperty("Interval", 0)
    TargetString = PropBag.ReadProperty("TargetString", "")
End Sub

`Save design-time settings

```

Microsoft® Visual Basic® 6.0 Developer's Workshop

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "Interval", updnRate.Value, 0
    PropBag.WriteProperty "TargetString", TargetString, ""
End Sub
```

Dear John, How Do I... Display a Property Pages Dialog Box?

Property pages let you set the design-time properties of an ActiveX control from a tabbed dialog box rather than from the Visual Basic Properties window. You will want to add property pages to controls that have groups of related properties so that you can access them easily, without having to scroll around in the Properties window.

You can also use property pages to display lists of valid settings that are determined at design time. For example, the Blinker control's property page displays a list of the objects on a form for the TargetString property, as shown in Figure 6-7.

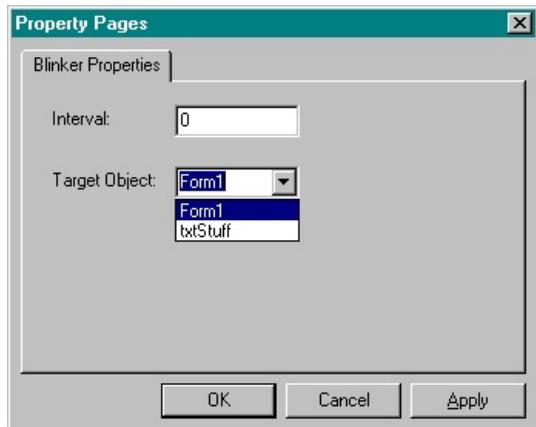


Figure 6-7. The Blinker control's property page.

To add a property page to a control project, select the control project in the Project Explorer window, choose Add Property Page from the Project menu, and double-click the Property Page icon. You can name the property page and specify the text to appear on the property page tab when it is being used by setting the Name and Caption properties, respectively, in the Properties window. To connect a property page to a control, first open the control's UserControl window and select the control. In the Properties window, double-click on the PropertyPages property to display the Connect Property Pages dialog box, as shown in Figure 6-8 below. Check the appropriate property page in the Available Property Pages list, and then click OK.

When you connect a property page to an ActiveX control, Visual Basic adds a (Custom) item to the list of design-time properties displayed for that control. Double-clicking on (Custom) displays the control's property page.

As with the UserControl window, the property page window is in design mode when it is open. You can draw controls on the property page and write code to respond to events just as you would on a form. The property page has a built-in SelectedControls collection that you use to get the instance of the control that the property page refers to. Use the property page's SelectionChanged event procedure to initialize the data you display in the controls on a property page.

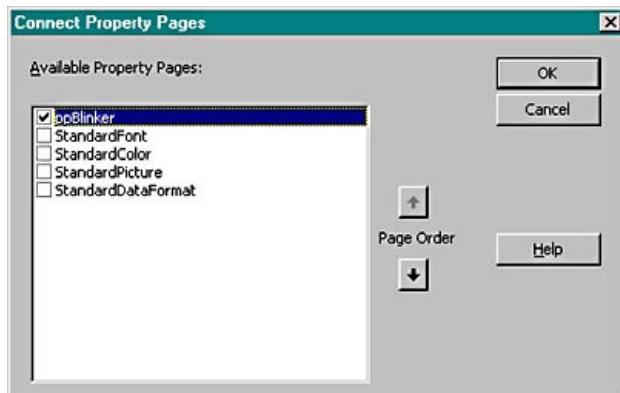


Figure 6-8. Using the Connect Property Pages dialog box to connect a property page to a control.

The following code sets the initial values for the Interval and TargetObject properties, which are displayed on the Blinker property page in the *txtInterval* text box and *cmbTargetString* combo box, respectively.

```

Private Sub PropertyPage_SelectionChanged()
    'Set property page interval to match
    'control's setting
    txtInterval = SelectedControls(0).Interval
    'Build a list of objects for TargetString
    Dim frmParent As Form
    Dim ctrIndex As Control
    Dim strTarget As String
    'Get form the control is on
    Set frmParent = SelectedControls(0).Parent
    strTarget = SelectedControls(0).TargetString
    If strTarget <> "" Then
        'Add current property setting to
        'combo box
        cmbTargetString.List(0) = strTarget
    End If
    If frmParent.Name <> strTarget Then
        'Add form name to combo box
        cmbTargetString.AddItem frmParent.Name
    End If

    'Add each of the controls on the form to
    'combo box
    For Each ctrIndex In frmParent.Controls
        'Exclude Blinker control
        If TypeName(ctrIndex) <> "Blinker" Or _
            ctrIndex.Name = strTarget Then
            cmbTargetString.AddItem ctrIndex.Name
        End If
    Next ctrIndex
    'Display current TargetString setting
    cmbTargetString.ListIndex = 0
End Sub

```

NOTE

The SelectedControls collection is not available within the property page's Initialize event.

Notice that SelectedControls(0) returns the currently selected object—in this case, the Blinker control. I had to add a Parent property to the Blinker control so that this property page could get information about the Blinker control's container. The following code shows the Blinker control's Parent property:

```

'~~~.Parent
Public Property Get Parent() As Object
    Set Parent = UserControl.Parent
End Property

```

You use the property page's ApplyChanges event procedure to write the settings on the property page to the ActiveX control's properties. The following code retrieves the settings from the property page and stores them in the ActiveX control's properties:

```

Private Sub PropertyPage_ApplyChanges()
    'Save settings on the property page
    'in the control's properties
    SelectedControls(0).Interval = txtInterval.Text
    SelectedControls(0).TargetString = _
        cmbTargetString.List _
        (cmbTargetString.ListIndex)
End Sub

```

You need to notify the property page if the user changes any of the settings on the property page. The built-in Changed property tells Visual Basic to apply the property changes when the user clicks OK or Apply. The code below sets the Changed property if either setting on the Blinker control's property page changes.

```
Private Sub txtInterval_Change()
    Changed = True
End Sub

Private Sub cmbTargetString_Change()
    Changed = True
End Sub
```

Property pages can be displayed by right-clicking on a control and choosing Properties from the context menu, by double-clicking on the (Custom) property in the Properties window, or by clicking the ellipsis button in the (Custom) property's setting field in the Properties window.

To add an ellipsis button to a property in the Properties window, follow these steps:

1. Select the Code window of the ActiveX control.
2. From the Tools menu, choose Procedure Attributes to display the Procedure Attributes dialog box.
3. Click Advanced to expand the dialog box.
4. Select the name of the property in the Name drop-down list, and select the property page to display in the Use This Page In Property Browser drop-down list, as shown in Figure 6-9.
5. Click OK.

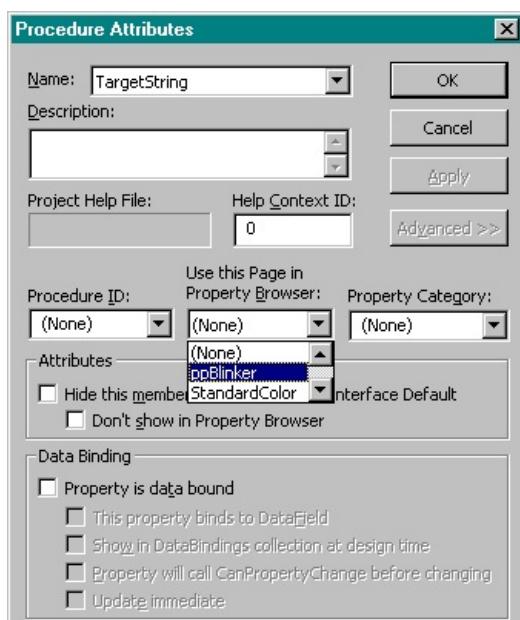


Figure 6-9. Using the Procedure Attributes dialog box to associate a property page with a specific property.

To display the Blinker control's property page, shown in Figure 6-10, click the ellipsis button in the TargetString property.

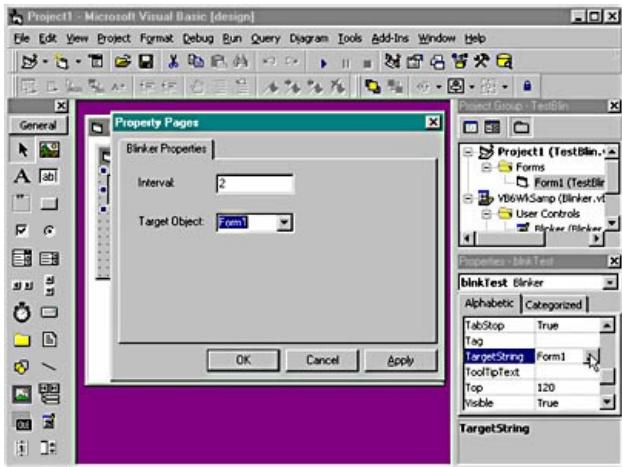


Figure 6-10. The Blinker control's Property Pages dialog box as displayed by clicking the ellipsis button.

When a user displays a Property Pages dialog box by clicking the ellipsis button, the property page should set the focus on the appropriate field. The property page `EditProperty` event procedure shown here sets the focus on the appropriate control when the user clicks the ellipsis button in the `TargetString` or `Interval` property of the Blinker control:

```
Private Sub PropertyPage_EditProperty(PropertyName As String)
    'Set focus on the appropriate control
    Select Case PropertyName
        Case "TargetString"
            cmbTargetString.SetFocus
        Case "Interval"
            txtInterval.SetFocus
        Case Else
    End Select
End Sub
```

Dear John, How Do I... Load a Property Asynchronously?

ActiveX controls used on Web pages may need to load property settings asynchronously. This allows the user's Web browser to display the contents of a Web page while transferring graphics or other large pieces of data in the background.

Figure 6-11 shows an Asynchronous Animation control based on the Animation control found in the Microsoft Windows Common Controls-2 6.0 (MSCOMCT2.OCX).



Figure 6-11. An Asynchronous Animation control playing an Audio Video Interleaved (AVI) file that was downloaded in the background.

The Asynchronous Animation control's AVIFile property takes a string argument that can be a local file specification or a Uniform Resource Locator (URL). The AsyncRead method begins the transfer of the file to the local machine, saving the file in the Windows Temp directory with a name generated by Visual Basic. The following code shows the AVIFile property:

```
Option Explicit

Dim mstrAVISourceFile As String
Dim mstrTempAVIFile As String

`~~~.AVIFile
Property Let AVIFile(Setting As String)
    If UserControl.Ambient.UserMode _
        And Len(Setting) Then
        AsyncRead Setting, vbAsyncTypeFile, "AVIFile"
        mstrAVISourceFile = Setting
    End If
End Property

Property Get AVIFile() As String
    AVIFile = mstrAVISourceFile
End Property
```

The AsyncRead method triggers the AsyncReadComplete event when the transfer is complete. The AsyncReadComplete event procedure is a general handler that runs for all asynchronous events. Use the AsyncProp.PropertyName value in a Select Case statement to execute specific code for each asynchronous property in your control. The AsyncReadComplete event procedure for the sample control opens and plays the AVI file by using an Animation control named *aniControl*, as shown here:

```
`General event handler for all async read complete events
Private Sub UserControl_AsyncReadComplete _
    (AsyncProp As AsyncProperty)
    Select Case AsyncProp.PropertyName
        `For AVIFile property
        Case "AVIFile"
            `Store temporary filename
            mstrTempAVIFile = AsyncProp.Value
            `Open file
            aniControl.Open mstrTempAVIFile
            `Play animation
            aniControl.Play
        Case Else
    End Select
End Sub
```

When the control terminates, be sure to clean up any temporary files you created. Well-behaved Internet applications should not fill up the user's disk with unneeded temporary files. The following code closes the AVI file and then deletes the temporary file containing the data:

```
Private Sub UserControl_Terminate()
    'Delete temporary file
    If Len(mstrTempAVIFile) Then
        aniControl.Close
        Kill mstrTempAVIFile
    End If
End Sub
```

To use the Asynchronous Animation control, simply draw the control on a form and set the AVIFile property from an event procedure. For example, the following code uses an Asynchronous Animation control named *aaniFindFile* to load the Find File animation from the Visual Studio CD-ROM:

```
Private Sub Form_Load()
    aaniFindFile.AVIFile =
        "d:\common\graphics\avis\findfile.avi"
End Sub
```

SEE ALSO

- Chapter 8, "[Creating Internet Components](#)," for information about embedding the Asynchronous Animation control in a Web page.

Dear John, How Do I... Create a Control for Use with a Database?

ActiveX controls can display data from database records by providing a *data binding* to one or more of the control's properties. A data binding is a relationship between the property and a field in a database record or query. For example, the Blinker control created in the preceding sections could have its Interval property bound to a numeric field in a database. To add a binding to a control property, follow these steps:

1. Load the control's project in Visual Basic.
2. From the Tools menu, choose Procedure Attributes to display the Procedure Attributes dialog box.
3. Click on the Advanced button. Visual Basic displays the full Procedure Attributes dialog box, as shown in Figure 6-9.
4. In the Name combo box, select the name of the property to bind.
5. Click on the Property Is Data Bound check box. Select the other check boxes in the Data Binding group as described below and click OK.
 - Select the This Property Binds To DataField check box if the property is the main data-bound property for the control. If the control has only one data-bound property, select this check box. If it has more than one, select the check box for the property that will contain the most important field from the database record. A control can have only one property with this check box selected.
 - Select the Show In DataBindings Collection At Design Time check box to display the property in the Data Bindings dialog box at design time. Leave this check box clear if the property is to be set only at runtime.
 - Select the Property Will Call CanPropertyChanged Before Changing check box if the user or program can change the data displayed in the property. This allows validation of the data entered in the property.
 - Set the Update Immediate check box to make changes to the data field bound to this property the moment that the property changes, rather than waiting for Visual Basic to make the change at the record level.

The Employee control in Figure 6-12 is a simple data-bound control created for use with the NWIND database that ships with Visual Basic.

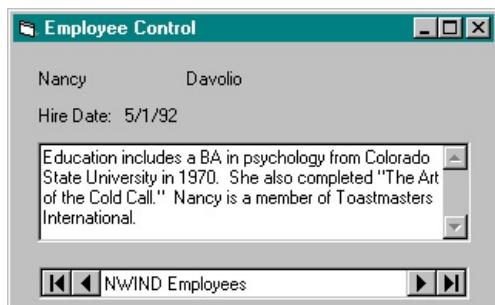


Figure 6-12. The Employee control is designed to display fields in the NWIND database's Employee recordset.

The Employee control (DatCtl.VBP) is composed of several labels and a text box which display the data from the Employees table in the NWIND database. The control's FirstName, LastName, HireDate, and Notes properties are defined by the following code:

`Properties section

```
Public Property Get FirstName()
    FirstName = lblFirstName.Caption
End Property

Public Property Let FirstName(Setting)
    lblFirstName.Caption = Setting
    PropertyChanged FirstName
End Property

Public Property Get LastName() As String
    LastName = lblLastName.Caption
End Property

Public Property Let LastName(Setting As String)
    lblLastName.Caption = Setting
    PropertyChanged LastName
End Property

Public Property Get HireDate() As String
    HireDate = lblHireDate.Caption
End Property

Public Property Let HireDate(Setting As String)
    lblHireDate.Caption = Setting
    PropertyChanged HireDate
End Property

Public Property Get Notes() As String
    Notes = txtNotes.Text
End Property

Public Property Let Notes(Setting As String)
    txtNotes.Text = Setting
    PropertyChanged Notes
End Property
```

Notice that each of the Property Let procedures includes a PropertyChanged statement. According to the Visual Basic documentation, you should include these statements for all properties that can be data-bound, even if they are available only at runtime.

To use the Employee control, follow these steps:

1. Start a new Standard EXE project.
2. Add the DatCtl.VBP project to create a new project group.
3. Draw a Data control on a form, and set the control's DatabaseName property to the NWIND.MDB database installed in the Visual Basic program directory. Set the RecordSource property to Employees.
4. Draw the Employee control on the form, and click the ellipsis button in the DataBindings property. Visual Basic displays the Data Bindings dialog box as shown in Figure 6-13.

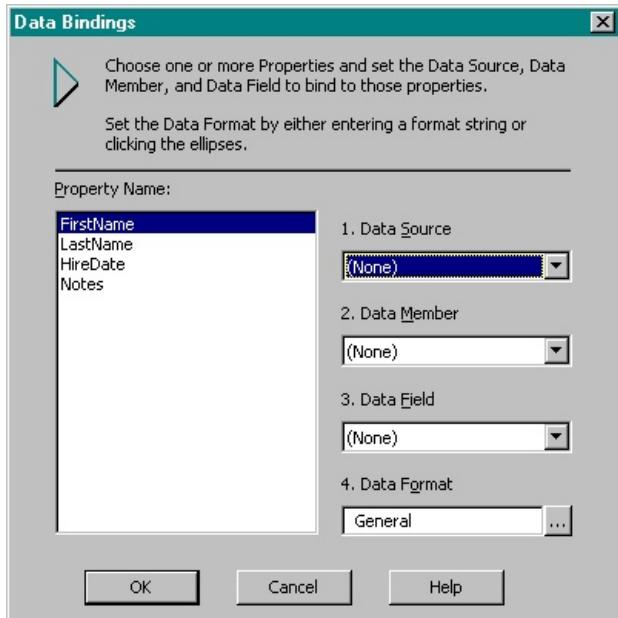


Figure 6-13. Use the Data Bindings dialog box to associate properties with specific fields in a recordset.

5. Select the FirstName property in the Property Name list and select Data1 from the Data Source drop-down list. Select FirstName from the Data Field drop-down list.
6. Repeat step 5 for each of the properties in the Employee control, and then click OK.
7. Run the project. As you use the Data control to move between records, the Employee control displays the information from the NWIND database, as shown in Figure 6-14.

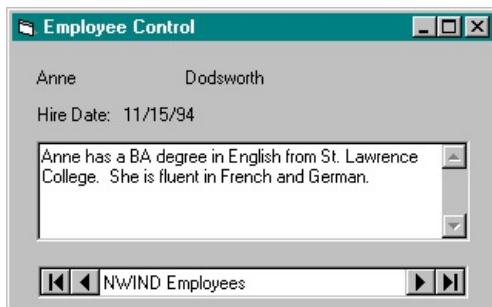


Figure 6-14. Using the Employee control with the Data control to display records from NWIND.MDB.

Dear John, How Do I... Use the DataRepeater Control?

The DataRepeater control is a type of container for the data-bound ActiveX controls you create. To use the DataRepeater control you must already have created a compiled, data-bound control such as the Employee control described in the earlier section, "[Dear John, How Do I... Create a Control for Use with a Database?](#)"

The DataRepeater control lets you replace page views of a database with a scrolling list. It's sort of like the FlexGrid control, but you can include controls within the grid in addition to data. Figure 6-15 demonstrates the differences among a form designed for viewing a database table, a FlexGrid view of the same table, and a view using the DataRepeater control.

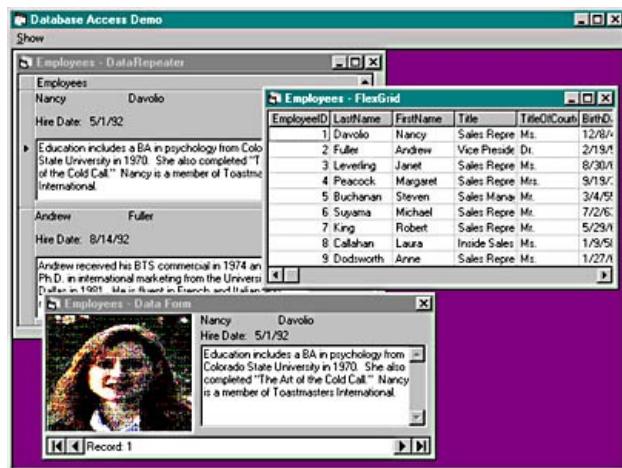


Figure 6-15. A custom form, a FlexGrid control, and a DataRepeater control offer three different ways to view employee records in the NWIND database.

NOTE

The DataRepeater control requires a compatible data source, such as the ADO Data control. The DataRepeater control won't work with the intrinsic Data control.

To use the DataRepeater control, follow these steps:

1. Choose Components from the Project menu. Visual Basic displays the Components dialog box.
2. Select the Microsoft DataRepeater Control 6.0 and the Microsoft ADO Data Control 6.0 check boxes, and then click OK. Visual Basic adds the ADO Data control and the DataRepeater control to the Toolbox.
3. Draw a DataRepeater control on a form.
4. Draw an ADO Data control on the same form as the DataRepeater control.
5. Click the ellipsis button in the ADO Data control's ConnectionString property to display the Property Pages dialog box.
6. Select the Use ODBC Data Source Name option, and then select a data source from the drop-down list (skip steps 7-11) or create a new data source by clicking the New button. (Steps 7-11 explain how to create a new ODBC Microsoft Access data source.)
7. Select the System Data Source (Applies To This Machine Only) option in the Create New Data Source window, and then click Next.
8. Select Microsoft Access Driver (*.MDB) in the Name list, click Next, and then click Finish.

9. Type in a name for the data source in the Data Source Name text box in the ODBC Microsoft Access Setup window.
10. Click the Select button, and choose the Access database for this data source. Click OK to close the ODBC Microsoft Access Setup window.
11. Choose the Use ODBC Data Source Name option, select the data source you created in step 9 from the drop-down list, and then click OK.
12. Click the ellipsis button in the ADO Control's RecordSource property to display the Property Pages dialog box.
13. Select 2 - adCmdTable from the Command Type drop-down list, select a table from the Table Or Stored Procedure Name drop-down list, and then click OK.
14. Select the DataRepeater control and set its DataSource property to the name of the ADO Data control created in step 4.
15. Set the DataRepeater control's RepeatedControlName property to the name of a data-bound ActiveX control.
16. Click the ellipsis button in the DataRepeater control's (Custom) property to display the Property Pages dialog box. Click on the RepeaterBindings tab to display the property page shown in Figure 6-16.
17. Select a property name from thePropertyName drop-down list and a record field to display in that property from the DataField drop-down list, and then click Add.
18. Repeat step 8 for each of the properties in the data-bound control. Click OK when done.

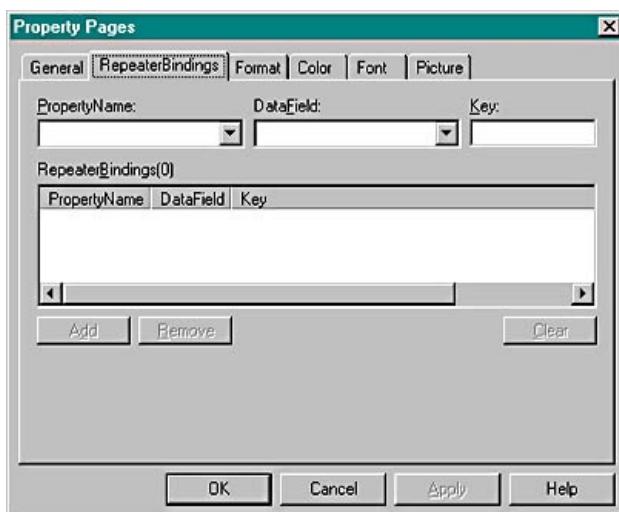


Figure 6-16. Bind the repeated control's properties to the fields in a record using the DataRepeater control's Property Pages dialog box.

At runtime, you can scroll through records using the arrows on the ADO Data control or the scroll bar on the DataRepeater control. For this reason, you may want to hide the ADO Data control at runtime.

Dear John, How Do I... Create a Container Control?

If you set a user control's ControlContainer property to True, objects drawn on top of that control can be moved and resized as a group. The Microsoft Tabbed Dialog and DataRepeater controls are two examples of container controls that ship with Visual Basic. Figure 6-17 shows a simple container control created in Visual Basic using Shape and Label controls to create a frame.

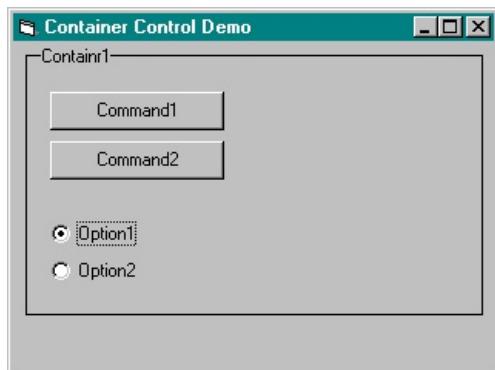


Figure 6-17. The Containr sample control's ControlContainer property is True, so it mimics the behavior of a Frame control.

The code for the Containr control (Containr.VBP) handles resizing the frame as shown below:

```
'User interaction section
Private Sub UserControl_Resize()
    'Resize the frame to match control
    shpFrame.Width = UserControl.ScaleWidth - shpFrame.Left
    shpFrame.Height = UserControl.ScaleHeight - shpFrame.Top
End Sub
```

The Containr control's caption is initialized using the Extender object, which gives access to the built-in properties that Visual Basic and other control containers provide to all objects. The Extender object is available to the InitProperties procedure, but will cause an error if you try to access it earlier in the control's life-cycle, such as from the control's Initialize event. The following code displays the control name that Visual Basic automatically assigns when a user creates an instance of the control on a form:

```
'Control maintenance section
Private Sub UserControl_InitProperties()
    'Display appropriate caption
    lblTitle.Caption = Extender.Name
End Sub
```

The Caption property sets and returns the caption displayed in lblTitle. This property is Read/Write and available at design time, so it has Property Let and Get procedures as well as code in the ReadProperties and WriteProperties event procedures, as shown below.

```
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    lblTitle.Caption = PropBag.ReadProperty("Caption")
End Sub
```

```
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "Caption", lblTitle.Caption
End Sub
```

```
'Properties section
Public Property Get Caption() As String
    Caption = lblTitle.Caption
End Property
```

```
Public Property Let Caption(Setting As String)
    lblTitle.Caption = Setting
End Property
```

The Controls property returns the collection of controls contained in the Containr control. This is the only feature you don't get with the standard Frame control, and it is included with this example to show you how to use the ContainedControls collection. The ContainedControls collection is created by Visual Basic when you set ControlContainer to True; it is not available on other types of controls. The following code shows the definition for the read-only Controls property:

```
'Read-only property
Public Property Get Controls() As Collection
    Set Controls = UserControl.ContainerControls
End Property
```

Chapter Seven

Using Internet Components

Internet programming means different things to different people. From the standpoint of commercial software, it might mean developing new client/server tools that use the Web. To a sales and marketing department, it might mean creating a Web page that can take order information to sell products over the Internet. And to a human resources department, it might mean publishing an employee handbook that runs as a stand-alone application easily updated over an intranet.

Because of this diversity, I've divided the subject of Internet programming into three chapters for this edition:

This chapter discusses how to use the ActiveX controls that come with Visual Basic to handle everything from low-level protocol communication to the creation of your own Web browser based on Internet Explorer (IE).

Chapter 8, "[Creating Internet Components](#)," shows you how to create the different types of formlike containers provided by Visual Basic for distributing content over the Internet. Chapter 8 also has information about creating ActiveX controls and Web classes to be used over the Internet.

Chapter 9, "[Creating Internet Applications](#)," puts the information from [Chapter 7](#) and [Chapter 8](#) to work by explaining how to create three different classes of application in Visual Basic.

Dear John, How Do I... Select the ActiveX Component to Use?

The Microsoft Visual Basic Professional and Enterprise Editions include three ActiveX controls that let you access three different levels of Internet communication:

- The Winsock control (MSWINSCK.OCX) provides low-level access to the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) network protocols used on the Internet. The Winsock control is used to create chat applications and to perform direct data transfers between two or more networked computers.
- The Internet Transfer control (MSINET.OCX) lets you copy files from Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) servers and lets you browse directories of files and retrieve data synchronously or asynchronously. This control is used to create FTP browsing and file transfer applications.
- The WebBrowser control (SHDOCVW.DLL) packs all the capabilities of Internet Explorer into a single control. Use it to add an Internet browser to an existing application or to control Internet Explorer from another application through Automation.

NOTE

The WebBrowser control is called Microsoft Internet Controls in the Visual Basic Components dialog box. It is installed as part of the IE setup. IE version 4.0 is included on the Visual Studio CD-ROM.

The following sections talk about these different levels of Internet access and provide examples of how to use each of these controls.

Dear John, How Do I... Understand Internet Protocol Layers?

The Internet is a system by which a growing number of different computer networks can communicate. The Internet manages this diversity through a series of layers called *protocols*. Like the social protocols we observe in everyday life, the Internet protocols let each layer know what to expect. Figure 7-1 shows how some of the most important protocols are arranged.

The transport layer marks the boundary of where your application ends and the Internet begins. This layer consists of the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). These two protocols determine the way in which applications send and receive data over the Internet—specifically, they decide how the connection is made and how the data is packaged.

You can use UDP or TCP for direct communication over the Internet. For example, you might want to send some application-defined data format or transmit a simple message to multiple machines. Because they operate at a lower level, UDP and TCP don't incur the overhead that is present in application-level protocols such as HTTP or FTP.

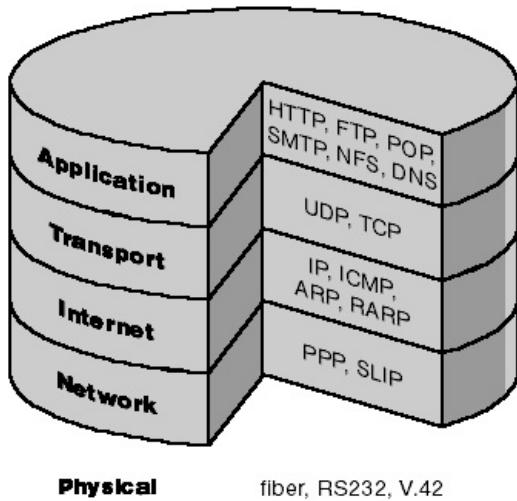


Figure 7-1. The protocol layers help manage the complexity of Internet communication.

As mentioned at the beginning of this chapter, the Winsock control provides direct access to the UDP and TCP protocols, and the Internet Transfer control provides direct access to the FTP and HTTP protocols. The WebBrowser control interprets FTP and HTTP data, formatting it as it would appear in IE. The control you choose depends on the type of application you wish to create, as described in the following sections.

Dear John, How Do I... Set Up Networking?

To work with most of the Internet components discussed in this and the next two chapters, you'll need access to at least two networked computers running the Transmission Control Protocol/Internet Protocol (TCP/IP). Windows 95 and Windows NT include the TCP/IP protocol, but you should check to make sure it is installed. To do this, double-click the Network icon in the Control Panel, and then click on the Protocols tab in Windows NT or the Configuration tab in Windows 95. Windows lists the protocols that are installed on your computer, as shown in Figure 7-2 below.

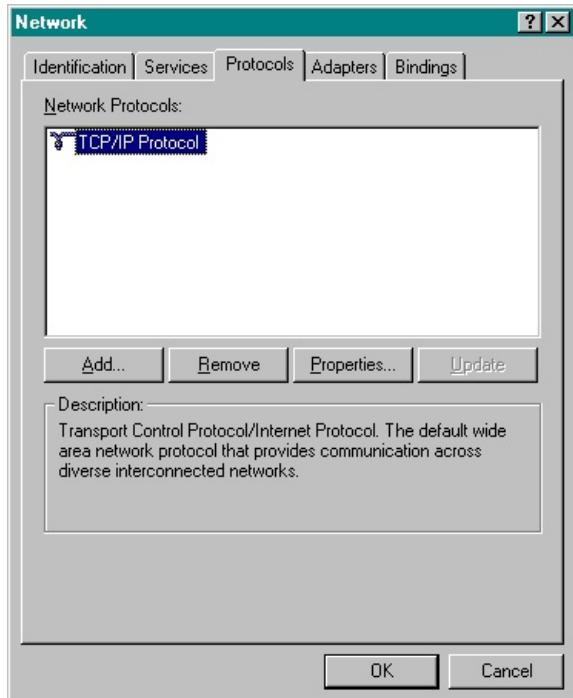


Figure 7-2. The TCP/IP protocol must be installed in order to use the Winsock control.

If you don't have a full-blown network, you can use Windows NT Server Remote Access Service (RAS) to create a dial-in server and dial in to that machine using the Dial-Up Networking utility. RAS is installed as an option of Windows NT Server and the Dial-Up Networking utility is an accessory included with both Windows NT and Windows 95.

If you are using RAS and dial-up networking, you probably won't be able to use "friendly names" when connecting to other computers on the network. Instead, you'll have to use a fixed Internet Protocol (IP) address, which can be set using the TCP/IP Properties dialog box, as shown in Figure 7-3.

The IP address can be used anywhere a friendly name is expected. For instance, you can use it with the Winsock control's RemoteHost property, in the IE's Address text box, or with the Internet Transfer control's OpenURL (Uniform Resource Locator) method.

You will probably also want to install Microsoft Personal Web Server (PWS) or Microsoft Internet Information Server (IIS) in order to publish files on one machine and view them from another using IE. This is handy for transferring files from one machine to the other and allows you to debug Active Server Pages (ASP) and test the installation of Internet components before deploying them. PWS and IIS are installed as part of the server components available with Microsoft Visual Studio.

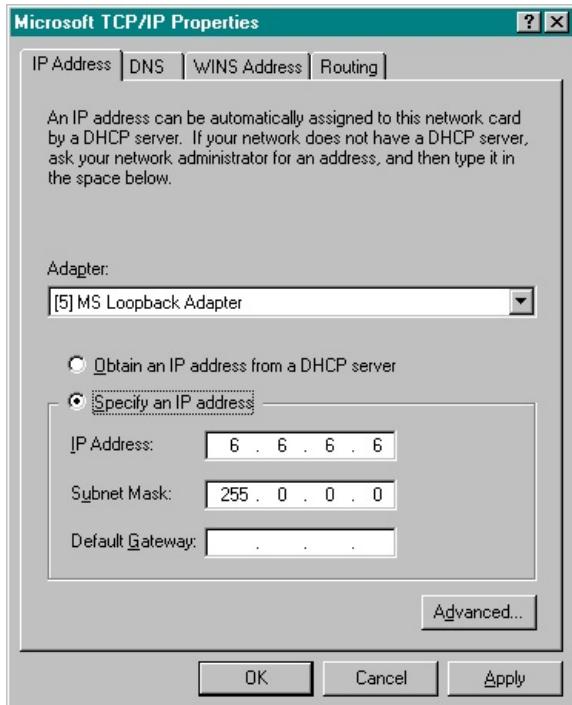


Figure 7-3. The TCP/IP Properties dialog box lets you set a fixed IP address to which you'll be able to connect using RAS and Dial-Up Networking.

Dear John, How Do I... Communicate Using Winsock?

Applications connect to the Internet through an IP address and a port number. The combination of an IP address and a port number is called a *socket*. You establish a connection between two machines by creating a pair of sockets that match.

Use the Winsock control (MSWINSCK.OCX) to create matching sockets for use with UDP or TCP. Your choice of protocol depends on the type of communication you wish to establish:

- UDP provides "connectionless" communication. You send out data without knowing when or if anyone receives them. This is useful for network file services, chat applications, and broadcasting of operator messages over the network.
- TCP requires you to open a connection with another computer before transmitting data and to close the connection when done. TCP maintains the sequence of transmission and provides confirmation that the data were received correctly. This is useful for transferring files and other large data streams, sending and receiving electronic mail, and establishing direct user-to-user communication.

The following sections show how to use the Winsock control to create Internet applications with UDP and TCP. Before you get carried away and start programming your own mail system, however, you should realize that Winsock is a low-level tool. There are already implementations of application-level protocols for Visual Basic. These include the Internet Transfer control (MSINET.OCX) for FTP and HTTP and other commercially available controls for Post Office Protocol (POP), audio, and three-dimensional (3D) data. Information about third-party controls is available on the Internet. Here are some starting points:

- See Mabry Software, Inc. Internet Pack for POP, Finger, Whois, Simple Mail Transfer Protocol (SMTP), and other controls at <http://www.mabry.com>
- See devSoft Inc. IP*Works! for POP, SMTP, DNS, and other controls at <http://dev-soft.com>
- Template Graphics Software, Inc. TGS 3D Developer Toolkit products, at <http://www.tgs.com>

Broadcasting with UDP

Using the Winsock control with UDP is a two-part process: you use one set of steps to send data and another set to receive them.

To send data, add a Winsock control to a form and then follow these general steps:

1. Set the RemotePort property to the port number used by other machines to receive the data you are going to send.
2. Set the RemoteHost property. This is the IP address or the friendly name of the host computer which will receive your data.
3. Use the Bind method to open a local port from which to send the data.
4. Use the SendData method to transmit the data.

To receive data using UDP, follow these general steps:

1. Set the RemotePort property to the port number used by the Bind method in step 3 of the previous procedure.
2. Use the Bind method to open a local port over which to receive the data. This is the same port number used in the RemotePort property in step 1 of the previous procedure.
3. Use the GetData method in the DataArrival event procedure to retrieve the data sent.

A Broadcasting Sample

Because you don't need to identify the remote host when receiving data, UDP can be used to get messages from many different computers. This many-to-one relationship lets you create administrative tools that monitor the status of the workstations on a network, such as the Systems Status Monitor (SysStat.VBP) shown in Figure 7-4.

Local machine command line

SysStat.exe wombat1

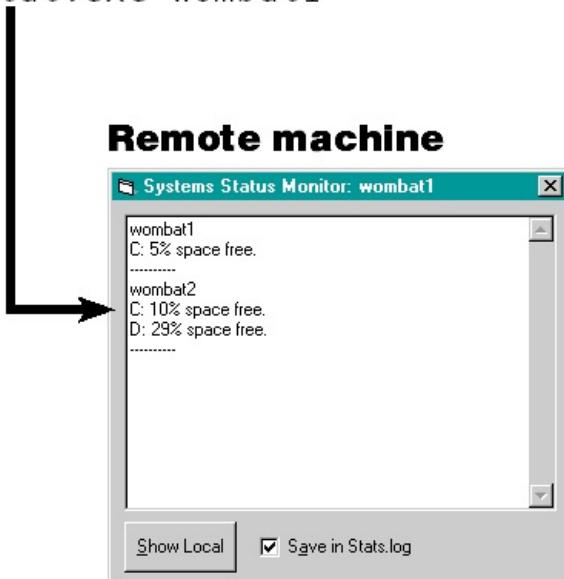


Figure 7-4. The Systems Status Monitor reports the amount of free disk space from workstations to an administrative machine.

The Systems Status Monitor has two functions: When invoked with a command line, it sends the percentage of free disk space from the local machine to the remote machine identified in the command line. When invoked without a command line, the Systems Status Monitor acts as the passive recipient of disk information sent from workstations.

The following Form_Load event procedure shows how the Winsock control is set to send or receive data, depending on the command line.

```

Option Explicit
`Create object for log file
Dim fsysLog As New Scripting.FileSystemObject

`Command line specifies the name or IP address of
`the machine to send data to
Private Sub Form_Load()
    `Start this application in send mode
    `if a command line is specified

    If Command <> "" Then
        `Set a remote port to send to
        sckUDP.RemotePort = 1002
        `Identify the host to send to
        sckUDP.RemoteHost = Command
        `Bind to a local port to send from
        sckUDP.Bind 1001
        `Send information
        sckUDP.SendData ShowStats
        `End this application
        Unload Me
    `Start in receive mode if no
    `command line
    Else
        `Specify remort port to listen for
        sckUDP.RemotePort = 1001
    End If
End Sub

```

```

`Specify local port to receive from
sckUDP.Bind 1002
`Display information along with this
`machine's "friendly" name
Me.Caption = Me.Caption & ":" & sckUDP.LocalHostName
End If
End Sub

```

The Winsock control's Error event procedure is called if there is no Internet connection or if the specified remote host name is invalid. Because the Systems Status Monitor uses UDP, error checking is minimal. The Systems Status Monitor simply displays any errors that occur, as shown here:

```

`Handle transmission errors
Private Sub sckUDP_Error(ByVal Number As Integer, _
    Description As String, ByVal Scode As Long, _
    ByVal Source As String, ByVal HelpFile As String, _
    ByVal HelpContext As Long, CancelDisplay As Boolean _
)
    `Display error messages
    MsgBox Number & " " & Description, vbCritical, App.Title
End Sub

```

The Winsock control's DataArrival event procedure is called when data is received over the bound port. The GetData method retrieves the data that has arrived and clears the receive buffer so that the next item in the queue is available.

```

`Handle data as it comes in
Private Sub sckUDP_DataArrival(ByVal bytesTotal As Long)
    Dim strData As String
    `Retrieve the data
    sckUDP.GetData strData
    `Display it
    ShowText strData
    `Save data in log file if check box selected
    If chkLog.Value Then
        fsysLog.CreateTextFile(CurDir & "\stats.log", _
            True).Write txtReceive.Text
    End If
End Sub

```

The amount of data that can be sent via UDP as a single chunk is determined by your network. You can experiment to find the upper limit, but it is not a good idea to rely on UDP for exchanging large amounts of data. For that, you should use TCP.

One-on-One Chatting with TCP

To use the Winsock control with TCP, you must establish a connection before attempting to transmit data. Creating the connection has two parts: one machine requests a connection, and the other machine accepts it.

To request a connection, add a Winsock control to a form and then follow these general steps:

1. Set the RemoteHost property. This is the IP address or the friendly name of the host computer which will receive the data you are going to send.
2. Set the RemotePort property to the port used by other machines to receive your data.
3. Use the Connect method to request a connection to the other machine.
4. Use the SendData method to transmit the data.

To accept a connection, follow these general steps:

1. Set the LocalPort property to the port specified by the RemotePort property in step 2 of the preceding procedure.
2. Use the Listen method to open the local port for connection requests.
3. Use the Accept method in the ConnectionRequest event procedure to establish the connection.

Once the connection is established, data can be exchanged using the SendData and GetData methods, just as with UDP. Unlike UDP, however, TCP requires that you use the Close method to end the existing connection before you can create a new connection.

A Chatting Sample

The Chat sample (Chat.VBP) demonstrates how to use Winsock with TCP to create a connection between two machines. Once the connection is established, two users can exchange messages by typing in the text box as shown in Figure 75.

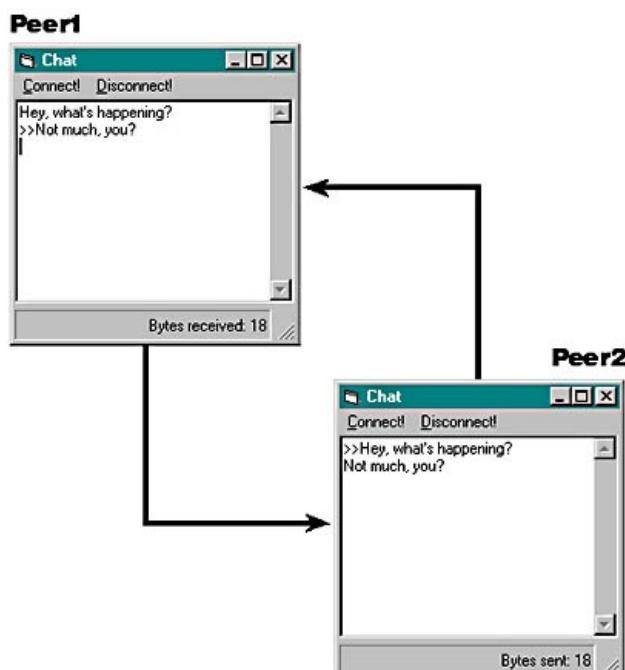


Figure 7-5. The Chat sample allows two users to send messages to one another by using TCP.

The Chat sample starts out listening for connection requests. The Form_Load event procedure sets the local port to listen on and then starts listening, as shown in the following code.

```
Option Explicit

`Start out listening for connection
`requests
Private Sub Form_Load()
    `Set the port to listen on
    sckTCP.LocalPort = 1002
    `Begin listening
    sckTCP.Listen
    `Update status bar
    ShowText "Listening"
End Sub
```

To initiate a connection, call the Connect method. The Chat sample allows users to specify a machine name or an IP address to connect through an InputBox, closes any existing connection, and then establishes the new connection as shown below.

```
Private Sub mnuConnect_Click()
    Dim strRemoteHost As String
```

Microsoft® Visual Basic® 6.0 Developer's Workshop

```
'Get the name of a computer to connect to
strRemoteHost = InputBox("Enter name or IP address of computer " & _
    "to connect to.", vbOKCancel)
`Exit if cancelled
If strRemoteHost = "" Then Exit Sub
`Close any open connections
sckTCP.Close
`Set the name of the computer to connect to
sckTCP.RemoteHost = strRemoteHost
`Specify a port number on remote host
sckTCP.RemotePort = 1002
`This seems to prevent some TCP errors
DoEvents
`Request the connection
sckTCP.Connect
End Sub
```

The ConnectionRequest event occurs when the listening machine receives a request from another machine. The Chat sample accepts all requests, as shown below.

```
Private Sub sckTCP_ConnectionRequest(ByVal requestID As Long)
    sckTCP.Close
    sckTCP.Accept requestID
    ShowText "Accepting request from " & sckTCP.RemoteHostIP
End Sub
```

Once the listening machine accepts a connection, either machine can transmit data to the other using the SendData method. The Chat sample uses the txtChat text box to compose messages to send and display messages received. The KeyPress event procedure below keeps track of what the user types and sends the message when the user presses Enter.

```
Private Sub txtChat_KeyPress(KeyAscii As Integer)
    Static strSend As String
    `Make sure there is a connection
    If sckTCP.State <> sckConnected Then Exit Sub
    `Send data when user presses Enter
    If KeyAscii = Asc(vbCr) Then
        `Send the string
        sckTCP.SendData strSend
        `Clear the variable
        strSend = ""
    Else
        `Keep track of what is being typed
        strSend = strSend & Chr(KeyAscii)
    End If
End Sub
```

The SendData method above triggers the DataArrival event on the receiving side of the connection. The Chat sample uses GetData to retrieve information from the message queue and displays it in the txtChat text box as shown by the following code.

```
Private Sub sckTCP_DataArrival(ByVal bytesTotal As Long)
    Dim strText As String
    `Get data
    sckTCP.GetData strText
    `Display data received
    txtChat = txtChat & ">>" & strText & vbCrLf
    `Move cursor to end
    txtChat.SelStart = Len(txtChat)
    ShowText "Bytes received: " & bytesTotal
End Sub
```

To end the connection, simply call the Close method. The Chat sample includes a menu item to

disconnect and return to listening as shown by the following code.

```
Private Sub mnuDisconnect_Click()
    sckTCP.Close
    DoEvents
    sckTCP.Listen
    ShowText "Listen"
End Sub
```

The Close method triggers the Close event on the other machine. When a Close event occurs, the Chat sample returns to listening for new connections as shown by the following code.

```
Private Sub sckTCP_Close()
    ShowText "Close"
    'When connection by remote machine, go back to listening
    sckTCP.Close
    sckTCP.Listen
    ShowText "Listen"
End Sub
```

TCP connections provide a great deal more error information than UDP connections, so it is important to display error information to the users. The Error event happens whenever an exception occurs in establishing or using a connection. The Chat sample displays errors using the following code.

```
'Display error information
Private Sub sckTCP_Error(ByVal Number As Integer, _
    Description As String, ByVal Scode As Long, _
    ByVal Source As String, ByVal HelpFile As String, _
    ByVal HelpContext As Long, CancelDisplay As Boolean)
    ShowText "Error " & Number & " " & Description
End Sub
```

For more information on events that occur during a TCP connection, see the Chat sample (Chat.VBP) included on the companion CD-ROM. Additional description is also available in the topic "Using the Winsock Control" in the Visual Basic online help.

Dear John, How Do I... Create an FTP Browser?

The Visual Basic online help for the Internet Transfer control demonstrates the pieces you would use to create an FTP browser, but it doesn't assemble those pieces into a working application. This is a problem because the Internet Transfer control is asynchronous—how the events and error handling interact is the most difficult aspect of using the control.

Figure 7-6 below shows a simple FTP browser I've created using two text boxes and an Internet Transfer control. You enter the URL of an FTP server in the Address text box, and then select a file or a directory from the contents text box. If the selection is a directory, the application displays that directory. If the selection is a file, the browser saves the file in the Windows Temp directory.

When the user presses Enter, the Address text box executes requests by setting the Internet Transfer control's URL property and calling the Execute method. The OpenURL method performs the same action when requesting a specific file. However, when you use the OpenURL method to return the contents of a directory, HTML source code indicating the directory contents typically is returned. Therefore, in this example, I have steered clear of the OpenURL method. The code following Figure 7-6 shows the KeyPress event procedure for the *txtAddress* text box.



Figure 7-6. A simple FTP browser created using the Internet Transfer control.

```

Private Sub txtAddress_KeyPress(KeyAscii As Integer)
    If KeyAscii = Asc(vbCr) Then
        'Eat keystroke
        KeyAscii = 0
        'Select text
        txtAddress.SelStart = 0
        txtAddress.SelLength = Len(txtAddress)
        On Error GoTo errOpenURL
        'Set FTP address to view
        inetBrowse.URL = txtAddress
        'Get directory
        inetBrowse.Execute , "Dir "
        txtAddress = inetBrowse.URL
    End If
    Exit Sub

errOpenURL:
    Select Case Err.Number
        Case icBadUrl
            MsgBox "Bad address. Please reenter."
        Case icConnectFailed, icConnectionAborted, _
              icCannotConnect
            MsgBox "Unable to connect to network."
        Case icInetTimeout
            MsgBox "Connection timed out."
        Case icExecuting
            'Cancel previous request
            inetBrowse.Cancel
            'Check whether cancel worked
    End Select
End Sub

```

```

        If inetBrowse.StillExecuting Then
            Caption = "Couldn't cancel request."
        `Resubmit current request
        Else
            Resume
        End If
    Case Else
        Debug.Print Err.Number, Err.Description
    End Select
End Sub

```

Trapping Errors

It's important to trap any errors that occur when you submit a request to the Internet Transfer control. The icExecuting error is particularly important. The Internet Transfer control processes all requests asynchronously; however, it can process only one request at a time. If you cancel a pending request, be sure to check the StillExecuting property before resuming, as shown in the previous code. Some requests can't be canceled and using only a Resume statement will result in an infinite loop!

The following code shows the DblClick event procedure for the *txtContents* text box, in which directory listings are displayed. This code builds the URL string and executes a Dir command if the selection is a subdirectory, or a Get command if the selection is a file.

```

Private Sub txtContents_DblClick()
    `Browse selected directory
    If txtContents.SelLength Then
        `If selection is a directoryDear John, How Do I...
        If Right(txtContents.SelText, 1) = "/" Then

            `Add selected item to address
            txtAddress = txtAddress & "/" & _
                Left(txtContents.SelText, _ 
                    txtContents.SelLength - 1)
            `Trap errors (important!)
            On Error GoTo errBrowse
            `Show directory
            mstrDir = Right(txtAddress, Len(txtAddress) -
                - Len/inetBrowse.URL))
            inetBrowse.Execute , "Dir " & mstrDir & "/*"
            `Otherwise, it's a file, so retrieve it
        Else
            Dim strFilename
            `Build pathname of file
            mstrDir = Right(txtAddress, Len(txtAddress) -
                - Len/inetBrowse.URL)) & "/" & _
                txtContents.SelText
            mstrDir = Right(mstrDir, Len(mstrDir) - 1)
            strFilename = mstrDir
            Do
                strFilename = Right(strFilename,
                    Len(strFilename) - InStr(strFilename, "/"))
            Loop Until InStr(strFilename, "/") = 0 `Retrieve file
            inetBrowse.Execute , "Get " & mstrDir & _
                " " & mstrTempDir & strFilename
        End If
    End If
Exit Sub
errBrowse:
    If Err = icExecuting Then
        `Cancel previous request
        inetBrowse.Cancel
        `Check whether cancel worked
        If inetBrowse.StillExecuting Then
            Caption = "Couldn't cancel request."

```

```

`Resubmit current request
Else
    Resume
End If
Else
    `Display error
    Debug.Print Err & " " & Err.Description
End If
End Sub

```

The Execute and OpenURL methods trigger the StateChanged event. It's important to remember that both methods do this, because OpenURL appears to be a synchronous method; however, StateChanged events still occur and can cause problems with reentrancy.

The following code updates the form's caption to keep you abreast of the request's progress and then uses the GetChunk method to retrieve a directory listing if the command executed was Dir.

```

Private Sub inetBrowse_StateChanged(ByVal State As Integer)
    Select Case State
        Case icError
            Debug.Print inetBrowse.ResponseCode & " " & _
                inetBrowse.ResponseInfo
        Case icResolvingHost, icRequesting, icRequestSent
            Caption = "SearchingDear John, How Do I... "
        Case icHostResolved
            Caption = "Found."
        Case icReceivingResponse, icResponseReceived
            Caption = "Receiving data."
        Case icResponseCompleted
            Dim strBuffer As String
            `Get data
            strBuffer = inetBrowse.GetChunk(1024)
            `If data is a directory, display it
            If strBuffer <> "" Then
                Caption = "Completed."
                txtContents = strBuffer
            Else
                Caption = "File saved in " & _
                    mstrTempDir & "."
            End If
        Case icConnecting, icConnected
            Caption = "Connecting."
        Case icDisconnecting
        Case icDisconnected
        Case Else
            Debug.Print State
    End Select
End Sub

```

NOTE

The current documentation suggests that GetChunk will work with the Execute method's GET command, but that does not seem to be the case. The GET command's syntax specifies a source file and a destination file; therefore, GetChunk is not needed when you are copying a file from a server.

The complete code for the FTP browser, including the GetTempPath API function declaration and the Form_Load event procedure, can be found on the companion CD-ROM.

Dear John, How Do I... Control Internet Explorer?

An interesting feature of the WebBrowser control is the fact that its source file, SHDOCVW.DLL, also contains a *type library* for IE. A type library contains all the information needed to create and control ActiveX objects through Automation (formerly OLE Automation). You can reference SHDOCVW.DLL in your Visual Basic projects to create and control instances of the IE application.

This technique is practical when you are debugging Internet applications written in Visual Basic. By default, IE caches Web pages; however, you will usually want fresh copies of each Web page when debugging. The WebTool add-in sample, shown in Figure 7-7, starts an instance of IE with caching turned off, making it easier to debug Internet applications in Visual Basic.

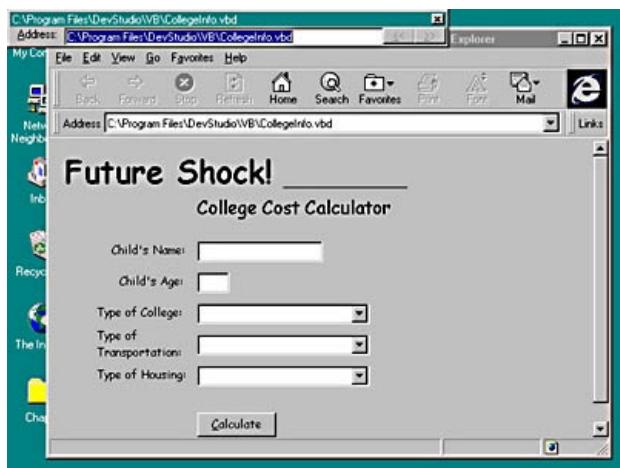


Figure 7-7. The WebTool add-in always loads fresh Web pages rather than using cached pages.

In addition to the standard connection class (CLS) and registration startup module (BAS) used by all add-ins, the WebTool add-in (WEBTOOL.VBP) includes a single form with a text box for entering URLs and forward and back command buttons.

The following code creates an instance of IE and sets the *txtAddress* text box to the path in which Visual Basic stores temporary Visual Basic Document (VBD) files while debugging Internet applications. VBD files are discussed in detail in Chapter 9, "[Creating Internet Applications](#)."

```

Option Explicit

' deletes cached page, replaces the Navigate method's
' navNoReadFromCache flag
Private Declare Function DeleteUrlCacheEntry Lib _
    "WinInet.DLL" (strURL As String) As Boolean

'Specify the path where VB.EXE is installed
Const VBPath = "file:///C:/Program Files/DevStudio/VB/"

`Create an Internet Explorer object variable. Note that
`you must use the "_V1" object to be able to bind to
`the ieView_Quit event
Private WithEvents ieView As SHDocVw.WebBrowser_V1
`In IE3, the following declaration was used:
`Private WithEvents ieView As InternetExplorer

Private Sub Form_Load()
    `Establish a reference to application object
    Set ieView = GetObject("", "InternetExplorer.Application")
    `Be sure Internet Explorer is visible
    ieView.Visible = True
    `Start with VB.EXE path because that's where VBD
    `files are stored during debugging
    txtAddress = VBPath
End Sub

```

Unfortunately, you can't use the GetObject function with the first argument empty to get a running instance of IE. The application doesn't allow it, so you need to start a new instance when you establish your object reference. You also need to be sure to make the instance visible by setting its Visible property to *True*.

The following code does the navigation work. The DeleteUrlCacheEntry API function guarantees that the Navigate method loads a new version of the URL, rather than loading the URL from an older version cached on your machine.

```
Private Sub txtAddress_KeyPress(KeyAscii As Integer)
Dim blnResult As Boolean
If KeyAscii = Asc(vbCr) Then
    'Eat keystroke
    KeyAscii = 0

    'Select text
    txtAddress.SelLength = Len(txtAddress)
    'Delete this URL if it is cached
    blnResult = DeleteUrlCacheEntry(ByName txtAddress)
    'Navigate to address
    ieView.Navigate txtAddress
End If
End Sub
```

The *ieView* object variable is declared using WithEvents, so this form can intercept events from Internet Explorer. The CommandStateChange event is used to enable or disable the forward and back command buttons.

```
Private Sub ieView_CommandStateChange( _
    ByVal Command As Long, _
    ByVal Enable As Boolean _
)
    'Enable or disable Back and Forward command buttons
    'based on whether there is an address to go to
    Select Case Command
        Case CSC_NAVIGATEBACK
            cmdBack.Enabled = Enable
        Case CSC_NAVIGATEFORWARD
            cmdForward.Enabled = Enable
        Case CSC_UPDATECOMMANDS
    End Select
End Sub

Private Sub cmdBack_Click()
    ieView.GoBack
End Sub

Private Sub cmdForward_Click()
    ieView.GoForward
End Sub
```

Internet Explorer triggers the NavigateComplete event when the Web page has been displayed and triggers the Quit event when the user closes the application. The following code responds to those events in the WebTool add-in.

```
Private Sub ieView_NavigateComplete(ByVal URL As String)
    'Update text box with the final address
    txtAddress.Text = URL
    txtAddress.SelLength = Len(txtAddress)
    'Display the Web page title in the form's caption
    Caption = ieView.LocationName
End Sub

Private Sub ieView_Quit(Cancel As Boolean)
```

```
'Close this application if user closes Internet Explorer  
End  
End Sub
```

All the code used to create the WebTool add-in can be found on the companion CD-ROM.

SEE ALSO

- Chapter 27, "[Advanced Programming Techniques](#)," for more information about creating an add-in

Chapter Eight

Creating Internet Components

The preceding chapter discussed how to use the Internet components that ship with Visual Basic. Those components provide the client and peer services necessary to write chat, file server, and browser applications that use the Internet. This chapter covers how to create your own components for use over the Internet. These components can take several forms:

- ActiveX controls extend the capabilities of any Internet application, whether it uses Hypertext Markup Language (HTML), Dynamic HTML (DHTML), or ActiveX documents. Chapter 6, "[ActiveX Controls](#)," covers the fundamentals of creating these components, but this chapter discusses the special considerations for using them over the Internet.
- HTML pages make up most of the material on the Internet. Visual Basic doesn't include tools for authoring HTML directly, but it does provide components that can be used in HTML such as ActiveX components and Visual Basic, Scripting Edition (VBScript). This chapter shows you how to use these tools in HTML.
- DHTML pages are a new type of component that Visual Basic can create. This chapter shows how to use the DHTML Page Designer and discusses things you need to consider to use these pages within an application.
- ActiveX documents are Visual Basic's earlier attempt at dynamic Web pages before the DHTML standard was proposed. These components are different from both Visual Basic forms and DHTML, so again there are special considerations.
- Web classes manage server-side objects in Internet Information Server (IIS) applications. These classes run under IIS and make use of the IIS object model.

This chapter shows how to use Visual Basic to create or enhance these components. The topics in this chapter are closely related to the topics in Chapter 9, "[Creating Internet Applications](#)," so you may want to read these chapters together to understand the breadth of Internet programming with Visual Basic.

Dear John, How Do I... Create ActiveX Controls for Internet Use?

You can use ActiveX controls in any Visual Basic application, including those that run over the Internet. However, there are some special considerations when creating controls for use over the Internet.

Digital Signature All executable components should bear a "digital signature" identifying the author and certifying that they are safe to run and safe to use in scripts. You can get a digital signature from VeriSign (<http://www.verisign.com>) or other vendors for a fee.

Threading Model ActiveX controls must use a threading model that is compatible with their host application. Single-threaded applications can use controls with either single threading or apartment threading, but apartment-threaded applications can only use apartment-threaded controls. DHTML applications are apartment threaded.

Asynchronous Loading By default, all objects load asynchronously over the Internet. You need to be careful that objects are initialized before you refer to them in code. Use the IsNull function to see if an object exists before referring to it.

Packaged for Distribution ActiveX controls installed over the Internet must have a setup program to install them. The easiest way to do this is to use the Package and Deployment Wizard to create the necessary .CAB and .HTM files. See Chapter 9, "[Creating Internet Applications](#)," for information about using the Package and Deployment Wizard to create Internet setup programs.

The Asynchronous Animation control created in Chapter 6, "[ActiveX Controls](#)," is designed for use over the Internet. The following sections use that control in HTML, DHTML, and ActiveX document components in each of their samples, which can be found on the companion CD-ROM.

Dear John, How Do I... Use ActiveX Controls with VBScript?

HTML documents are the most common form of data on the Internet. HTML documents displayed in Internet Explorer can include ActiveX controls and VBScript, JScript, or Java applets that respond to events. For example, Figure 8-1 shows the Asynchronous Animation control created in Chapter 6, "[ActiveX Controls](#)," being used in an HTML document named htAni.HTM.

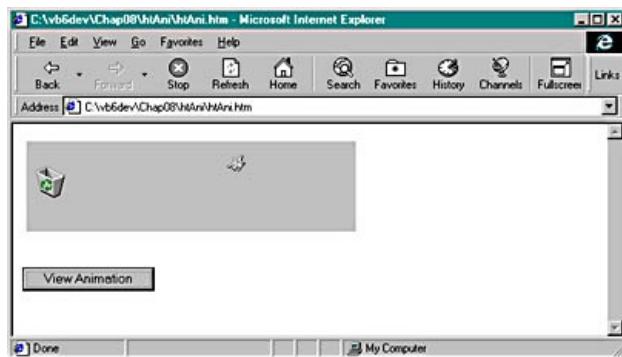


Figure 8-1. Asynchronous Animation control embedded in a Web page.

The HTML code for this Web page is shown below. Notice that the Asynchronous Animation and common dialog controls are included using the <OBJECT> tag and are automated using VBScript code.

```
<HTML>
<P>
<OBJECT ID="asyncAnimation" WIDTH=200 HEIGHT=80
CLASSID="CLSID:1B9C2FBC-0156-11D2-A061-00AA005754FD"
/>

<OBJECT ID="dlgFile" HEIGHT=50 WIDTH=50
CLASSID="CLSID:F9043C85-F6F2-101A-A3C9-08002B2F49FB"
/>

<P>
<INPUT TYPE="BUTTON" VALUE="View Animation" NAME="butView">

<SCRIPT LANGUAGE="VBScript">
`Event procedure for the intrinsic button control
Sub butView_OnClick
    `Make sure objects exist
    If IsNull(dlgFile) Or IsNull(asyncAnimation) Then Exit Sub
    `Get a file
    dlgFile.ShowOpen
    strFile = dlgFile.FileName
    `If a filename was entered
    If strFile <> "" Then
        `Dear John, How Do I...
        `Display it in the asyncAnimation control
        asyncAnimation.AVIFile = strFile
        `Reset FileName
        dlgFile.FileName = ""
    End If
End Sub

`Handle error events from the asyncAnimation control
Sub asyncAnimation_FileError(Number, Description)
    `Display a message with the error
    MsgBox "File is not a valid animation.", _
           vbOKOnly And vbCritical, "Error " & Number
End Sub
</SCRIPT>
</HTML>
```

The easiest way to get the class ID information for the control is by running the Package and Deployment

Wizard on the control's project file as described in the section, "[Dear John, How Do I... Install ActiveX Documents over the Internet?](#)" in Chapter 9, "[Creating Internet Applications](#)." The Package and Deployment Wizard generates an .HTM file with an <OBJECT> tag that includes the control on the page.

You can modify the generated .HTM file by adding text, graphics, intrinsic controls, other ActiveX components, and executable scripts. There are a few points to remember when working with VBScript:

- All variables are of the type Variant. You can't declare variables using the Dim statement. Instead, variables are created dynamically the first time they are used.
- All parameters are Variants. Be sure to delete the type declarations in procedure definitions if you move code from Visual Basic to VBScript. Also remember to omit data types when declaring event procedures for ActiveX objects.
- VBScript does not include file I/O statements. Use the FileSystemObject object to work with drives, directories, and files.

See the online Help topics "VBScript Language Reference" and "VBScript Tutorial" in the Visual Studio MSDN Library for more complete information about using VBScript. See also *Inside Microsoft Visual Basic Scripting Edition*.

Dear John, How Do I... Create DHTML Documents?

DHTML is included in the next version of HTML, known as HTML 4.0. DHTML makes Web pages more interactive by exposing a programmable Document Object Model (DOM) and providing cascading style sheets (CSS).

NOTE

There are competing implementations of DHTML from Microsoft and Netscape. The World Wide Web Consortium (W3C) referees such standards and has posted a recommendation at its Web site: <http://www.w3c.org>. Future versions of both browsers may support the same DHTML.

Visual Basic provides a new designer for working with DHTML pages. The Visual Basic DHTML Page Designer, shown in Figure 8-2, displays a hierarchical diagram of the page and an edit area you can use to type text and add objects directly to the page.

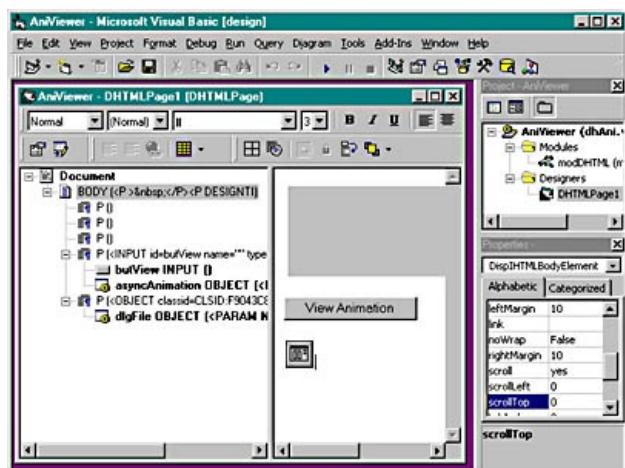


Figure 8-2. Creating a new Web page in the Visual Basic DHTML Page Designer.

Using the DHTML Page Designer is different than creating a Visual Basic form, ActiveX document, user control, or property page. The edit area of the designer is more like a browser client than any other Visual Basic component. Specifically, you will notice these major differences when working with the DHTML Page Designer:

- DHTML pages identify controls by their ID property, rather than their Name property. The DHTML designer automatically assigns an ID to each control you add to the page. If you change a control's ID, you must make sure to use a unique name since the designer does not check to make sure the name is unique. Visual Basic does not generate an error message if you accidentally assign the same ID to two different controls.
- Visual Basic's intrinsic controls are not available. Instead you must use the intrinsic HTML controls or ActiveX controls.
- While debugging, Internet Explorer sometimes stops responding when a Visual Basic element displays a window. You must switch to Visual Basic to see the window, and you must close the window before Internet Explorer begins responding again. This problem does not occur in the compiled application.

Figure 8-3 shows a DHTML page created in Visual Basic using the Asynchronous Animation control created in Chapter 6, "ActiveX Controls," the Common Dialog control, and the HTML Button control. The sample can be found on the companion CD-ROM in the project dhAni.VBP.

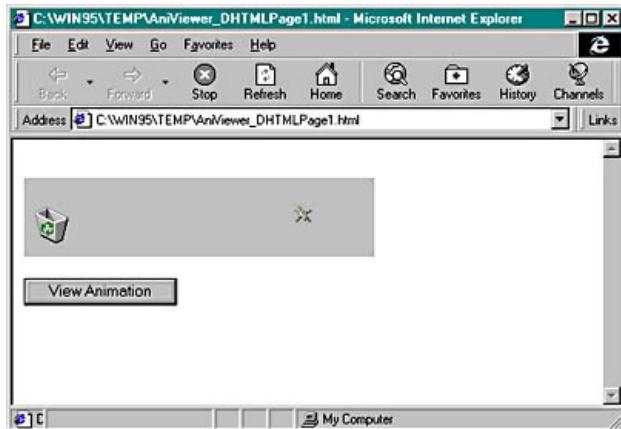


Figure 8-3. The Animation Viewer displays Windows animations in a Web page using DHTML.

I have provided a sample in Figure 8-3 that resembles the sample in the preceding section to demonstrate the differences between using controls and writing code in HTML and DHTML. Even though the resulting pages look identical in the browser, the Visual Basic code used in dhAni.VBP is somewhat different from the VBScript code in htAni.HTM. The DHTML page allows variable declarations and parameter types, and requires control references to be qualified with the Document object, as shown in the following code:

```
'Displays a Windows animation on the Web page
Private Function butView_onclick() As Boolean
    'Make sure objects exist
    If IsNull(dlgFile) _
        Or IsNull(asyncAnimation) Then Exit Function
    Dim strFilename As String
    'Display the Open File dialog box
    Document.dlgFile.ShowOpen
    'Get the file name
    strFilename = Document.dlgFile.FileName

    'If a file was selectedDear John, How Do I...
    If strFilename <> "" Then
        'Dear John, How Do I... show it in the AsyncAni control
        Document.asyncAnimation.AVIFile = _
            strFilename
    End If
End Function

Private Sub asyncAnimation_FileError(Number As Long, _
    Description As String)
    'Display a message with the error
    MsgBox "File is not a valid animation.", _
        vbOKOnly And vbCritical, "Error " & Number
End Sub
```

You might've noticed that the onclick event procedure for the butView button is a Function rather than a Sub. In DHTML, event procedures use return values to enable their default actions. For instance, if you write code for an onclick event procedure for a hyperlink, be sure to return True if you want the browser to go to the link after processing your code. Otherwise, the browser will run your code but not go to the link. The following code demonstrates this:

```
Private Function lnkHome_onclick() As Boolean
    Dim intGo As Integer
    'Get a response
    intGo = MsgBox("Do you want to go home?", vbYesNo)
    'If the user chose yesDear John, How Do I...
    If intGo = vbYes Then
        'Dear John, How Do I... enable the default action (go to link)
        lnkHome_onclick = True
    Else
```

```
'Dear John, How Do I... cancel the default action (don't go to link)
lnkHome_onclick = False
End If
End Function
```

For more information on DHTML, such as how to include multiple pages in a DHTML application, see Chapter 9, "[Creating Internet Applications](#)."

Dear John, How Do I... Create ActiveX Documents?

ActiveX documents are made up of Visual Basic user documents, forms, code modules, and any other components you choose to add. A user document is a unique type of Visual Basic container designed to be displayed in a browser.

NOTE

ActiveX documents aren't covered by World Wide Web Consortium (W3C) standards but are part of Microsoft's ActiveX component strategy which is a de facto software industry standard that evolved from object linking and embedding (OLE). ActiveX documents are supported by Internet Explorer, but not by Netscape Navigator.

User documents are very similar to forms. All the Visual Basic intrinsic controls are available, and you add them to your user document in the same way you would add them to a form. You start to encounter differences, however, when writing code. Since user documents load and unload at the behest of the browser, you need to remember the following points:

- References between objects should wait until all objects are loaded. For instance, a user might click on a button that refers to a control that is still downloading. Use the IsNull function to test whether an object exists before referring to it.
- Data on other user documents are not automatically available. You need to store data items that are shared across user documents in global variables or in a shared object.

Figure 8-4 shows the now familiar Asynchronous Animation control running on an ActiveX document in the browser. The ActiveX document looks like the HTML and DHTML pages created earlier. The sample can be found on the companion CD-ROM in the project axAni.VBP.

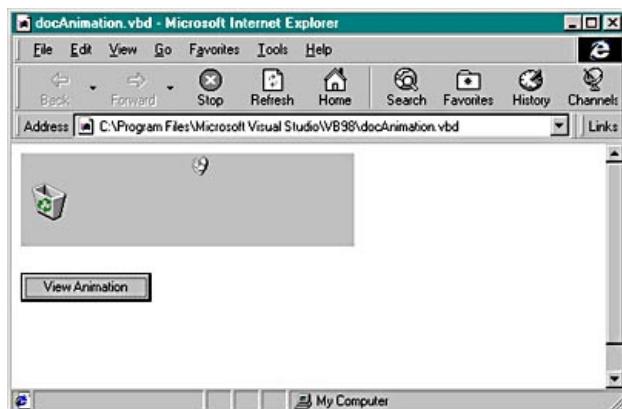


Figure 8-4. The Animation Viewer displays Windows animations in a Web page using an ActiveX document. It's déjà vu!

The code for the user document, shown below, would be identical to a sample created using a form.

```
'Displays a Windows animation on the Web page
Sub cmdView_click()
    'Make sure objects exist
    If IsNull(dlgFile) Or IsNull(asyncAnimation) Then Exit Sub
    Dim strFilename As String
    'Display the Open File dialog box
    dlgFile.ShowOpen
    'Get the file name
    strFilename = dlgFile.FileName
    'If a file was selected
    If strFilename <> "" Then
        'Dear John, How Do I...
        'Dear John, How Do I... show it in the AsyncAni control
        asyncAnimation.AVIFile = strFilename
```

```
End If
End Sub

Private Sub asyncAnimation_FileError(Number As Long, _
    Description As String)
    'Display a message with the error
    MsgBox "File is not a valid animation.", vbOKOnly And vbCritical, _
        "Error " & Number
End Sub
```

ActiveX documents are certainly easier for Visual Basic programmers to create than applications using HTML or DHTML; however, it is harder to maintain the content in ActiveX documents. HTML and DHTML are better tools for publishing material than ActiveX documents since they are supported by a number of different text editors and don't have to be recompiled if their contents change. The relative advantages of each application type is discussed in Chapter 9, "[Creating Internet Applications.](#)"

Dear John, How Do I... Create Webclasses?

A webclass is simply a class that manages the creation and destruction of objects used by an IIS application. Visual Basic automatically creates a webclass when you start a new IIS application.

NOTE

To use webclasses, you must have Internet Information Server (IIS) or Personal Web Server (PWS) installed. References to IIS in this section apply equally to PWS.

You can easily modify the default webclass to write HTML strings to the browser. For example, the following WebClass_Start event procedure creates a Dice object and displays the results of two rolls:

```
Private Sub WebClass_Start()
    Dim sQuote As String
    `Create a Dice object
    Dim diceObject As New Dice
    `Define quote character for building HTML strings
    sQuote = Chr$(34)
    `Set counter for this page
    Session("Tries") = Session("Tries") + 1
    `Write a reply to the user
    With Response
        .Write "<HTML>"
        .Write "<body>"
        .Write "<h1><font face=" & sQuote & "Arial" & sQuote &
            ">Chance -- Try " & Session("Tries") & "</font></h1>"
        .Write "<p>You rolled a " & diceObject.Roll & _
            " and a " & diceObject.Roll & "."
        .Write "</p>"
        .Write "<p>Refresh or reload to roll again.</p>"
        .Write "</body>"
        .Write "</html>"
    End With
End Sub
```

When you run the Chance sample (Chance.VBP), Visual Basic sets up a virtual directory for your project in IIS, creates an Active Server Page (ASP) that references the webclass, and then executes the ASP. Internet Explorer then starts and displays the result shown in Figure 8-5.

The Chance sample also illustrates use of the Session object to save a property. The following line creates a Tries property in the Session object and increments it by one:

```
Session("Tries") = Session("Tries") + 1
```

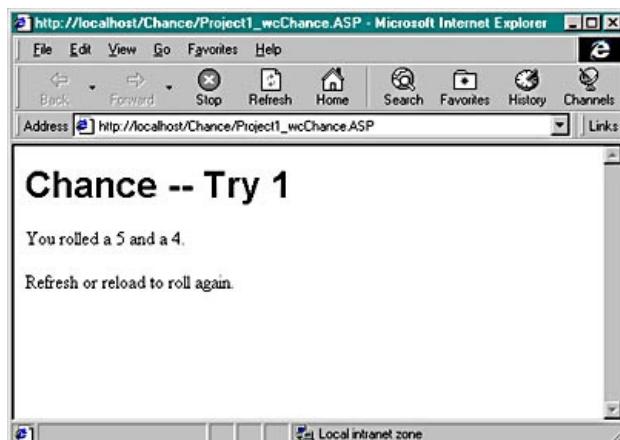


Figure 8-5. Internet Explorer rolls the dice every time you click Refresh.

To retain this setting between refreshes, the webclass's StateManagement property is set to wcRetainInstance. If you change it to wcNoState, the webclass will not retain Session object information and the page will always display 1 as the number of tries.

The Chance sample uses the Response object's Write method to send HTML to the browser. This is a quick way to demonstrate a webclass but is not the best way to display a lot of content or to update content. You can solve these problems by storing your content in separate HTML template files. That technique is shown in Chapter 9, "[Creating Internet Applications](#)."

Chapter Nine

Creating Internet Applications

Visual Basic can create three different types of applications to run in a browser over the Internet:

- Dynamic HTML (DHTML) applications are downloaded over the Internet to run on the user's machine. DHTML applications use the HTML version 4.0 controls, hyperlinks, and formatting to accomplish their tasks.
- Internet Information Server (IIS) applications run on the Internet server and send and receive data from the user's machine. IIS applications use the IIS object model and ActiveX objects installed on the Internet server.
- ActiveX document applications are downloaded over the Internet to run on the user's machine. ActiveX documents use the standard Visual Basic controls and other ActiveX objects and provide complete control of the browser, including the ability to add menus and control the appearance of the browser window.

This chapter shows you how to create multi-page applications using each of these different technologies. I use the same basic sample, a college cost calculator, to demonstrate each of the application styles. The different versions of the college cost calculator can be found on the companion CD-ROM.

Dear John, How Do I... Choose an Application Type?

The large number of ways to create Internet applications arises out of the evolution of the Internet. Microsoft, Netscape, Sun Microsystems, and other companies have released competing tools that each seek to become the standard for Internet development.

So far, there is no clear single standard. Instead you must choose a development approach based on your current needs. Visual Basic provides four different approaches to creating Internet-based applications: simple HTML using ActiveX controls and VBScript, DHTML, ActiveX documents, and IIS applications. Choose the approach that best fits your needs based on the following considerations:

Target browser If your application needs to work on Netscape Navigator as well as Internet Explorer (IE), you'll need to create an IIS application. Netscape doesn't support ActiveX objects in HTML, VBScript, Microsoft's implementation of DHTML, or ActiveX documents.

Location of processing IIS applications are designed for server-side processing. This is ideal for accessing data that resides on the server but can result in performance problems on very busy sites. ActiveX documents, DHTML, and HTML containing ActiveX objects execute on the client's machine. Components must be downloaded once before running.

Ease of maintenance Applications that present a lot of text should use IIS, DHTML, or HTML. These application types allow you to place content in separate files without recompiling and redistributing executable components. ActiveX documents must be recompiled and redistributed whenever their content changes.

Control of client ActiveX documents allow you to control the appearance of the browser and permit unfettered access to the user's file system. IIS, DHTML, and HTML applications can't modify the browser and have limited file access.

Safety Any executable component, such as an ActiveX document or an ActiveX control, can be digitally signed to certify that its author claims it won't wreak havoc on the client's machine. However, users may be reluctant to accept components from unfamiliar authors. IIS applications don't raise this concern since their executable portions run on the server, not the client's machine.

Future support and standards Microsoft and Netscape currently support incompatible forms of DHTML. This will probably be rectified when Netscape releases Navigator 5.0. Navigator 5.0 also promises to support ActiveX components.

If you are creating an Internet application to be deployed on the World Wide Web, your only real choice is an IIS application because Netscape Navigator doesn't support other application types. If you are creating an application that runs over a corporate intranet, you can select from any of these technologies since in this case it's much easier to dictate the browser that clients will use.

Internet Explorer (IE) and Netscape Navigator should become more compatible with their next versions, so some of these points may become less important. However, it is difficult to assess compatibility without seeing actual versions of the products.

Dear John, How Do I... Create a DHTML Application?

A DHTML application project contains one DHTML designer for each HTML page in the project. The project can also include class modules, code modules, forms, and other executable components. When compiled, the executable components reside in a DLL file that is referenced from each HTML file in the application. This keeps the content separate from the executable components. Figure 9-1 shows the Future Shock application (DHFShock.VBP) at design time.

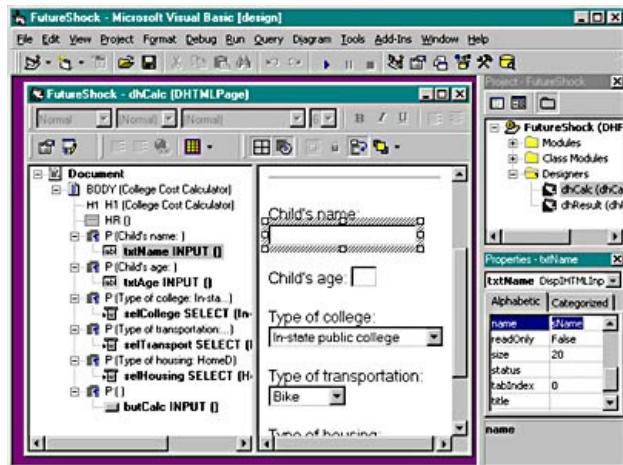


Figure 9-1. Each DHTML designer handles one HTML page.

The DHTML designer provides very limited editing capabilities. To use other editing tools, you'll want to associate the designer with an external HTML file. To do this, select IDHTMLPageDesigner in the Properties window and specify an existing HTML file in the SourceFile property. After Visual Basic loads the HTML file, you can edit it by clicking the Launch Editor toolbar button.

Getting Input Using DHTML

To make an element on a DHTML page programmable, assign it an ID property. Elements that provide events must have an ID before those events will appear in Visual Basic. Each of the controls on the dhCalc designer shown in Figure 9-1 has an ID as shown by the following HTML code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<HEAD>
<META content="text/html; charset=iso-8859-1" http-equiv=Content-Type><TITLE>Future Shock!</TITLE>
<META content='MSHTML 4.72.2106.6' name=GENERATOR>
</HEAD>
<BODY bgColor=#ffffff>
<H1 id="">College Cost Calculator</H1>
<HR>
<P>Child's name:<br>
<INPUT id=txtName maxLength=20 name=sName>
<P>Child's age:<br>
<INPUT id=txtAge maxLength=2 name=sAge size=2>
<P>Type of college: <SELECT id=selCollege name=iCollege>
    <OPTION selected value=0>In-state public college
    <OPTION value=1>Out-of-state public college
    <OPTION value=2>Private college</SELECT>
<P>Type of transportation: <SELECT id=selTransport name=iTransport>
    <OPTION selected value=0>Bike
    <OPTION value=1>Bus
    <OPTION value=2>Used car
    <OPTION value=3>New car</SELECT>
<P>Type of housing: <SELECT id=selHousing name=iHousing>
    <OPTION selected value=0>Home
    <OPTION value=1>Dorm
    <OPTION value=2>Apartment</SELECT>
```

```
<P><INPUT id=butCalc type=button value=Calculate> </P></BODY></HTML>
```

The IDs enable the Calculate button to respond to click events and allow you to retrieve the values entered in the text and selection boxes on the page. The onclick event procedure shown below retrieves data from the dhCalc page and calculates college costs using the Child object:

```
Private Function butCalc_onclick() As Boolean
    Dim vntPayment, vntLumpSum, vntTotalCost
    Dim chdScholar As FutureShock.Child
    'Check fields for valid data
    If Len(txtName.Value) = 0 Or Len(txtAge.Value) = 0 Then
        BaseWindow.alert "You must enter name and age information."
        Exit Function
    End If
    'Create the Child object
    Set chdScholar = New FutureShock.Child
    'Set object properties
    chdScholar.Name = txtName.Value
    chdScholar.Age = txtAge.Value
    chdScholar.College = selCollege.Value
    chdScholar.Transport = selTransport.Value
    chdScholar.Housing = selHousing.Value
    'Calculate cost
    chdScholar.CollegeCost vntPayment, vntLumpSum, vntTotalCost
    'Set state properties so next page can get data
    PutProperty BaseWindow.Document, "Name", txtName.Value
    PutProperty BaseWindow.Document, "Payment", vntPayment
    PutProperty BaseWindow.Document, "LumpSum", vntLumpSum
    PutProperty BaseWindow.Document, "TotalCost", vntTotalCost
    'Display the results page
    BaseWindow.Navigate "dhResult.htm"
End Function
```

Displaying Results Using DHTML

Once you get the data and perform your calculations, you'll want to display the results. How you refer to other pages in your DHTML application depends on how you store the files in your project. If you store the HTML source in an external HTML file, you use that filename when you want to display that page.

For instance, the FutureShock application stores the source for the dhResult page in the source file dhResult.HTM. This makes it easier to edit the file using an external HTML editor. To display that page, the butCalc_onclick event procedure uses the following line of code:

```
BaseWindow.Navigate "dhResult.htm"
```

If the HTML source was stored in a designer file (.DSR), you must include the project name and an underscore (_) before the filename. The code below shows the onclick event procedures for two simple DHTML pages that navigate back and forth between each other.

```
'DHTMLPage1 code
Private Function Button1_onclick() As Boolean
    BaseWindow.Navigate "DHTMLProject_DHTMLPage2.HTML"
End Function

'DHTMLPage2 code
Private Function Button2_onclick() As Boolean
    BaseWindow.Navigate "DHTMLProject_DHTMLPage1.HTML"
End Function
```

Be sure to use the PutProperty method to save any data you want to be available to the next page before you call the Navigate method. After a page is no longer displayed, you can't get at its elements or any of the data stored in its local variables. In the FutureShock application, the modDHTML code module contains the definitions for the PutProperty and GetProperty methods. This module is created

automatically when you start a new DHTML project.

```
`PutProperty:  
`  Store information in a cookie by calling this  
`  function.  
`  The required inputs are the named Property  
`  and the value of the property you would like to store.  
  
`Optional inputs are:  
`  expires : specifies a date that defines the valid  
`            life time of the property. Once the expiration  
`            date has been reached, the property will no  
`            longer be stored or given out.  
  
Public Sub PutProperty(objDocument As HTMLDocument, _  
    strName As String, vntValue As Variant, _  
    Optional Expires As Date)  
  
    objDocument.cookie = strName & "=" & CStr(vntValue) & _  
        IIf(CLng(Expires) = 0, "", "; expires=" & _  
            Format(CStr(Expires), "ddd, dd-mmm-yy hh:mm:ss") & _  
            " GMT") ` & _  
  
End Sub  
  
`GetProperty:  
`  Retrieve the value of a property by calling this  
`  function. The required input is the named Property,  
`  and the return value of the function is the current value  
  
`  of the property. If the property cannot be found or has  
`  expired, then the return value will be an empty string.  
  
Public Function GetProperty(objDocument As HTMLDocument, _  
    strName As String) As Variant  
    Dim aryCookies() As String  
    Dim strCookie As Variant  
    On Local Error GoTo NextCookie  
  
        `Split the document cookie object into an array of cookies  
        aryCookies = Split(objDocument.cookie, ";")  
        For Each strCookie In aryCookies  
            If Trim(VBA.Left(strCookie, InStr(strCookie, "=") - 1)) _  
                = Trim(strName) Then  
                GetProperty = Trim(Mid(strCookie, _  
                    InStr(strCookie, "=") + 1))  
                Exit Function  
            End If  
        NextCookie:  
            Err = 0  
        Next strCookie  
    End Function
```

The dhResult page uses **** tags to mark the places where the calculated data are displayed. Use **** tags when you want to display results in line with other text. Use **<DIV>** tags to display results on separate lines. The following HTML code shows the definition for dhResult:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">  
<HTML>  
<HEAD>  
<META content="text/html; charset=iso-8859-1" http-equiv=Content-Type><TITLE>Future Shock!</TITLE>  
<META content='MSHTML 4.72.2106.6' name=GENERATOR>  
</HEAD>
```

```
<BODY bgColor="#ffffff id="">
<H1>College Cost Calculator</H1>
<HR>
<P>Amount you need to invest today to pay for <SPAN id=spnName1>Name</
    SPAN> to
go to college: <SPAN id=spnLumpSum>LumpSum</SPAN>
<P>Amount you need to save each month to send <SPAN id=spnName2>Name</
    SPAN> to
college: <SPAN id=spnPayment>Payment</SPAN>
<P></P>
<P>Total amount you need to have by the time <SPAN id=spnName3>Name</
    SPAN> is
18: <SPAN id=spnTotalCost>TotalCost</SPAN>
<P><INPUT id=butBack type=button value="Go Back"> </P></BODY></HTML>
```

You perform the replacements in the dhResult Document object's onreadystatechange event. The onreadystatechange event occurs after the elements on a page are created, but before they are displayed. Trying to use the DHTMLPage Load or Initialize events results in an error, since the page elements don't exist at those points. Use the InnerText property to perform the replacements, as shown here:

```
`This event occurs just before displaying the page
Private Sub Document_onreadystatechange()
    Dim vntName, vntPayment, vntTotalCost, vntLumpSum
    `Retrieve property values from modDHTML
    vntName = GetProperty(BaseWindow.Document, "Name")
    vntPayment = GetProperty(BaseWindow.Document, "Payment")
    vntTotalCost = GetProperty(BaseWindow.Document, "TotalCost")
    vntLumpSum = GetProperty(BaseWindow.Document, "LumpSum")
    `Display the value with <SPAN> tags
    With BaseWindow.Document
        .All("spnName1").innerText = vntName
        .All("spnName2").innerText = vntName
        .All("spnName3").innerText = vntName
        .All("spnPayment").innerText = vntPayment
        .All("spnTotalCost").innerText = vntTotalCost
        .All("spnLumpSum").innerText = vntLumpSum
    End With
End Sub
```

Finally, the button on the bottom of the results page allows you to navigate back to the dhCalc page using the history object's back method, as shown here:

```
Private Function butBack_onclick() As Boolean
    `Go back to previous page
    BaseWindow.history.back
End Function
```

Dear John, How Do I... Create IIS Applications?

IIS applications store content in Active Server Pages (ASP). Active Server Pages are standard text files with an ASP extension that contain HTML and scripting code. ASP files are processed by the Internet server at runtime to send HTML to client browsers. The application's executable components are stored in a DLL file that runs only on the Internet server. The server does all the processing, and client browsers send and receive only HTML-format data. Figure 9-2 shows this structure in operation.

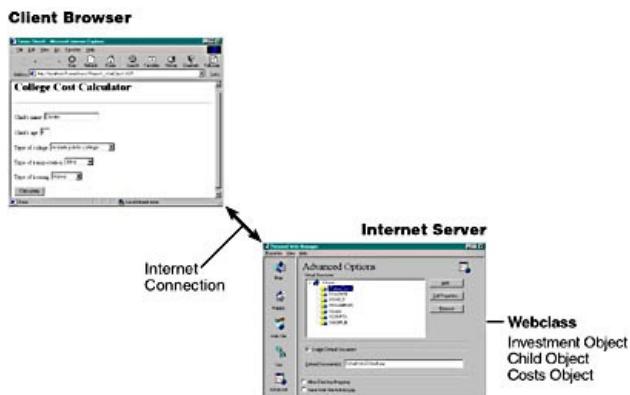


Figure 9-2. Webclasses manage objects used by ASP in providing responses to browsers.

NOTE

To create an IIS application, you need to be running Windows NT with IIS installed or Windows 95 with Personal Web Server (PWS) installed.

At design-time, IIS applications consist of one webclass designer and any number of HTML template files, class modules, and code modules. IIS applications do not usually contain forms, since their display would occur on the server and not on the client machine.

To create an IIS application, start a new Visual Basic project and double-click the IIS Application icon in the New Project dialog box. Visual Basic creates a new webclass designer that includes the following code:

```
Private Sub WebClass_Start()
    Dim sQuote As String
    sQuote = Chr$(34)
    'Write a reply to the user
    With Response
        .Write "<HTML>"
        .Write "<body>

            .Write "<h1><font face=" & sQuote & "Arial" & sQuote & _
                ">WebClass1's Starting Page</font></h1>""
            .Write "<p>This response was created in the Start " & _
                "event of WebClass1.</p>""
            .Write "</body>"
            .Write "</html>"
    End With
End Sub
```

This code uses the ASP Response object to write a heading and some text to the browser. If you run the code, Visual Basic takes the following actions:

1. Creates a virtual directory for the project directory if it does not already exist.
2. Creates an ASP file for the webclass.
3. Processes the ASP file and sends the result to the client browser for display.

The ASP that Visual Basic generates makes sure the webclass manager is running, gets an instance of the webclass runtime, then creates an instance of your webclass. The following ASP document was generated for the webclass code shown earlier:

```
<%
Server.ScriptTimeout=600
Response.Buffer=True
Response.Expires=0

If (VarType(Application("WC~WebClassManager")) = 0) Then
    Application.Lock
    If (VarType(Application("WC~WebClassManager")) = 0) Then
        Set Application("WC~WebClassManager") =
            Server.CreateObject("WebClassRuntime.WebClassManager")
    End If
    Application.UnLock
End If

Application("WC~WebClassManager").ProcessNoStateWebClass =
    "Project1.WebClass1", _
    Server, _
    Application, _
    Session, _
    Request, _
    Response
%>
```

It's important to notice that the generated ASP file doesn't write any text to the browser. All that is done by the executable code in the webclass. This is where webclasses differ from the way Visual Basic 5 used ASP to create IIS applications.

- In Visual Basic 5, you wrote your interface in the ASP file using HTML statements, and referenced server objects using VBScript statements within the same ASP file.
- With Visual Basic 6, you can write your interface in the webclass using the Write and WriteTemplate methods to send HTML data to the browser. The server objects are readily available within the webclass using standard Visual Basic language.

The following sections show how to use a webclass to get information from a user, do some processing, and send a response.

Displaying HTML Templates

The first thing you notice in the code that Visual Basic generates for a new webclass is the way the code uses the Response object's Write method to send HTML to the browser. This works fine for a simple example, but it soon becomes tedious. Fortunately, Visual Basic lets you add HTM files to your webclass as HTML template Web items.

To add an HTM file to a webclass, click the Add HTML Template WebItem button in the webclass designer. Figure 9-3 shows a webclass containing two HTML templates.

The htmCalc template shown in Figure 9-3 displays a data entry form in the browser. The HTML for htmCalc is as follows:

```
<HTML>
<HEAD><TITLE>Future Shock!</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>College Cost Calculator</H1>
<HR>
    <FORM METHOD="POST" NAME="frmCalc">
        <P>
            Child's name: <INPUT TYPE=TEXT SIZE=20 MAXLENGTH=20
                NAME="strName">
```

```

<P>
Child's age: <INPUT TYPE=TEXT SIZE=2 MAXLENGTH=2
    NAME="strAge">
<P>
Type of college:
<SELECT NAME="optCollege">
    <OPTION VALUE=0>In-state public college
    <OPTION VALUE=1>Out-of-state public college
    <OPTION VALUE=2>Private college
</SELECT>
<P>
Type of transportation:
<SELECT NAME="optTransport">
    <OPTION VALUE=0>Bike
    <OPTION VALUE=1>Bus
    <OPTION VALUE=2>Used car
    <OPTION VALUE=3>New car
</SELECT>
<P>
Type of housing:
<SELECT NAME="optHousing">
    <OPTION VALUE=0>Home
    <OPTION VALUE=1>Dorm
    <OPTION VALUE=2>Apartment
</SELECT>
<P>
<INPUT TYPE="SUBMIT" VALUE="Calculate" NAME="butCalc">
</FORM>
</BODY>
</HTML>

```

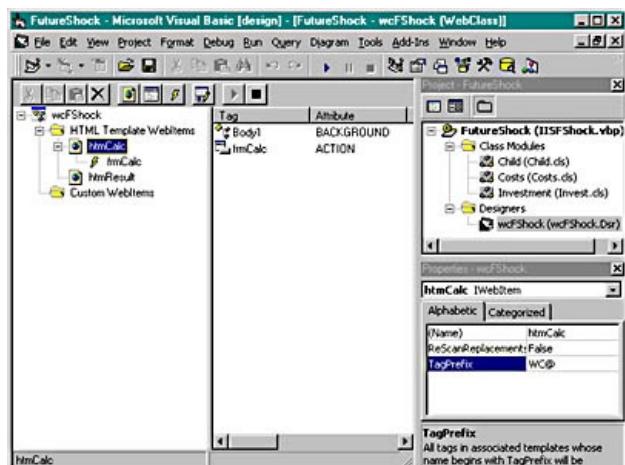


Figure 9-3. The webclass designer can include HTML pages, images, and other data.

To display this HTML template from a webclass, use the template's WriteTemplate method. For example, to display htmCalc when the webclass starts, replace the default WebClass_Start event procedure with the following code:

```

Private Sub WebClass_Start()
    'Display the calculator page
    htmCalc.WriteTemplate
End Sub

```

The preceding code displays the College Cost Calculator form as shown in Figure 9-4.

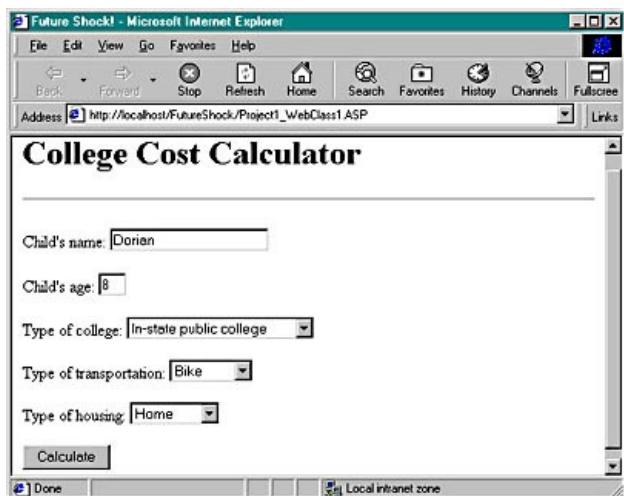


Figure 9-4. The htmCalc template displays an HTML form used to retrieve data from the user.

Getting Input Using IIS

To retrieve the data entered on the htmCalc template, you need to associate an event procedure with the HTML form's default action and add code to access the fields on the form. To associate an event procedure with an HTML form's default action, follow these steps:

1. In the left pane of the webclass designer window, select the HTML template containing the form.
2. In the right pane of the webclass designer window, right click the name of the form (in this case, frmCalc) to display the pop-up menu.
3. Select View Code from the pop-up menu. Visual Basic creates an event procedure if one does not already exist and associates that procedure with the form's default event.

NOTE

Webclasses respond to events that the browser sends the server, such as submitting a form or requesting a page. Other events, such as the HTML Button object's onclick event, occur only on the client machine. If you want to respond to client events you need to add VBScript event procedures to the page.

To access the data on the form, use the Request object's Form collection. Each item in the Form collection corresponds to an item on the form through the item's NAME attribute in the HTML template. For example, the following code gets the data from each form field, calculates college costs using that data, and then displays the htmResult template:

```
Private Sub htmCalc_frmCalc()
`Check for valid data
If Len(Request.Form("strName")) = 0 Or _
Len(Request.Form("strAge")) = 0 Then
    Response.Write "<p>Name and age are required.</p>"
    Exit Sub
End If
`Create an object variable
Set mchdScholar = New Child
`Get form data
With Request
    mvntName = .Form("strName")
    mchdScholar.Name = .Form("strName")
    mchdScholar.Age = .Form("strAge")
    mchdScholar.College = .Form("optCollege")
```

```

mchdScholar.Transport = .Form("optTransport")
mchdScholar.Housing = .Form("optHousing")
End With
`Calculate costs
mchdScholar.CollegeCost mvntPayment, mvntLumpSum, mvntTotalCost
`Write result; triggers ReplaceToken event
htmResult.WriteTemplate
End Sub

```

NOTE

While the DHTML applications use the ID attribute to identify items in code, webclasses use the NAME attribute. This is a subtle and potentially confusing difference.

Displaying Results Using IIS

Once you've gotten your data and processed it, you'll want to display the results through the browser. The easiest way to do this is by building a response in code using the Response object's Write method. For example, the following code displays the total cost of a college education as calculated in the preceding section:

```

Private Sub htmCalc_frmCalc()
    `Code omitted here (same as previous section)Dear John, How Do I...
    Dim sQuote As String
    sQuote = Chr$(34)
    `Write result using Response.Write
    With Response
        .Write "<HTML>"
        .Write "<body>"
        .Write "<h1><font face=" & sQuote & "Arial" & sQuote & _
            ">Results</font></h1>""
        .Write "<p>The cost of sending " & mvntName &
            " to college is: " & mvntTotalCost & "</p>""
        .Write "</body>"
        .Write "</html>"
    End With
End Sub

```

A better way to display large amounts of text is to display an HTML template using the template's WriteTemplate method. You replace variable data within a template by tagging the variables with <WC@> and </WC@> tags, as shown here:

```

<HTML>
<HEAD><TITLE>Future Shock!</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Results</H1>
<HR>
Amount you need to invest today to pay for <WC@>strName</WC@> to
go to college: <WC@>LumpSum</WC@>.
<P>
Amount you need to save each month to send <WC@>strName</WC@> to
college: <WC@>Payment</WC@>.
<P>
Total amount you need to have by the time <WC@>strName</
WC@> is 18: <WC@>TotalCost</WC@>.
<p> <WC@>htmCalc</WC@>Go Back.</a>
</FORM>
</BODY>
</HTML>

```

Before Visual Basic writes the template to the browser, it checks the template for replacement tags. If it encounters any, Visual Basic triggers the ProcessTag event. The following code shows how to use a Select Case statement to replace variables in the preceding HTML template with the appropriate data at runtime:

```

Private Sub htmResult_ProcessTag(ByVal TagName As String, _
    TagContents As String, SendTags As Boolean)
    'Remove extra spaces and carriage returns—seems to be
    'a bug in webclasses
    TagContents = Trim(TagContents)
    TagContents = Replace(TagContents, vbCrLf, "")
    'Replace strings in template
    Select Case TagContents
        Case "strName"
            TagContents = mvntName
        Case "Payment"
            TagContents = mvntPayment
        Case "LumpSum"
            TagContents = mvntLumpSum
        Case "TotalCost"
            TagContents = mvntTotalCost
        Case "htmCalc"
            'Create a link back to htmCalc
            TagContents = "<a href=" & GetURL(htmCalc) & ">"
        Case Else
    End Select
End Sub

'Return the URL for a WebItem
Function GetURL(htmObject As Object) As String
    Dim strURL As String
    'Get the URL (includes extra stuff)
    strURL = urlfor(htmObject)
    'Strip off the non-URL text
    strURL = Mid(strURL, 1, InStr(1, strURL, "?") - 1)
    'Return the result
    GetURL = strURL
End Function

```

Visual Basic triggers the ProcessTag event each time it encounters a replacement tag. You can step through the code to see how this works. Once it reaches the end of the template, it displays the modified HTML data as shown in Figure 9-5.

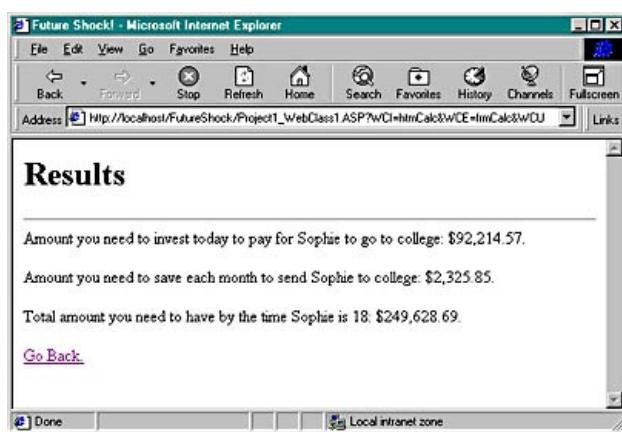


Figure 9-5. Sending a kid to college (or writing this program?) is a lot of hard work.

Dear John, How Do I... Create ActiveX Document Applications?

ActiveX document applications use Visual Basic user documents to present information and respond to events. User documents are a lot like forms; however, they differ significantly from forms in these ways:

- Data in user documents is not readily available to other parts of your application. You cannot refer to the value of a property in a user document from outside that document.
- Applications based on user documents need IE to run. You can't debug them directly within Visual Basic alone.
- Compiled applications must start from an HTM file that loads the code components of the application before displaying Visual Basic Document (VBD) files. You can't start the compiled executable (EXE) file or view the VBD files from IE without this HTM file.

The following sections illustrate these differences.

Getting Input Using ActiveX Documents

The Future Shock application (AXFShock.VBP) calculates the cost of sending a child to college. It contains the two user documents shown in Figure 9-6. The udCalc document (udCalc.DOB) gathers information used to project future college expenses. The udResult document (udResult.DOB) presents the results of the calculation.

In order to share information, the two user documents rely on the global object gchdScholar, which is declared in the code module DECLARES.BAS, as shown here:

```
'Global object that shares data between
`user documents
Public gchdScholar As New Child
```

The gchdScholar object is an instance of the Child class (defined in CHILD.CLS), which includes properties for each of the data fields in the udCalc user document. The user document udCalc sets the values of these properties and then navigates to the udResult user document when the user clicks the Calculate button.

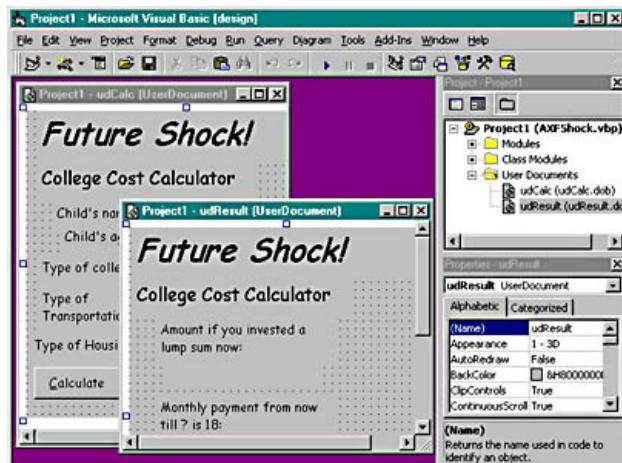


Figure 9-6. User documents to collect data and display results in Internet Explorer.

Navigating to an address from the Visual Basic debugging environment requires code different from that in the compiled version of an Internet application. During debugging, Visual Basic stores a temporary copy of each compiled user document—Visual Basic Document (VBD) file—in the Visual Basic home directory. After the application becomes final, the VBD files are installed on the Internet server. Future Shock uses the #IfDear John, How Do I... ThenDear John, How Do I... #Else directive to handle this difference.

The following code shows how the udCalc user document initializes its controls, sets the gchdScholar properties, and navigates to the udResult user document:

```

Option Explicit
#Const DEBUG_MODE = True

Private Sub UserDocument_Initialize()
    'Initialize list boxes
    cmbCollege.AddItem "In-state public college", chdState
    cmbCollege.AddItem "Out-of-state public college", _
        chdOutOfState
    cmbCollege.AddItem "Private college", chdPrivate
    cmbTransport.AddItem "Bike", chdBike
    cmbTransport.AddItem "Bus", chdBus
    cmbTransport.AddItem "Used car", chdUsedCar
    cmbTransport.AddItem "New car", chdNewCar
    cmbHousing.AddItem "Home", chdHome
    cmbHousing.AddItem "Dormitory", chdDorm
    cmbHousing.AddItem "Apartment", chdApartment
    cmbCollege.ListIndex = 0
    cmbTransport.ListIndex = 0
    cmbHousing.ListIndex = 0
End Sub

`Display Costs Web page
Private Sub cmdCalculate_Click()
    'Check for valid data
    If Len(txtName) = 0 Or Len(txtAge) = 0 Then
        MsgBox "Name and age are required."
        Exit Sub
    End If
    'Set Child properties
    gchdScholar.Age = txtAge
    gchdScholar.Name = txtName
    gchdScholar.Transport = cmbTransport.ListIndex
    gchdScholar.Housing = cmbHousing.ListIndex
    gchdScholar.College = cmbCollege.ListIndex

    'Navigating differs in debug and release versions
    #If DEBUG_MODE Then
        'Use local address during debugging
        Hyperlink.NavigateTo "C:\Program Files\" & _
            "Microsoft Visual Studio\VB98\udResult.VBD"
    #Else
        'Use network address at release
        Hyperlink.NavigateTo "http://\6.6.6.6\" & _
            "vb6dev\chap09\AXFShock\udResult.VBD"
    #End If
End Sub

```

Each user document includes a built-in `Hyperlink` object that you use to navigate to Uniform Resource Locators (URLs) and to go forward or backward in IE's cache of documents. The `NavigateTo` method is roughly equivalent to a form's `Show` method.

Displaying Results Using ActiveX Documents

When IE first navigates to the `udResult` document, the `Initialize` event is triggered. The code for the `Initialize` event procedure adds the name of the child to one of the labels on the user document. The `Initialize` event happens only once per IE session.

Next, the `Show` event occurs. The code for the user document `Show` event procedure tells the `Child` object to calculate his or her college costs (smart kid!) and then displays the results in labels on the document. The `Show` event is triggered every time the user document is displayed.

The code for the `udResult` user document's `UserDocument_Initialize`, `UserDocument_Show`, and `cmdBack_Click` event procedures is shown here:

```
Option Explicit
```

```
`Use the Initialize event to initialize information
```

```
`that will stay the same throughout session
```

```
Private Sub UserDocument_Initialize()
```

```
    `Fill in name in label
```

```
    lblPaymentDesc = Left(lblPaymentDesc, _
```

```
        InStr(1, lblPaymentDesc, "?") - 1) & _
```

```
        gchdScholar.Name & _
```

```
        Right(lblPaymentDesc, _
```

```
            Len(lblPaymentDesc) - _
```

```
            InStr(1, lblPaymentDesc, "?"))
```

```
End Sub
```

```
`Use the Show event to update information when the
```

```
`user revisits this page
```

```
Private Sub UserDocument_Show()
```

```
    Dim LumpSum, Payment, TotalCost
```

```
    `Calculate college costs
```

```
    gchdScholar.CollegeCost Payment, LumpSum, TotalCost
```

```
    `Display costs
```

```
    lblLumpSum = LumpSum
```

```
    lblPayment = Payment
```

```
    lblTotalCost = TotalCost
```

```
End Sub
```

```
`Navigate back to the data entry page
```

```
Private Sub cmdBack_Click()
```

```
    Hyperlink.GoBack
```

```
End Sub
```

Actually, I could have put all the Initialize code in the Show event procedure, but I wanted to illustrate the difference. If you click the Back button, enter information for a second child, and then click Calculate, udResult recalculates the costs but doesn't change the name of the child displayed in the second label. This minor bug makes a point: use the Initialize event to set static data in a user document, and use the Show event to set data you want to refresh every time the user displays the page.

The code used in the Child class (CHILD.CLS), the Costs class (COSTS.CLS), and the Investment class (INVEST.CLS) can be found on the companion CD-ROM.

SEE ALSO

-
- "[Dear John, How Do I... Install ActiveX Documents over the Internet?](#)" below to learn how to install Internet applications over a network

Dear John, How Do I... Install ActiveX Documents over the Internet?

ActiveX documents must be installed and registered before they can be displayed in IE. If you try to load an ActiveX document (VBD file) in IE without first installing the document, IE simply assumes that you want to copy the VBD file to your disk and displays the File Download dialog box shown in Figure 9-7 following. If you select Open This File From Its Current Location from this dialog box, the file is copied, and then you are prompted to select an application that will open the file.



Figure 9-7. If you open a VBD file over the Internet, IE assumes you want to copy the file.

Use the Package and Deployment Wizard to create a setup program for your ActiveX documents by following these steps:

1. To run the Package and Deployment Wizard, choose the Microsoft Visual Basic 6.0 program group from the Windows Start button, then choose the Microsoft Visual Basic 6.0 Tools group, and select the Package and Deployment Wizard.
2. Click the Browse button, select the project file (VBP) for your ActiveX document application, and then click Package. The Package and Deployment Wizard checks the project files and if the sources are newer than the compiled files, it asks you if you'd like to recompile the application. Next, the Wizard displays the Packaging Script dialog box.
3. In the Packaging Script dialog box, enter a name for the packaging script and click Next. The Wizard displays the Package Type dialog box.
4. In the Package Type dialog box, select Internet Package and click Next. The Wizard displays the Package Folder dialog box.
5. In the Package Folder dialog box, select the root folder on your Internet server from which users will access your ActiveX document application. Alternatively, you can save the package in a folder on your local machine and move it later. When you click Next, the Wizard displays the Included Files dialog box.
6. In the Included Files dialog box you can click Add to include other files that your ActiveX document application may need. For instance, you may want to include application data files that weren't detected by the Wizard when it scanned the project file in step 2. When you are done, click Next and the Wizard displays the File Source dialog box.
7. The File Source dialog box lets you use the Microsoft servers for the Visual Basic runtime and associated distribution files. If you are deploying your application over an intranet, you'll probably want to select the Download From Alternate Web Site option so that the Visual Basic runtime and OLE Automation files can be copied from the current server or from another location on your network. If your application is being distributed over the Internet, you'll probably want to select the Download From Microsoft Web Site option so that users always get the latest versions of these files. When you click Next, the Wizard displays the Safety Settings dialog box.

8. The Safety Settings dialog box lets you specify whether components are safe for initialization or scripting. For each of the components in the Future Shock application, select Yes in both the Safe For Initialization column and the Safe For Scripting column. When you click Next, the Wizard displays the Finished dialog box.

9. When you click Finish, the Wizard builds the installation and startup files for the application and displays a report of the files created.

Based on the options that you select, the Package and Deployment Wizard generates the types of files listed in the following table. The first three file types are the ones you should copy to your Internet site.

Package and Deployment Wizard File Types

File Type	Description
CAB	Compressed versions of the application's executable and dependent files
HTM	Generated Web page that automatically installs the files from the CAB file on the user's machine and then opens the installed VBD files in the browser
VBD	Compiled user documents
DDF	Project file used by the Package and Deployment Wizard to create the CAB file
INF	Setup information file used to customize the installation of the application
BAT	A batch file used to rebuild the application from the command line, rather than through the Package and Deployment Wizard's graphical user interface

Once you have copied the setup program files to your Internet site, users can run the application by opening the generated HTM file. This file installs and registers the executable components on the user's machine and then displays hyperlinks for each of the ActiveX documents in your application.

You'll probably want to modify the generated HTM file so that your first ActiveX document is displayed when the user opens the page. The following HTML code shows the modifications to the HTM file generated for the Future Shock application. The modifications are shown in boldface type. In order to shorten the code, I've also deleted some comment blocks generated by the Wizard:

```

<HTML>
<HEAD>
<TITLE>AXFShock.CAB</TITLE>
</HEAD>
<BODY>
<OBJECT ID="Child"
CLASSID="CLSID:8EC98958-03B9-11D2-8E90-000000000000"
CODEBASE="AXFShock.CAB#version=1,0,0,0">
</OBJECT>
<OBJECT ID="Costs"
CLASSID="CLSID:8EC9895D-03B9-11D2-8E90-000000000000"
CODEBASE="AXFShock.CAB#version=1,0,0,0">
</OBJECT>

<OBJECT ID="Investment"
CLASSID="CLSID:8EC9895F-03B9-11D2-8E90-000000000000"
CODEBASE="AXFShock.CAB#version=1,0,0,0">
</OBJECT>
<a href=udCalc.VBD>udCalc.VBD</a>
<a href=udResult.VBD>udResult.VBD</a>

<SCRIPT LANGUAGE="VBScript">
Sub Window_OnLoad
    Document.Navigate "udCalc.VBD"
End Sub

```

```
</SCRIPT>
</BODY>
</HTML>
```

The CLASSID attributes in the HTM file are the class IDs of the components used in the Future Shock application. It's important to remember that Visual Basic generates new class IDs every time you compile the application. You'll need to run the Package and Deployment Wizard again each time you change code and recompile.

Once you've created a setup program and copied the HTM, CAB, and VBD files to your Internet site, you can run the application by following these steps:

1. In IE's Address text box, type the URL of your application's HTM file. IE checks to see whether the ActiveX components used by the application are registered on your machine. If they aren't registered, IE attempts to install the required components from the CAB files.
2. Depending on your IE security settings and information gathered by the Package and Deployment Wizard, Internet Explorer may display a warning before each component is installed. If you respond Yes to each warning, Internet Explorer completes the installation and displays the first document in your application.

Dear John, How Do I... Install DHTML Applications over the Internet?

Like ActiveX documents, DHTML applications also must be packaged before they can be run over the Internet. You can follow the same instructions found in the preceding section, "["Dear John, How Do I... Install ActiveX Documents over the Internet?"](#)" to run the Package and Deployment Wizard on your DHTML application.

The Package and Deployment Wizard handles DHTML applications only slightly different than ActiveX documents. The main difference is that the Wizard modifies the HTM files that are created when you compile a DHTML project. The Package and Deployment Wizard changes the first few lines of these files to reference the CAB files created by the Wizard. This causes the appropriate executable files to be installed when the user opens the HTM file over the Internet.

You shouldn't have to modify the Wizard-generated HTM file, as you must for ActiveX document applications.

Dear John, How Do I... Deploy IIS Applications over the Internet?

Since the executable components for an IIS application remain on the server, you don't need to take special steps to install anything over the Internet. Instead, you deploy the compiled application by copying its files to the Internet server from which the application will be accessed.

Compiled IIS applications consist of an Active Server Page (ASP) file, a Global.ASA file, one or more DLL or EXE files, and HTM files. Executable files (DLL and EXE) should not be stored in a public directory since they run only on the server.

Chapter Ten

API Functions

A powerful feature of Visual Basic is its ability to call procedures that reside in dynamic link library (DLL) files, including the application programming interface (API) functions that are provided—and used—by Windows. The Windows API functions and their syntax are described in the Platform SDK online help. Access to the thousands of Windows API functions, as well as other functions contained in DLLs, extends the capabilities of Visual Basic far beyond those of many other programming languages.

In this chapter, I show you a way to declare these functions in an easy-to-read format by using Visual Basic's line continuation character. I also demonstrate a few useful functions to give you a head start in using these API calls. I describe a few simple details and potential gotchas that seem to trip up most of us once or twice as we begin to experiment with API function calling.

Dear John, How Do I... Call API Functions?

To use API functions, you simply declare them in your source code and then call them just as you would any other function in Visual Basic. We'll take a look at these two steps in more detail in the sections that follow.

Declarations

Because API functions are not internal to Visual Basic, you must explicitly declare them before you can use them. The Visual Basic online help covers the exact syntax for the Declare statement, but I've discovered a few tricks that you will probably find useful.

Some API function declarations are quite long. In the past, either you lived with a long declaration or you somehow condensed the declaration so that it would all fit comfortably on one line. For example, the following code shows the standard declaration for the API function GetTempFileName. In versions of Visual Basic prior to Visual Basic 4, this entire declaration was entered on one line in the source code file:

```
Private Declare Function GetTempFileName Lib "kernel32" Alias
"GetTempFileNameA" (ByVal lpszPath As String, ByVal lpPrefixString
As String, ByVal wUnique As Long, ByVal lpTempFileName As String)
As Long
```

Here's a shortened version of this declaration also entered on one line, which makes the declaration more manageable but somewhat less readable:

```
Private Declare Function GetTempFileName& Lib "kernel32" Alias
"GetTempFileNameA" (ByVal Pth$, ByVal Prf$, ByVal Unq&, ByVal Fnm$)
```

Visual Basic lets you format these declarations in another way, keeping the longer, more readable parameter names but using much shorter lines that are visible in a normal-size edit window. The line continuation character is the key to this improved format. The following code shows the same declaration in a style I find easy to read. (You can modify the layout to suit your own style.)

```
Private Declare Function GetTempFileName
Lib "kernel32" Alias "GetTempFileNameA" (
    ByVal lpszPath As String,
    ByVal lpPrefixString As String,
    ByVal wUnique As Long,
    ByVal lpTempFileName As String
) As Long
```

You'll find this style used throughout this book wherever I declare API functions.

32-Bit Function Declarations

Notice in the previous example that the GetTempFileName function name is actually aliased to a function named GetTempFileNameA. In Windows 95, 32-bit function declarations involving string parameters have been renamed from their 16-bit predecessors because they've been rebuilt using 32bit coding specifications, although they still use ANSI strings internally (hence the A). Be aware that in 32-bit versions of Visual Basic, Windows API function names are case sensitive.

NOTE

A third set of functions has been defined for 32-bit Windows NT development in which strings are internally manipulated in the system DLLs as Unicode strings. In this case, the original function names now have the suffix *W* (for *Wide* characters). The Unicode function versions are not supported in Windows 95.

To be sure you get the properly formatted function declaration, you can access the Windows API function declarations in the WIN32API.TXT file, which you'll find in the \Program Files\Microsoft Visual

Studio\Common\Tools\Winapi directory. There are several ways to get the function declarations from this file into your application. You can load the file into WordPad and copy the desired declarations into your Visual Basic applications by hand, making the editing changes mentioned above. Or you might prefer to use the API Viewer Add-In, which comes with Visual Basic, to automate this process.

The API Viewer lets you load a text API file or a database API file and easily browse its contents. Items such as function declarations can be selected, copied to the clipboard, and pasted into Visual Basic. You can start the API Viewer by choosing Microsoft Visual Basic 6.0 Tools\API Text Viewer from the Visual Basic directory on the Windows Start menu or by choosing API Viewer from the Add-Ins menu. To add the API Viewer to the Add-Ins menu, choose Add-In Manager from the Add-Ins menu, select VB 6 API Viewer in the Add-In Manager window, and then click OK.

In Chapter 30, "[Development Tools](#)," I'll show you how to create an add-in to the Visual Basic environment that helps you insert API function declarations.

Strings

There are a couple of gotchas to watch out for when you are passing strings as arguments to API functions. One is that API functions will not create any string space for you. You must create a string space long enough to handle the longest possible string that could be returned before you pass the string to the function. For example, the following code declares the API function GetWindowsDirectory, which returns the path to the Windows directory. The string buffer for the path must be large enough to hold the data returned by GetWindowsDirectory. Before the function is called, the string variable *strWinPath* is built to be 144 bytes in length using the *Space* function, as shown. Failure to build a string parameter to a sufficient length will cause an API function to return no data in the string.

```
Option Explicit
```

```
Private Declare Function GetWindowsDirectory _  
Lib "kernel32" Alias "GetWindowsDirectoryA" ( _  
    ByVal lpBuffer As String, _  
    ByVal nSize As Long _  
) As Long

Private Sub Form_Click()  
    Dim strWinPath As String  
    Dim lngRtn As Long  
    Const MAXWINPATH = 256  
    strWinPath = Space(MAXWINPATH)  
    lngRtn = GetWindowsDirectory(strWinPath, MAXWINPATH)  
    strWinPath = Left(strWinPath, lngRtn) `Truncate at the 0 byte  
    Print strWinPath  
End Sub
```

If you incorporate this code fragment into a program, the Windows directory path, as returned by the GetWindowsDirectory API function, is displayed when the form is clicked. The result is shown in Figure 10-1.



Figure 10-1. The Windows directory path as returned by GetWindowsDirectory.

Also note that returned strings are terminated by a byte of value 0. In the sample code above, the function returns the length of the string, but in many cases when you use API functions you won't know the actual length of the string data unless you look for the 0 byte. If the API function doesn't return the length of the returned string, here's a way to lop off those extra spaces on the string:

```
strWinPath = Left(strWinPath, InStr(strWinPath, Chr(0)) - 1)
```

SEE ALSO

- The APIAddin application in Chapter 30, "[Development Tools](#)," for a demonstration of a tool that selects, copies, and pastes API functions into your source code

Dear John, How Do I... Pass the Address of a Procedure to an API Function?

Passing the address of a Visual Basic procedure to an API function is a trick that's been reserved for real code gurus up to now. Visual Basic includes the `AddressOf` operator, which is used for this C programming technique. In C, this process is known as a *callback*. A callback refers to a Visual Basic procedure that is called while the API function is being executed.

The parameters for the called Visual Basic procedure are determined by the API function. For example, in the following code the `EnumChildWindows` API function calls the Visual Basic `ChildWindowProc` function. The code should be placed in a code module (BAS file) and not in a form or class module. When the project's Startup Object is set to Sub Main and this code is executed, the window handles for the Visual Basic programming environment are displayed in the Immediate window.

```
Option Explicit
```

```
Private Declare Function GetActiveWindow _
Lib "User32" () As Long

Private Declare Function EnumChildWindows _ 
Lib "User32" (
    ByVal hWnd As Long,
    ByVal lpWndProc As Long,
    ByVal lp As Long
) As Long

Sub Main()
    Dim hWnd As Long
    Dim lngX As Long
    'Get a handle to the active window
    hWnd = GetActiveWindow()
    If (hWnd) Then
        'Call EnumChildWindows API, which calls
        'ChildWindowProc for each child window and then ends
        lngX = EnumChildWindows(hWnd, AddressOf ChildWindowProc, 0)
    End If
End Sub

'Called by EnumChildWindows API function
Function ChildWindowProc( _
    ByVal hWnd As Long,
    ByVal lp As Long
) As Long
    'hWnd and lp parameters are passed in by EnumChildWindows
    Debug.Print "Window: "; hWnd
    'Return success (in C, 1 is True and 0 is False)
    ChildWindowProc = 1
End Function
```

The `EnumChildWindows` function passes the `hWnd` and `lp` parameters to `ChildWindowProc`. The format of the called procedure is described in the Platform SDK online help topic for `EnumChildWindows`. By convention, called procedures end with the suffix *Proc*. This suffix tells you that the procedure is not called directly—instead, it is the target of a callback procedure.

Callbacks can generate some confusing results while you are debugging. For instance, the preceding example enumerates all the window handles in the Visual Basic programming environment while the procedure is being executed in Visual Basic. It enumerates a different set of window handles when it is compiled and executed as a stand-alone.

Keep these key points in mind as you work with the `AddressOf` operator:

- The `AddressOf` operator can be used only on procedures that reside in a code module (BAS file), and then only as part of an argument to a procedure. This prevents you from passing the addresses of procedures that are part of an object or a form.

- The AddressOf operator can reference only procedures in the same project.

Dear John, How Do I... Understand ByVal, ByRef, and As Any in an API Function Declaration?

This question is not so important in 32-bit Visual Basic programming because most of the As Any declarations have been replaced with explicit parameter data types. Nonetheless, I'll go ahead and describe a specific case for which this is a concern—the WinHelp API function.

Many 16-bit API function declarations (and a few 32-bit ones) have one or more parameters declared as As Any instead of as a specific data type such as Integer or String. This enables these parameters to be used to pass a variety of data types, depending on the intended use of the function. For example, in the following code, which uses the WinHelp API function to display a help file, consider the WinHelp function's fourth parameter, which is declared as As Any. (This function will be demonstrated in more detail in Chapter 17, "[User Assistance](#).") Depending on the value of wCommand, the last parameter can be used to pass a long integer or a pointer to a string.

```
Option Explicit
Const vbHelpContents = 3

Private Declare Function WinHelp _
Lib "User32" Alias "WinHelpA" ( _
    ByVal hWnd As Long, _
    ByVal lpHelpFile As String, _
    ByVal wCommand As Long, _
    ByVal dwData As Any _
) As Long

Private Sub Form_Click()
    Dim lngX As Long
    Dim lngY As Long
    lngX = WinHelp(Form1.hWnd, "metric.hlp", vbHelpContents, _
        ByVal lngY)
End Sub
```

By convention, all As Any parameters in API functions are declared by reference. (I've added the ByRef keyword to my declarations to explicitly declare them as such, but ByRef is the default when you don't see either ByVal or ByRef in a parameter declaration.) You'll also usually find the ByVal keyword stuck in front of the variable that is passed for the As Any parameter *at the place where the function is called*. This means that you must pay special attention to how you treat these parameters at the place in your application where the function is actually called. In fact, incorrect use of the ByRef and ByVal keywords can cause your application to crash. Take a close look at the sample call to WinHelp in the code. In this case, the long integer *lngY* is passed as the fourth parameter using the ByVal modifier, which ensures that a long integer value of 0 is passed.

Dear John, How Do I... Easily Add API Declarations?

Visual Basic includes the handy API Viewer Add-In (APILOAD.EXE) in the Program Files\Microsoft Visual Studio\Common\Tools\Winapi directory for browsing and inserting the long list of API constants, associated Type structures, and procedure declarations. The API Viewer Add-In uses public declarations by default. To use API functions in a form, class, user control, or user document module, you will have to add a Private keyword before the Declare statement.

Dear John, How Do I... Use API Calls to Get System Information?

You can use the Windows API functions to readily access a lot of information that Windows normally keeps hidden from Visual Basic applications. In the previous sections, I gave a general introduction to setting up API function calls in your applications. Let's explore some techniques and API functions that can be used to access system information. The following functions and techniques show how to use standard API calls to access just a few of the many types of data available from the system.

Determining the Version of the Operating System Using the SysInfo Control

A lot of system information that was formerly available only through the Windows API can now be accessed with the SysInfo control. This control provides system version, platform, and system event information. SysInfo is useful for writing code to handle differences between Windows 95 and Windows NT. SysInfo is also particularly convenient for the Plug and Play events that occur when a PCMCIA card is inserted in a laptop computer.

To use the SysInfo control, select Components from the Project menu, and then check the Microsoft SysInfo Control 6.0 check box in the Components dialog box. Once a reference to the SysInfo control has been set, the control is displayed in your Toolbox and can be placed on a form. The following code shows how to use the SysInfo control named sysOS to display the operating system and version number:

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim strMsg As String
    'Get platform and version
    Select Case sysOS.OSPlatform
        Case 0
            strMsg = "Unidentified"
        Case 1
            strMsg = "Windows 95, version " & CStr(sysOS.OSVersion)
        Case 2
            strMsg = "Windows NT, version " & CStr(sysOS.OSVersion)
    End Select
    'Display OS information
    Print strMsg
End Sub
```

Figure 10-2 shows an example of the output after the form has been clicked.

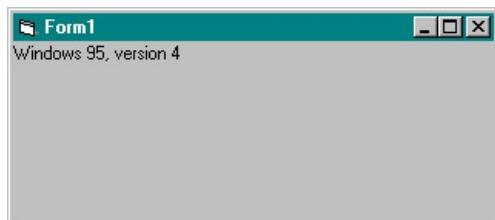


Figure 10-2. Using the SysInfo control to display the operating system and the version number.

Unfortunately, the SysInfo control doesn't include a lot of handy system information, such as system colors, CPU type, and elapsed time. The rest of this section describes how to obtain this information using Windows API functions.

Determining System Colors

The GetSysColor API function returns an RGB color for window items such as captions, menus, and borders. You can get color information about any of 29 window components by passing the appropriate constant to the GetSysColor function. The following code lists these constants and demonstrates the GetSysColor function by setting the main form's background color to match the current desktop color when you click anywhere on the form. Some of these components are available only in specific versions of Windows—for example, the constants from COLOR_3DDKSHADOW to the end of the list are for Windows 95.

```
Option Explicit
```

```
Private Enum SysColor
    COLOR_SCROLLBAR = 0
    COLOR_BACKGROUND = 1
    COLOR_ACTIVECAPTION = 2
    COLOR_INACTIVECAPTION = 3
    COLOR_MENU = 4
    COLOR_WINDOW = 5
    COLOR_WINDOWFRAME = 6
    COLOR_MENUTEXT = 7
    COLOR_WINDOWTEXT = 8
    COLOR_CAPTIONTEXT = 9

    COLOR_ACTIVEBORDER = 10
    COLOR_INACTIVEBORDER = 11
    COLOR_APPWORKSPACE = 12
    COLOR_HIGHLIGHT = 13
    COLOR_HIGHLIGHTTEXT = 14
    COLOR_BTNFACE = 15
    COLOR_BTNSHADOW = 16
    COLOR_GRAYTEXT = 17
    COLOR_BTNTXT = 18
    COLOR_INACTIVECAPTIONTEXT = 19
    COLOR_BTNHIGHLIGHT = 20
    COLOR_3DDKSHADOW = 21
    COLOR_3DLIGHT = 22
    COLOR_INFOTEXT = 23
    COLOR_INFOBK = 24
    COLOR_DESKTOP = COLOR_BACKGROUND
    COLOR_3DFACE = COLOR_BTNFACE
    COLOR_3DSHADOW = COLOR_BTNSHADOW
    COLOR_3DHIGHLIGHT = COLOR_BTNHIGHLIGHT
    COLOR_3DHILIGHT = COLOR_3DHIGHLIGHT
    COLOR_BTNHILIGHT = COLOR_BTNHIGHLIGHT
End Enum
```

```
Private Declare Function GetSysColor _
Lib "user32" (
    ByVal nIndex As Long _
) As Long
```

```
Private Sub Form_Click()
    Dim lngSystemColor As Long
    Dim intRed As Integer
    Dim intGreen As Integer
    Dim intBlue As Integer
    'Get color of desktop
    lngSystemColor = GetSysColor(COLOR_DESKTOP)
    'Set form's background color to same as desktop
    Me.BackColor = lngSystemColor
    'Split this color into its components
    ColorSplit lngSystemColor, intRed, intGreen, intBlue
    Print "R,G,B = "; intRed, intGreen, intBlue
End Sub
```

```
Function ColorSplit(lngRGBMix As Long,
intR As Integer, intG As Integer, intB As Integer)
    'Extract R, G, and B values from an RGB color
    intR = lngRGBMix And &HFF
    intG = (lngRGBMix \ &H100) And &HFF
    intB = (lngRGBMix \ &H10000) And &HFF
End Function
```

Notice that I've also provided a handy function, ColorSplit, to extract the red, green, and blue values from the RGB color value returned by these functions. You might find this function helpful for other graphics calculations—it's the inverse function of Visual Basic's RGB function, which returns a long integer formed by combining red, green, and blue color values, each in the range 0 through 255.

Figure 10-3 shows an example of the output when the form is clicked.

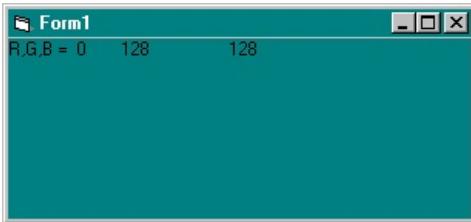


Figure 10-3. Using the *GetSysColor* API function to change the form's background color to the desktop color.

SEE ALSO

- Chapter 14, "[Graphics Techniques](#)," if you're interested in another approach to coordinated color schemes

Determining CPU Type

The following code uses the *GetSystemInfo* API function to determine the type of CPU in the system. By inspecting the *SYSTEM_INFO* Type structure, you'll find that this function returns several other useful bits of information about the system. In anticipation of more advanced systems that will be available in the near future, this data structure even returns the number of processors on the current system.

```
Option Explicit
```

```
Private Type SYSTEM_INFO
    dwOemID As Long
    dwPageSize As Long
    lpMinimumApplicationAddress As Long
    lpMaximumApplicationAddress As Long
    dwActiveProcessorMask As Long
    dwNumberOfProcessors As Long
    dwProcessorType As Long
    dwAllocationGranularity As Long
    dwReserved As Long
End Type

Private Declare Sub GetSystemInfo _
Lib "kernel32" (
    lpSystemInfo As SYSTEM_INFO _
)

Private Sub Form_Click()
    Dim Sys As SYSTEM_INFO
    GetSystemInfo Sys
    Print "Processor type: "; Sys.dwProcessorType
    Print "Number of processors: "; Sys.dwNumberOfProcessors
End Sub
```

Figure 10-4 shows an example of the output when the form is clicked.

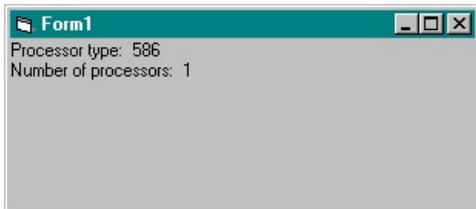


Figure 10-4. Using *GetSystemInfo* to determine the type and number of processors.

Determining Elapsed Time

The *GetTickCount* Windows API function returns the number of milliseconds that have elapsed since Windows was started. Depending on your system, this function returns a value with greater precision than that of Visual Basic's *Timer* function. The *Timer* function returns the number of seconds since midnight and includes fractions of seconds that provide the illusion of millisecond accuracy, but Visual Basic actually updates the value returned by *Timer* only 18.2 times per second. On my Pentium system, the value returned by *GetTickCount* is updated almost 100 times each second.

Another advantage of *GetTickCount* over the *Timer* function is that crossing the midnight boundary causes no problems. The *Timer* value returns to 0 at midnight, whereas the *GetTickCount* value returns to 0 only after about 49.7 days of continuous computer operation. In the following code, when the form is clicked, the *GetTickCount* API function is called and the time elapsed since Windows was started, in milliseconds, is printed on the form. This process is repeated nine more times to show how often the value returned from *GetTickCount* is updated.

```
Option Explicit

Private Declare Function GetTickCount _ 
Lib "kernel32" ( _
) As Long

Private Sub Form_Click()
    Dim i As Integer
    Dim lngMS As Long
    Print "Time elapsed since Windows was started:"
    For i = 1 To 10
        lngMS = GetTickCount
        Do While lngMS = GetTickCount
        Loop
        Print lngMS; " milliseconds"
    Next i
End Sub
```

Figure 10-5 shows an example of the output when the form is clicked.



Figure 10-5. Time elapsed since Windows was started, as reported by *GetTickCount* and the updated values.

Determining Drive Types

It's easy to determine whether the user's computer has one or two floppy drives or one or more hard drives or is connected to any remote drives; this is accomplished by calling the *GetDriveType* Windows

API function. You might want to use this function in a program that searches all available drives for a specific data file, for instance. In the following code, the GetDriveType function is used to detect all drives that are present on the system:

```

Option Explicit

`GetDriveType return values
Const DRIVE_REMOVABLE = 2
Const DRIVE_FIXED = 3
Const DRIVE_REMOTE = 4
Const DRIVE_CDROM = 5
Const DRIVE_RAMDISK = 6

Private Declare Function GetDriveType _ 
Lib "kernel32" Alias "GetDriveTypeA" ( _
    ByVal nDrive As String _ 
) As Long

Private Sub Form_Click()
    Dim i As Integer
    Dim lngDrive As Long
    Dim strD As String
    For i = 0 To 25 `All possible drives A to Z
        strD = Chr(i + 65) & ":\" 
        lngDrive = GetDriveType(strD)
        Select Case lngDrive
            Case DRIVE_REMOVABLE
                Print "Drive " & strD & " is removable."
            Case DRIVE_FIXED
                Print "Drive " & strD & " is fixed."
            Case DRIVE_REMOTE
                Print "Drive " & strD & " is remote."
            Case DRIVE_CDROM
                Print "Drive " & strD & " is CD-ROM."
            Case DRIVE_RAMDISK
                Print "Drive " & strD & " is RAM disk."
            Case Else
                End Select
        Next i
End Sub

```

The list of available drives is displayed when the form is clicked, as shown in Figure 106.

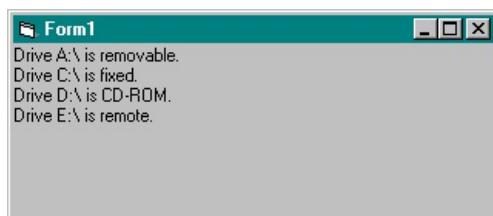


Figure 10-6. Using *GetDriveType* to return the types of drives on a system.

Dear John, How Do I... Add API Calls to an ActiveX Control?

Microsoft left out a lot of useful information when it built the SysInfo control. For example, you can use SysInfo to detect when the user changes the system colors, but you then have to use the Windows API to determine which colors were changed.

Fortunately, you can now create your own controls with Visual Basic—and controls make a great home for common API declarations. You can repackage the APIs you use all the time as methods or properties and then call them to get the information you need, without having long, messy API declarations in your code.

Adding API Declarations to an ActiveX Control

API declarations can be included in a user control module, just as they can be included in any other module. At the simplest level, all you have to do is create a new ActiveX Control project, declare the API function at the control's module level, and then call the API function from a Public procedure. The following code shows a GetTicks method in a control named *TickControl*:

```
Option Explicit

`TickControl control module
`API declaration
Private Declare Function GetTickCount _

Lib "kernel32" (
) As Long

Public Function GetTicks() As Long
    `Return number of milliseconds since Windows started
    GetTicks = GetTickCount
End Function
```

Notice that the API declaration must be Private. The GetTicks function simply repackages the GetTickCount API function so that it can be called from outside the module.

SEE ALSO

- Chapter 6, "[ActiveX Controls](#)," for more information about creating controls

Enhancing an Existing Control

Controls can contain other controls, which is handy when you want to add features to an existing control but don't have its source code. Remember that you'll have to install both controls on your user's system if you do this.

You can add all sorts of new properties and methods to enhance the SysInfo control by superclassing the control—*superclassing* means that an existing object is included in a new object, keeping all the old object's features and adding your own properties and methods.

Figure 10-7 shows the SysInfo control placed in the UserControl window. Code can be added to enhance, or superclass, the control.

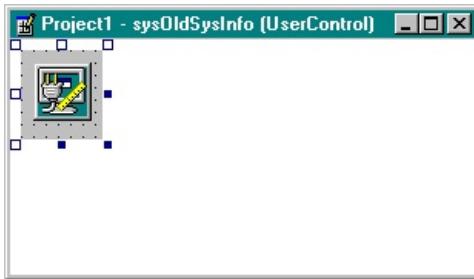


Figure 10-7. The SysInfo control can be enhanced by superclassing.

To superclass a control, you've got to reproduce all the properties, methods, and events provided by the base control. The code to accomplish this is trivial but tiresome to type, as shown in the following example:

```
'Delegate to SysInfo Properties
Public Property Get ACStatus() As Integer
    ACStatus = sysOldSysInfo.ACStatus
End Property
Public Property Get BatteryFullTime() As Long
    BatteryFullTime = sysOldSysInfo.BatteryFullTime
End Property
Public Property Get BatteryLifePercent() As Integer
    BatteryLifePercent = sysOldSysInfo.BatteryLifePercent
End Property
Public Property Get BatteryLifeTime() As Long
    BatteryLifeTime = sysOldSysInfo.BatteryLifeTime
End Property
Public Property Get BatteryStatus() As Integer
    BatteryStatus = sysOldSysInfo.BatteryStatus
End Property
Public Property Get OSBuild() As Integer
    OSBuild = sysOldSysInfo.OSBuild
End Property
Public Property Get OSPlatform() As Integer
    OSPlatform = sysOldSysInfo.OSPlatform
End Property
Public Property Get OSVersion() As Single
    OSVersion = sysOldSysInfo.OSVersion
End Property
Public Property Get ScrollBarSize() As Single
    ScrollBarSize = sysOldSysInfo.ScrollBarSize
End Property
Public Property Get WorkAreaHeight() As Single
    WorkAreaHeight = sysOldSysInfo.WorkAreaHeight
End Property
Public Property Get WorkAreaLeft() As Single
    WorkAreaLeft = sysOldSysInfo.WorkAreaLeft
End Property
Public Property Get WorkAreaTop() As Single
    WorkAreaTop = sysOldSysInfo.WorkAreaTop
End Property
Public Property Get WorkAreaWidth() As Single
    WorkAreaWidth = sysOldSysInfo.WorkAreaWidth
End Property
```

Reproducing properties and methods in this way is called *delegating*. You don't need to delegate common properties such as Name and Parent because these properties are provided by the container.

To reproduce all of the SysInfo events in the Enhanced SysInfo control, declare each event with an Event statement, and then raise the new event using a RaiseEvent statement in each of the SysInfo control's event procedures. Again, the code is trivial but tedious. An abbreviated version is shown here:

```
'Event declarations
```

```

Event ConfigChangeCancelled()
Event ConfigChanged( _
    OldConfigNum As Long, _
    NewConfigNum As Long)
Event DeviceArrival( _
    DeviceType As Long, _
    DeviceID As Long, _
    DeviceName As String, _
    DeviceData As Long)
Event DeviceOtherEvent( _
    DeviceType As Long, _
    EventName As String, _
    DataPointer As Long)
Event DeviceQueryRemove( _
    DeviceType As Long, _
    DeviceID As Long, _
    DeviceName As String, _
    DeviceData As Long, _
    Cancel As Boolean)
`And so onDear John, How Do I...

`Raise all the SysInfo events on the user control
Private Sub OldSysInfo_ConfigChangeCancelled()
    RaiseEvent ConfigChangeCancelled
End Sub
Private Sub OldSysInfo_ConfigChanged( _
    ByVal OldConfigNum As Long, _
    ByVal NewConfigNum As Long)
    RaiseEvent ConfigChanged(OldConfigNum, NewConfigNum)
End Sub
Private Sub OldSysInfo_DeviceArrival( _
    ByVal DeviceType As Long, _
    ByVal DeviceID As Long, _
    ByVal DeviceName As String, _
    ByVal DeviceData As Long)
    RaiseEvent DeviceArrival(DeviceType, DeviceID, DeviceName, _
        DeviceData)
End Sub
Private Sub OldSysInfo_DeviceOtherEvent( _
    ByVal DeviceType As Long, _
    ByVal EventName As String, _
    ByVal DataPointer As Long)
    RaiseEvent DeviceOtherEvent(DeviceType, EventName, DataPointer)
End Sub
Private Sub OldSysInfo_DeviceQueryRemove( _
    ByVal DeviceType As Long, _
    ByVal DeviceID As Long, _
    ByVal DeviceName As String, _
    ByVal DeviceData As Long, _
    Cancel As Boolean)
    RaiseEvent DeviceQueryRemove(DeviceType, DeviceID, DeviceName, _
        DeviceData, Cancel)
End Sub
`And so onDear John, How Do I...

```

Each SysInfo event raises an event of the same name in the enhanced control. After reproducing (or *wrapping*) all the SysInfo events, properties, and methods, you finally get to add some of your own, as shown here:

```

`Declarations for system colors
Public Enum SystemColor
    COLOR_SCROLLBAR = 0
    COLOR_BACKGROUND = 1
    COLOR_ACTIVECAPTION = 2

```

```
COLOR_INACTIVECAPTION = 3
COLOR_MENU = 4
COLOR_WINDOW = 5
COLOR_WINDOWFRAME = 6
COLOR_MENUTEXT = 7
COLOR_WINDOWTEXT = 8
COLOR_CAPTIONTEXT = 9
COLOR_ACTIVEBORDER = 10
`Constants omitted for brevity
End Enum

Private Declare Function GetSysColor _
Lib "user32" (
    ByVal nIndex As Long _
) As Long

Public Function GetSystemColor(nIndex As SystemColor) As Long
    GetSystemColor = GetSysColor(nIndex)
End Function
```

The `GetSystemColor` method returns the color setting for a specific Windows object. The values in the Enum are displayed in the Object Browser when you use the control in a new application.

SEE ALSO

- Visual Basic's ActiveX Control Interface Wizard for an easy way to create interfaces. This wizard can be added to Visual Basic by checking it in the Add-In Manager, which is accessed from the Add-Ins menu

Chapter Eleven

Multimedia

In this chapter, I show you two ways to add sound files and video clips to your applications. To try the code examples, you need to have a sound board and the appropriate multimedia drivers installed. You also need a sample WAV file to play sound and a sample AVI file to play a video clip.

NOTE

You can find many sample sound and video files on your Windows 95 CD-ROM. Right-click the Start button, and choose Find to search for *.WAV and *.AVI files on your CD-ROM drive.

Dear John, How Do I... Play a Sound (WAV) File?

There are several ways to play a sound file. Here we take a look at two very straightforward methods: one using the mciExecute API function supplied by Windows, and the other using the Multimedia control included with Visual Basic.

The mciExecute Function

The following code shows how to declare the mciExecute API function and then use it to play a WAV file. To try this example, add the code to a blank form in a new project, run the application, and click anywhere on the form. Be sure to change the path and file name in the mciExecute command to reference a WAV file existing on your system.

```
Option Explicit

Private Declare Function mciExecute _
Lib "winmm.dll" (
    ByVal lpstrCommand As String _
) As Long

Private Sub Form_Click()
    Dim intX
    intX = mciExecute("Play C:\Windows\Media\Office97\whoosh.wav")
    'Change filename to name of your sample WAV file
End Sub
```

This same function can be used to send other multimedia commands. For example, we'll soon see how to play a video clip with it.

The Multimedia Control

The Multimedia control included with Visual Basic is an excellent tool for playing sound files. With minor modifications to the code, you can also use this control to play video for Windows (AVI) files and multimedia movie (MMF) files and to control multimedia hardware. Here we concentrate on playing a sound file.

For the example that follows, draw a Multimedia control on a blank form in a new project. If the Multimedia control (whose class name, and thus its ToolTip name, is MMControl) is not in your Toolbox, add it by choosing Components from the Project menu and selecting Microsoft Multimedia Control 6.0 from the list on the Controls tab of the Components dialog box. Name this control *mciTest*, and set its Visible property to *False*. This example demonstrates how to use an invisible control at runtime. For a description of all the buttons on the control and an explanation of how the user can interact with them in the visible mode, refer to the Visual Basic online help.

I've set up the following example to play the sample sound file WHOOSH.WAV when you click anywhere on the form. You can add this code to any event-driven procedure you want, such as an error-trapping procedure or any other event for which you want to give audible notification to the user.

```
Option Explicit

Private Sub Form_Click()
    With mciTest
        .FileName = "C:\Windows\Media\Office97\Whoosh.wav"
        'Change filename to name of your sample WAV file
        .Command = "Sound"
    End With
End Sub
```

Dear John, How Do I... Play a Video (AVI) File?

It's surprisingly easy to play an AVI file on a system that is configured to play these files. You can use the `mciExecute` API function supplied by Windows or the Multimedia control.

The `mciExecute` Function

The code below shows how to declare the `mciExecute` API function and then use it to play a sample video file. To try this example, add the code to a blank form in a new project, run the application, and click anywhere on the form. I've assumed that you have a Windows 95 CD-ROM in your D drive and will use one of its AVI files as your sample video file. Notice that this code is almost identical to the code for playing a WAV file.

```
Option Explicit

Private Declare Function mciExecute _
Lib "winmm.dll" (
    ByVal lpstrCommand As String _
) As Long

Private Sub Form_Click()
    Dim x
    x = mciExecute("Play D:\Funstuff\Videos\Welcome1.avi")
    `Change filename to name of your sample AVI file
End Sub
```

Figure 11-1 shows this form and the video window in action.

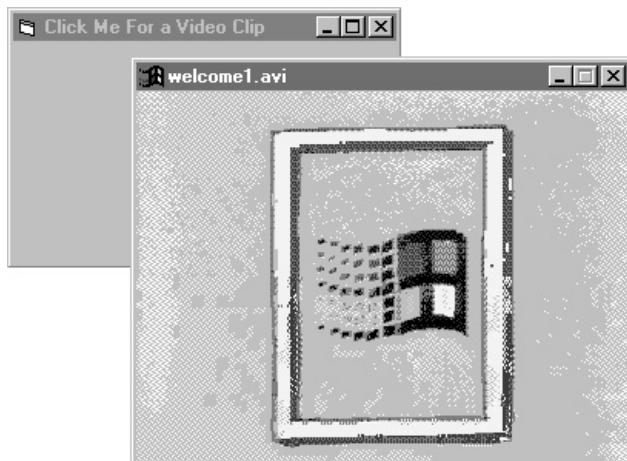


Figure 11-1. A video demonstration initiated by a click on the form.

The Multimedia Control

The Multimedia control included with Visual Basic works well for playing video files. The Visual Basic online help covers the user's interaction with the visible buttons, but I'll show you a simple way to use the control in an "invisible" mode to programmatically play a video.

To try this example, draw a Multimedia control on a blank form in a new project, name it `mciTest`, and set its `Visible` property to `False`. Add the following code, run the application, and click anywhere on the form. You'll need to change the `FileName` property setting to the name and location of your AVI file.

```
Option Explicit

Private Sub Form_Click()
    With mciTest
        .FileName = "D:\Funstuff\Videos\Welcome1.avi"
        .Command = "Open"
        .Command = "Play"
    End With
```

```
End Sub
```

```
Private Sub mciTest_Done(NotifyCode As Integer)
    mciTest.Command = "Close"
End Sub
```

Originally, I tried to put the Close command immediately after the Play command. This didn't work—the video would quit as soon as it started. I solved this problem by putting Close in the Done event code, as shown in the previous code. The system tells us automatically when the video has finished playing, at which time it's safe to perform the Close command.

Dear John, How Do I... Play an Audio CD?

The Multimedia control makes it easy for your Visual Basic applications to play audio (music) compact discs (CDs). You can set the control's Visible property to True and let the user interact with it in much the same way he or she would interact with a real CD player front panel. Or you can make the control invisible and do everything behind the scenes. This lets you design your own music playing application interface, customizing it to your heart's content.

The example I'll provide here uses this second, programmatic technique, although I'll keep things extremely simple. As shown in Figure 11-2, the Music application will display a small form containing Previous and Next buttons. When this program starts up, the first track (song) on a music CD in your CD-ROM drive will automatically start to play. When you click on the Previous or Next button, the application jumps to the start of the previous or next track on the CD. The currently playing track number and the total number of tracks available are displayed at the top of the form. When you close the form, the music stops.

It doesn't take much code to accomplish all the features presented by this small application, and it wouldn't take much to add a lot of other nice features if you're so inclined. The Multimedia control exposes a substantial set of useful properties and methods to handle just about anything you might want to do in an expanded version of the Music application. I've used a core set of these properties and methods — just enough to get the music playing.

Start with a new Standard EXE project. Set the form's Name property to frmMusic, save it as MUSIC.FRM, and save the project as MUSIC.VBP. Set the form's Caption property to "Music Machine" for flash, and then add four controls: a label named lblMusic, a timer named tmrMusic, a Multimedia control named mciMusic, and two command buttons named cmdPrev and cmdNext. Figure 11-2 shows the form at design time, and Figure 11-3 shows it in action when the music is playing. Add the following code to the form to tie it all together.

```

`MUSIC.FRM
Option Explicit

Private Sub cmdNext_Click()
    mciMusic.Command = "Next"
End Sub

Private Sub cmdPrev_Click()
    mciMusic.Command = "Prev"
End Sub

Private Sub Form_Load()
    With mciMusic
        .DeviceType = "CDAudio"
        .Command = "Open"
        .Command = "Play"
    End With
End Sub

Private Sub Form_Unload(Cancel As Integer)
    With mciMusic
        .Command = "Stop"
        .Command = "Close"
    End With
End Sub

Private Sub tmrMusic_Timer()
    With mciMusic
        lblMusic.Caption = "Track No.: " & .Track & _
                           vbCrLf & "No. Tracks: " & .Tracks
    End With
End Sub

```

The Music application sets the Multimedia control's DeviceType property to CDAudio to tell the control to

expect a music CD to be available, and it uses the Tracks and Track properties to display information about the CD's contents. Notice that the Command property is the most frequently used property in the Music application. The Command property provides the means for Visual Basic applications to actually control the playing of multimedia devices. I've used the Open, Play, Next, Prev, Stop, and Close commands to provide the minimum control demonstrated by this application. There are other commands available, just as there are other properties and methods. Check out the online documentation for the Multimedia control to learn more about this component's capabilities and features.

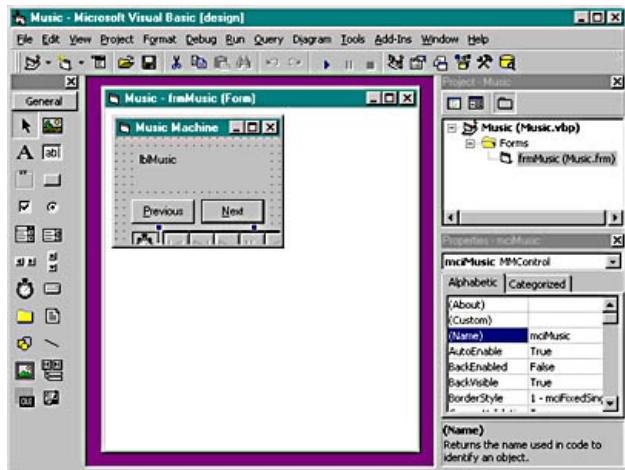


Figure 11-2. The Music form at design time, showing its four controls.



Figure 11-3. The Music form at run time, while the music is playing.

Chapter Twelve

Dialog Boxes, Windows, and Other Forms

In this chapter, I provide several helpful hints and tips for working with forms. Some of these techniques are old standards, such as positioning a form on the screen, and some are newer, such as creating a tabbed form. I've found each of these tips to be a useful addition to my bag of tricks.

Dear John, How Do I... Add a Standard About Dialog Box?

One of the standard elements of many Microsoft Windows applications is the About dialog box. Usually, the user activates an About dialog box by choosing About from the Help menu. Check out the Help menu of almost any Windows application, and you'll find About there.

You can easily add your own About dialog box to your applications. It doesn't have to be very fancy; in fact, you can use the MsgBox function to create a simple About dialog box. For a display with a more professional appearance, you can create an About dialog box using a form.

I've created a form with the Name property *About* and saved it as ABOUT.FRM. The About form can easily be loaded into a project. It has four Label controls, named *lblHeading*, *lblApplication*, *lblVB*, and *lblCopyright*, and one command button, named *cmdOK*. When the form is called, it will look something like Figure 12-1, depending on what strings are passed to it.

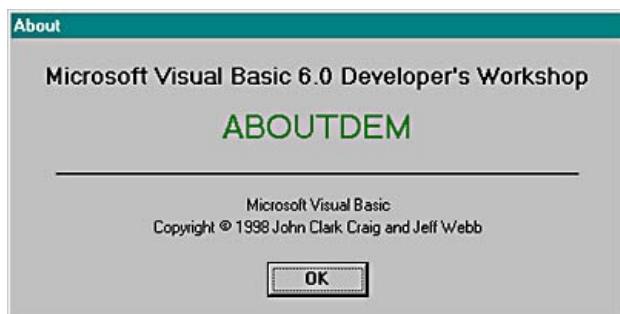


Figure 12-1. A standard About dialog box.

Here is the code for the About form:

```
Option Explicit

Private Sub cmdOK_Click()
    'Cancel About form
    Unload Me
End Sub

Private Sub Form_Load()
    'Center this form
    Left = (Screen.Width - Width) \ 2
    Top = (Screen.Height - Height) \ 2
    'Set defaults
    lblApplication.Caption = "- Application -"
    lblHeading.Caption = "- Heading -"
    lblCopyright.Caption = "- Copyright -"
End Sub

Public Sub Display()
    'Display self as modal
    Show vbModal
End Sub

Property Let Heading(Heading As String)
    'Define string property for Heading
    lblHeading.Caption = Heading
End Property

Property Let Application(Application As String)
    'Define string property for Application
    lblApplication.Caption = Application
End Property

Property Let Copyright(Copyright As String)
    'Build complete Copyright string property
    lblCopyright.Caption = "Copyright © " & Copyright

```

```
End Property
```

Notice that in the code I've used Property Let procedures to assign strings to the Heading, Application, and Copyright labels of the About form. By using these property procedures, your calling application can set these properties to whatever strings you want without worrying about public variables or directly referencing the Caption properties of the Label controls on the About form. For example, to have the calling application assign the string *ABOUTDEM* to the application caption on the About form, the code would be this:

```
About.lblApplication.Caption = "ABOUTDEM"
```

By treating the form as an object and using Property Let procedures, the call would be much easier, as shown here:

```
About.Application = "ABOUTDEM"
```

Another advantage of using property procedures is that other actions can be initiated by the object when a property is assigned. For example, the string assigned to the Copyright property is concatenated to a standard copyright notice before it's displayed.

The About form code also uses one public method, Display, which I've substituted for the standard Show method. I've added the Display method so that you don't have to include the *vbModal* constant as an argument in the Show method. If the *vbModal* constant is included as an argument, it specifies the form as modal and requires the user to respond to the form before interaction with other forms in an application is allowed. Having the form set as modal is the standard state in which to display the About dialog box. If you use the Display method, which is part of the About form module, you don't need to address the issue of modal or nonmodal state when you display the form. You could also add code to the Display method to enhance the display. For example, you could use a Timer control to display the About dialog box for a specific period and then hide it from view. As a general rule, object-oriented programming (OOP) transfers the responsibility for taking actions to the objects themselves, and this Display method follows that rule.

If you want a different style of generic About dialog box, feel free to modify the About form to your heart's content. Be sure to change the default string property settings to something appropriate for your purposes. These default strings are set in the Form_Load event procedure.

I've used this About dialog box in many of the demonstration applications in Part III of this book. It was easy to add this form to each project: I added an About menu item to the standard Help menu and added a few lines of code to activate the About form when this menu item is selected.

To use this code, you need to create a new project and add the ABOUT.FRM file to the project. For a simple example, which creates an About dialog box similar to the one shown in Figure 12-1 earlier in the chapter, create a startup form with a menu to call the About form, and add the following code:

```
Option Explicit
```

```
Private Sub mnuAbout_Click()
    'Set properties
    About.Heading = "Microsoft Visual Basic 6.0 Developer's " & _
        "Workshop"
    About.Application = "ABOUTDEM"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    'Call a method
    About.Display
End Sub
```

The About Dialog Form Template

Visual Basic includes many template forms that you can add to your project and customize in whatever way you want. One of the template forms is an About Dialog form. One helpful feature of the About Dialog form template is an already created System Info button, which is a familiar feature in many About dialog boxes for Microsoft applications. The System Info button already contains all the code necessary for it to work properly—you don't need to supply any additional code. These template forms are located in your Visual Basic directory, in the TEMPLATE directory.

To add the About Dialog form to your project, choose Add Form from the Project menu to display the Add Form dialog box, as shown in Figure 12-2.

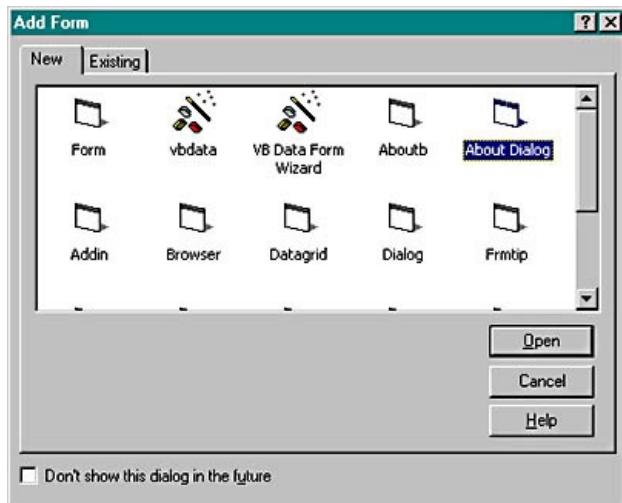


Figure 12-2. The Add Form dialog box, containing form templates.

On the New tab of the Add Form dialog box, select the About Dialog icon and click Open. The About Dialog form is added to your project. The following code shows an example of how you might customize your About Dialog form template:

```
Option Explicit
```

```
Private Sub mnuAbout_Click()
    frmAbout.Caption = App.Title
    frmAbout.lblTitle = "ABOUTDEM"
    frmAbout.lblVersion = "Version " & App.Major & "." & App.Minor _
        & "." & App.Revision
    frmAbout.lblDescription = "Microsoft Visual Basic 6.0 " & _
        "Developer's Workshop"
    frmAbout.lblDisclaimer = "Copyright © 1998 John Clark Craig " & _
        "and Jeff Webb"
    frmAbout.Show
End Sub
```

Figure 12-3 below shows the resulting About Dialog form.



Figure 12-3. An About dialog box created with Visual Basic's About Dialog form template.

SEE ALSO

- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for a demonstration of the use of an About dialog box

Dear John, How Do I... Automatically Position a Form on the Screen?

The best place to put code to position a form is in the form's Load procedure. This positions the form before it actually appears on the screen. To center a form, simply add two lines to the form's Load procedure that calculate and specify the location of the upper-left corner of your form, as shown in the following code:

```
Private Sub Form_Load()
    'Center this form
    Left = (Screen.Width - Width) \ 2
    Top = (Screen.Height - Height) \ 2
End Sub
```

NOTE

Notice that the backslash character (\) is used to execute integer division by 2. Integer division is faster than floating-point division, and in many situations (such as when you're centering a form) the result is to be rounded to the nearest integer value anyway. Use \ instead of / whenever it will work just as well.

Figure 12-4 shows a centered form.

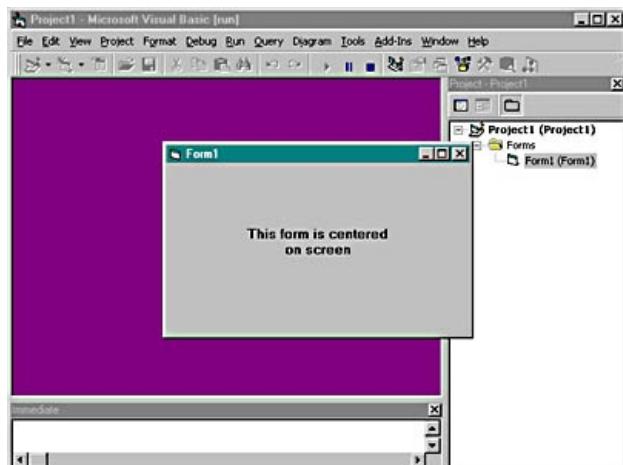


Figure 12-4. Sample form centered on the screen.

If you want to position a form somewhere else on the screen, it's best to create a general procedure to handle the positioning. For example, the Locate procedure shown here positions the center of your form anywhere on the screen, using absolute coordinates:

```
Private Sub Form_Load()
    'Position this form in left quarter, top third of screen
    Locate 1/4, 1/3
End Sub

Private Sub Locate(
    Optional Xadjust As Single = 0.5, _
    Optional Yadjust As Single = 0.5 _
)
    Left = Xadjust * Screen.Width - Width \ 2
    Top = Yadjust * Screen.Height - Height \ 2
End Sub
```

The parameters *Xadjust* and *Yadjust* can be any percentage of the screen's dimensions. If you call Locate without any arguments, the default settings will be used. In this case, the default settings will center the form on the screen.

Still another technique for positioning a form on the screen is to use a feature available in Visual Basic, the Form Layout window, shown in Figure 12-5. If the Form Layout window is not currently displayed, you can open it by choosing Form Layout Window from the View menu.

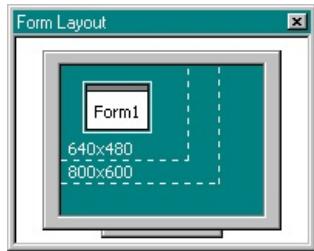


Figure 12-5. The Form Layout window.

In the Form Layout window, you can visually position the form anywhere on the screen. If you right-click on the screen and choose Resolution Guides from the pop-up menu, dotted lines are drawn indicating the different screen resolutions (640x480, 800x600, and 1024x768). Although this is a very easy way to position a form on the screen, you can't position the form independent of the resolution. If you want a form to always be centered no matter what resolution the user specifies, you should position the form using the preceding code.

SEE ALSO

- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for a demonstration of a form-positioning technique

Dear John, How Do I... Create a Floating Window?

This is a rather loaded question because you can create a form that is modal, floating, or topmost, and the form will stay visible in front of other forms and windows. Let's look at techniques for controlling a form in each of these ways.

Modal Mode

Forms are displayed programmatically using the Show method. If you include the constant *vbModal* as the value for the optional argument in the Show method of a form, the user must attend to the form before any other parts of the application will recognize keypresses or mouse activity. In a sense, you might call this behavior *application modal*. Here's a code line that demonstrates this use of the Show method:

```
frmTest.Show vbModal
```

Note that your modal form must have some way of hiding or unloading itself, directly or indirectly, so that the application can go on from there.

Floating Mode

I consider the floating mode a little different from the topmost mode, which we'll get to next, although the two terms are often used interchangeably. You might think of a floating window as continually bobbing back to the surface.

To create this type of form, add a Timer control to your form and set its Interval property to the speed at which you want the form to come floating back up. An interval of 500 milliseconds (0.5 second), for instance, is a reasonable value to try. Add the following lines to your Timer1_Timer event procedure to force the form to the top using the ZOrder method:

```
Private Sub Timer1_Timer()
    ZOrder
End Sub
```

Topmost Mode

You can use the Windows API function SetWindowPos to make a form stay on top even while you switch between applications. This creates a better effect than the bobbing motion just described and lets Windows do all the dirty work. Your form will stay on top of all other forms and windows until you close it. I've packaged calls to the SetWindowPos function in an OnTop property for the form. The code, shown below, then calls OnTop from the Form_Load and Form_Unload event procedures.

```
Option Explicit
```

```
`SetWindowPos flags
Private Const SWP_NOSIZE = &H1
Private Const SWP_NOMOVE = &H2
Private Const SWP_NOZORDER = &H4
Private Const SWP_NOREDRAW = &H8
Private Const SWP_NOACTIVATE = &H10
Private Const SWP_FRAMECHANGED = &H20
Private Const SWP_SHOWWINDOW = &H40
Private Const SWP_HIDEWINDOW = &H80
Private Const SWP_NOCOPYBITS = &H100
Private Const SWP_NOOWNERZORDER = &H200
Private Const SWP_DRAWFRAME = SWP_FRAMECHANGED
Private Const SWP_NOREPOSITION = SWP_NOOWNERZORDER

`SetWindowPos hwndInsertAfter values
Private Const HWND_TOP = 0
Private Const HWND_BOTTOM = 1
Private Const HWND_TOPMOST = -1
Private Const HWND_NOTOPMOST = -2
```

```
Private Declare Function SetWindowPos _  
Lib "user32" ( _  
    ByVal hwnd As Long, _  
    ByVal hWndInsertAfter As Long, _  
    ByVal x As Long, _  
    ByVal y As Long, _  
    ByVal cx As Long, _  
    ByVal cy As Long, _  
    ByVal wFlags As Long _  
) As Long  
  
Private mbOnTop As Boolean  
  
`~~~.OnTop  
Private Property Let OnTop(Setting As Boolean)  
`Set form's OnTop property  
    If Setting Then  
        `Make this form topmost  
        SetWindowPos hwnd, HWND_TOPMOST,  
            0, 0, 0, 0, SWP_NOMOVE Or SWP_NOSIZE  
    Else  
        `Make this form non-topmost  
        SetWindowPos hwnd, HWND_NOTOPMOST,  
            0, 0, 0, 0, SWP_NOMOVE Or SWP_NOSIZE  
    End If  
    mbOnTop = Setting  
End Property  
  
Private Property Get OnTop() As Boolean  
    `Return the private variable set in Property Let  
    OnTop = mbOnTop  
End Property  
  
Private Sub Form_Load()  
    `Place form on top of all others  
    OnTop = True  
End Sub  
  
Private Sub Form_Unload(Cancel As Integer)  
    `Put form back in normal ZOrder  
    OnTop = False  
End Sub
```

I have placed the code to return the form to normal ZOrder in the Form_Unload event procedure, but you can put this code anywhere in your application. For instance, you could set up the code so that you can toggle the form into and out of the topmost state by clicking a button. Using property procedures to contain the SetWindowPos calls makes it easy to toggle the setting. For instance, the following line of code switches the OnTop property on or off, depending on its initial setting:

```
OnTop = Not OnTop
```

You don't actually need to add all the window position constants to your code. I included all relevant constants for handy reference. You might want to create other form properties that use these constants by combining them with the *wFlags* parameter of SetWindowPos using the Or operator.

Dear John, How Do I... Create a Splash (Logo) Screen?

Often a large application will take several seconds to get up and running, its loading time varying according to both the amount of initialization needed and the speed of the user's system. One of the best ways to use the screen during this delay is to display a logo, a trademark, or what some have come to call a *splash screen*. The code below shows a straightforward way to display a splash screen named *frmSplash*.

```
Option Explicit
Private Sub Form_Load()
    ' Show this form
    Show
    ' Show splash screen
    frmSplash.Show
    DoEvents
    ' Perform time-consuming initializationsDear John, How Do I...
    Initialize
    ' Erase splash screen
    Unload frmSplash
End Sub
```

The code in this procedure should be inserted into the application's startup form. Usually, the startup form is the main form that will remain active as long as the application is running. Everything to control the splash screen can be done right here in the Form_Load event procedure because Visual Basic gives us some control over the order of events. The first Show method forces Windows to draw the main form on the screen. (Normally, this doesn't take place until after the Form_Load event procedure has finished.) The next Show method displays the splash screen, which is a form of your own design named *frmSplash*. I've followed this Show method with a DoEvents function to ensure that all elements of the splash screen form are completely drawn right away. The DoEvents function forces Visual Basic to yield control to the operating system until all pending operations are completed. The Initialize function represents the time-consuming tasks that your application performs at startup, such as loading data from files, loading resources from a resource file, loading forms into memory, and so on. After the initialization is complete, the splash screen form is unloaded, and everything's ready to go.

Figure 12-6 shows an example of a splash screen display centered on the application's main form.

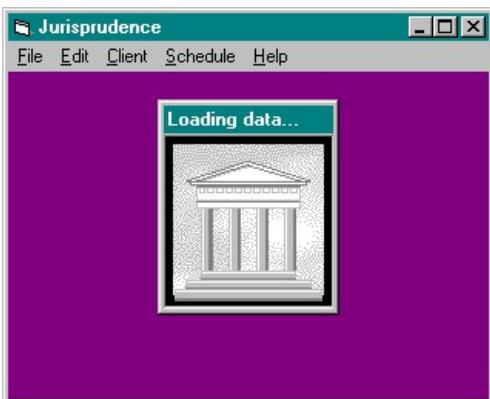


Figure 12-6. Imaginary application's splash screen in action.

The Splash Screen Form Template

As mentioned, Visual Basic includes many template forms. One of the template forms is a splash screen. To add the Splash Screen form to your project, choose Add Form from the Project menu. On the New tab of the Add Form dialog box, select the Splash Screen icon and click Open. The Splash Screen form is added to your project.

The following code shows an example of how you might customize the Splash Screen form template:

```
Option Explicit
```

```
Private Sub Form_Load()
```

```
'Specify splash screen labels
frmSplash.lblLicenseTo = App.LegalTrademarks
frmSplash.lblCompanyProduct = App.ProductName
frmSplash.lblPlatform = "Windows 95"
frmSplash.lblCopyright = App.LegalCopyright
frmSplash.lblCompany = App.CompanyName
frmSplash.lblWarning = " Warning: This program is protected " & _
    "by copyright law, so don't copy"
`Show splash screen
frmSplash.Show
DoEvents
`Perform time-consuming initializationsDear John, How Do I...
Initialize
`Erase splash screen
Unload frmSplash
End Sub
```

Notice that the App object is used here. The App object can access information about your application. Much of this information is set at design time on the Make tab of the Project Properties dialog box, which can be accessed by choosing Project Properties from the Project menu.

The code in the Splash Screen form template code module is shown here:

```
Private Sub Form_KeyPress(KeyAscii As Integer)
    Unload Me
End Sub

Private Sub Form_Load()
    lblVersion.Caption = "Version " & App.Major & "." & _
        App.Minor & "." & App.Revision
    lblProductName.Caption = App.Title
End Sub

Private Sub Frame1_Click()
    Unload Me
End Sub
```

Notice that the Splash Screen form template code module already has code that sets the version and product name for you, based on the settings on the Make tab of the Project Properties dialog box.

Figure 12-7 shows an example splash screen.

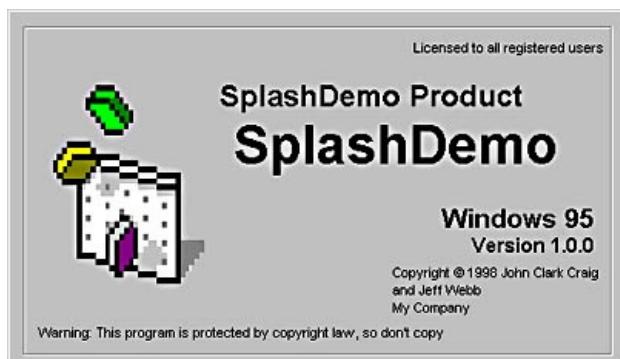


Figure 12-7. A splash screen created using the Splash Screen form template.

SEE ALSO

- The Jot application in Chapter 32, "[Databases](#)," for a demonstration of the incorporation of a splash screen

Dear John, How Do I... Use a Tabbed Control?

Visual Basic provides two tabbed controls: the TabStrip control, provided in the file COMCTL32.OCX, and the SSTab control, provided in the file TABCTL32.OCX. The SSTab control is an improved version of the Sheridan SSTab control included in Visual Basic 4.

You can add these tabbed controls to your Toolbox from the Components dialog box. The control description for the TabStrip control is Microsoft Windows Common Controls and for the SSTab control is Microsoft Tabbed Dialog Control.

The SSTab Control

The SSTab control is easier to use than the TabStrip control. The main difference between the TabStrip control and the SSTab control is that the TabStrip control does not provide a container for controls for each tab. For example, when you use the TabStrip control, you typically add a container control, such as a Frame or PictureBox control, to each tab as a control array. You then draw desired controls in each container and add code to display the appropriate container when the user clicks a tab.

With the SSTab control, shown in Figure 12-8, you can simply draw your controls on each tab at design time; the contents of each tab are then displayed automatically when the user selects a tab. You must draw your controls within the area for each tab instead of drawing the control elsewhere on the form and moving it to the tab area. If you simply move a control to the tab area, that control appears on top of the SSTab control but is not associated with a particular tab.

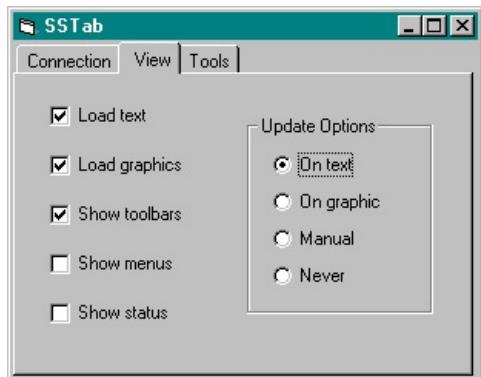


Figure 12-8. The SSTab control in action.

The SSTab control even includes a property for ToolTips. This is an improvement over the SSTab control included with Visual Basic 4, which did not support ToolTips. The SSTab control has two Style property settings that determine appearance: `ssStyleTabbedDialog` and `ssStylePropertyPage`. If the style is set to `ssStyleTabbedDialog`, the tabs look like those used in some Windows version 3.1 applications, and if it is set to `ssStylePropertyPage`, the tabs look like those used in Windows 95.

The only time you can't use the SSTab control is when you're programming for 16-bit Windows. In this case, you should use the 16-bit version of the SSTab control provided with Visual Basic 4.

Dear John, How Do I... Flash a Form to Get the User's Attention?

You can use the FlashWindow API function to toggle, or flash, the title bar of a window. This might be useful for critical conditions that demand user attention. To try this technique, create two forms, *frmControl* and *frmFlash*. Add a Timer control to the *frmFlash* form, and name it *tmrFlash*. Add the following code to the *frmFlash* form:

```
Option Explicit

Private Declare Function FlashWindow _  
Lib "user32" ( _  
    ByVal hwnd As Long, _  
    ByVal bInvert As Long _  
) As Long

Private Sub Form_Load()  
    tmrFlash.Enabled = False  
End Sub

Private Sub tmrFlash_Timer()  
    Dim lngRtn As Long  
    lngRtn = FlashWindow(hwnd, CLng(True))  
End Sub

Property Let Rate(intPerSecond As Integer)  
    tmrFlash.Interval = 1000 / intPerSecond  
End Property

Property Let Flash(blnState As Boolean)  
    tmrFlash.Enabled = blnState  
End Property
```

Add three command buttons (*cmdFast*, *cmdSlow*, and *cmdStop*) to the *frmControl* form to control the flashing, and change the Caption properties of these command buttons appropriately. Add the following code to the *frmControl* form:

```
Option Explicit

Private Sub cmdFast_Click()  
    frmFlash.Rate = 5  
    frmFlash.Flash = True  
End Sub

Private Sub cmdStop_Click()  
    frmFlash.Flash = False  
End Sub

Private Sub cmdSlow_Click()  
    frmFlash.Rate = 1  
    frmFlash.Flash = True  
End Sub

Private Sub Form_Load()  
    frmFlash.Show  
End Sub
```

Figure 12-9 shows this pair of forms in action.



Figure 12-9. The *frmControl* form, which controls the flashing of the *frmFlash* title bar.

In Figure 12-9, *frmControl* has the focus, but the title bar of *frmFlash* is flashing. Note that if the Windows task bar is visible, the task bar icon for *frmFlash* flashes right along with the title bar of the form itself. This task bar icon will flash even if the flashing form is covered by other windows.

I've set up Rate and Flash as properties of the flashing form, in keeping with the spirit of standard object-oriented programming (OOP) techniques. From anywhere in your application, you set the flash rate (in flashes per second) simply by assigning an integer to the form's Rate property and you set the Flash property to *True* or *False* to activate or deactivate the flashing effect. Take a look at the code for the *frmControl* command buttons to see how these properties are set.

SEE ALSO

- The Blinker ActiveX control sample in Chapter 6, "[ActiveX Controls](#)," to see how this functionality can be incorporated into an ActiveX control
- The Messages application in Chapter 34, "[Advanced Applications](#)," to see how a flashing form can be used in an application to get the user's attention

Dear John, How Do I... Move a Control to a New Container?

This is a slick capability of Visual Basic that could open the door to some creative programming techniques. Controls can be drawn within container controls, which currently include picture boxes and frames. The Container property of most controls is readable and writable, which means that you can make a control jump to a different container!

To see how this works, draw two Frame controls, named *fraLeft* and *fraRight*, on a blank form. Draw a command button named *cmdJump* in the middle of *fraLeft*. Figure 12-10 shows the general layout of this form.

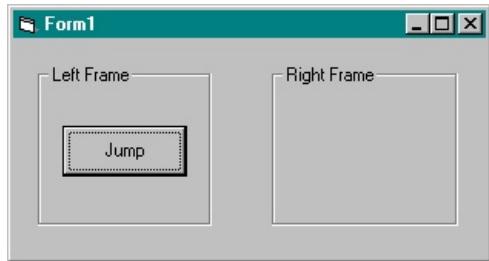


Figure 12-10. A demonstration of the *Container* property, which lets controls move into new control containers at runtime.

Add the following lines of code to the form:

```
Option Explicit
```

```
Private Sub cmdJump_Click()
    Set cmdJump.Container = fraRight
End Sub
```

When you run this program and click the command button, the button will jump to the center of the right frame. This simple demonstration hints at the flexibility of Visual Basic. Objects are much more dynamic and under the control of your program than they used to be.

Chapter Thirteen

The Visual Interface

A fundamental change in the Basic programming language was denoted when Microsoft added the word *Visual* to the name. At the very heart of Visual Basic's success has been the enhanced and easy-to-use visual interface that your programs present to the user. With recent enhancements, Visual Basic adds two significant improvements to the visual interface. First, it makes ToolTips a lot easier to create by adding the *ToolTipText* property to many controls in the Toolbox. Second, it provides many Windows 95 controls, including the UpDown control.

This chapter explains and demonstrates techniques for enhancing your program's interface by using features of Visual Basic and Windows and illustrates some creative programming techniques as well.

Dear John, How Do I... Use the Lightweight Controls?

Visual Basic's Shape, Line, Label, and Image controls have always been windowless, or lightweight. Now you can use windowless versions of the following controls too:

- Check box
- Combo box
- Command button
- Frame
- HScrollBar
- List box
- Option button
- Text box
- VScrollBar

A lightweight control lacks an hWnd property. If your application does not use API functions that require the control's hWnd, the lightweight version of the control will conserve computer resources and allow you to create applications that load faster. Lightweight controls do not support Dynamic Data Exchange (DDE). In all other respects, the lightweight control retains the same behavior as the standard version.

Each of the controls listed above is available in both flavors: the normal windowed version and the new lightweight windowless one. In general, I suggest using the lightweight version of each control unless you need the features and capabilities provided by the windowed version.

Dear John, How Do I... Add a Horizontal Scrollbar to a List Box?

Sometimes it's hard to predict the width of text that will appear in a list box. The `SendMessage` Windows API function provides an easy way to tell a list box to add a horizontal scrollbar to itself if the text lines are too long. The following code shows a working example of this technique.

Start a new project, and add a list box and a command button to a form. I've named these controls `lstTest` and `cmdShrinkList`, respectively. Draw the list box so that it is fairly wide on the form—the command button will shrink it during the demonstration.

Add the declarations to your module for the `SendMessage` API function, as follows:

```
Private Const LB_SETHORIZONTALEXTENT = &H194

Private Declare Function SendMessage _
Lib "user32" Alias "SendMessageA" ( _
    ByVal hwnd As Long, _
    ByVal wMsg As Long, _
    ByVal wParam As Integer, _
    ByVal lParam As Long _
) As Long
```

The `SendMessage` function takes four arguments. The `hwnd` parameter specifies a window handle. The `wMsg` parameter specifies the message to be sent to the window. In this example, the message will be a constant that indicates that a horizontal scrollbar capability should be added to the list box. The `wParam` and `lParam` parameters specify information related to the message. In this example, `wParam` specifies the width in pixels of the list box at which the horizontal scrollbar should be added and `lParam` is not used.

The code for the command button is shown below. This button's purpose is to shrink the list box width by 10 percent each time the button is clicked. When the list box is so narrow that the text doesn't fit, a horizontal scrollbar will automatically appear at the bottom edge of the list box.

```
Private Sub cmdShrinkList_Click()
    lstTest.Width = lstTest.Width * 9 \ 10
End Sub
```

The horizontal scrollbar capability is added to the list box when the form loads. The following code loads a fairly long string into the list box and then adds the scrollbar by calling the `SendMessage` function. Because the `SendMessage` function is expecting a `wParam` value in pixels, you should work in pixel units. The unit of measurement can easily be set to pixels by setting the form's `ScaleMode` property to `vbPixels`. The threshold width of the list box at which the scrollbar appears can be computed by using the `TextWidth` function with the longest string. The `TextWidth` function returns a value in units based on the `ScaleMode` property—in this case, pixels.

```
Private Sub Form_Load()
    Dim strLongest As String, lngRtn As Long
    'Set ScaleMode to Pixels
    ScaleMode = vbPixels
    'Place text in list
    strLongest = "This is a list of months of the year"
    lstTest.AddItem strLongest
    lstTest.AddItem "January"
    lstTest.AddItem "February"
    lstTest.AddItem "March"
    lstTest.AddItem "April"
    lstTest.AddItem "May"
    lstTest.AddItem "June"
    lstTest.AddItem "July"
    lstTest.AddItem "August"
    lstTest.AddItem "September"
    lstTest.AddItem "October"
    lstTest.AddItem "November"
    lstTest.AddItem "December"
```

```
' Set form's font properties to match list box
Form1.Font.Size = lstTest.Font.Size
`Set list box scrollbar threshold width
lngRtn = SendMessage(lstTest.hwnd, LB_SETHORIZONTALEXTENT, _
    Form1.TextWidth(strLongest), ByVal 0&)
End Sub
```

When you run the program, if the list box in its first appearance is wide enough to display the entire string, you'll need to click the command button one or more times before the horizontal scrollbar will appear. With each click, the list box width is reduced to 90 percent of its previous width. Figure 13-1 and Figure 13-2 show the list box with and without the horizontal scrollbar.

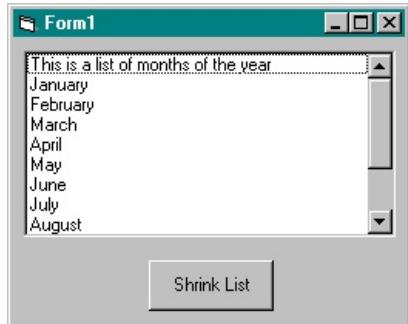


Figure 13-1. A list box that is wide enough to display the longest string; horizontal scrollbar doesn't appear.

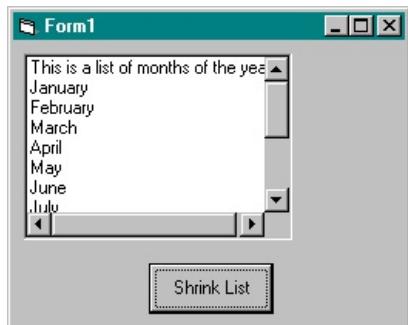


Figure 13-2. A list box that is too narrow to display the longest string; horizontal scrollbar appears.

The TextWidth method returns the width of a string in the current font of a form, a PictureBox control, or a Printer object, but this method doesn't apply to list boxes. To work around this limitation, and to automatically allow for variations in the list box's font size, I copied the value of the Size property of the list box Font object to the Size property of the form's Font object just before using the form's TextWidth method to determine and set the threshold width at which the list box horizontal scrollbar will appear. If other properties of the list box's Font object are altered, you should also copy these properties to the form's Font object properties.

Dear John, How Do I... Create a Toolbar?

To create a toolbar for 32-bit programming, I suggest using the Toolbar control included with Visual Basic as part of the Microsoft Windows Common Controls 6.0 (MSCOMCTL.OCX). To place a series of picture buttons on one of these toolbars, you'll also need to add an ImageList control to your form. The ImageList control lets you store images that can be referenced by their index or their key. You can set properties, such as adding images to the ImageList control, in the Property Pages dialog box for each control. To open the Property Pages dialog box, select the Toolbar or ImageList control on the form and double-click the Custom property in the Properties window. The Property Pages dialog box for the Toolbar control lets you insert buttons, add images from the ImageList control, specify key names, add ToolTips, and set other properties. The Property Pages dialog box for the ImageList control lets you add images, specify the size of the images, and specify color properties.

The Visual Basic online help is a good source for information about using the Toolbar and ImageList controls. The following code shows an example of how to respond to toolbar button clicks:

```
Option Explicit
Private Sub tlbTest_ButtonClick(ByVal Button As ComctlLib.Button)
    Select Case Button.Key
        Case Is = "anchor"
            MsgBox "You clicked the Anchor button."
        Case Is = "umbrella"
            MsgBox "You clicked the Umbrella button."
        Case Is = "phone"
            MsgBox "You clicked the Phone button."
        Case Is = "cherry"
            MsgBox "You clicked the Cherry button."
        Case Is = "atom"
            MsgBox "You clicked the Atom button."
        Case Is = "airplane"
            MsgBox "You clicked the Airplane button."
    End Select
End Sub
```

Figure 13-3 shows a Toolbar control with images from an associated ImageList control after the Airplane button has been clicked.

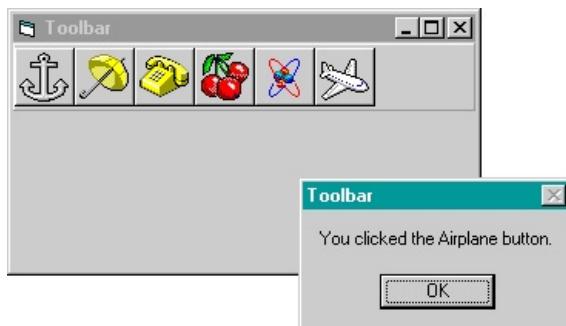


Figure 13-3. The Toolbar control after the Airplane button has been clicked.

For maximum flexibility, or to create a toolbar for the earlier, 16-bit version of Visual Basic, you can simulate a toolbar by using a combination of PictureBox, Image, and perhaps SSSPanel controls. Add a PictureBox control to a form, set its Align property to *1 - Align Top*, and add command buttons inside this picture box. The *1 - Align Top* setting causes the picture box to stretch itself across the top edge of the containing form, even if the form is resized by the user.

You'll probably want to set the BackColor property of the picture box to light gray if it is not already that color. You can also try changing the BorderStyle property to enhance the control's appearance. To improve the effect, consider adding ToolTip capability to these buttons, as described in Chapter 17, "[User Assistance](#)."

SEE ALSO

- "[Dear John, How Do I... Add a Status Display to My Application?](#)" in Chapter 17, "User Assistance"
- "[Dear John, How Do I... Add ToolTips?](#)" in Chapter 17
- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for an example of the use of a toolbar

Dear John, How Do I... Dynamically Change the Appearance of a Form?

Here is one useful technique for dynamically changing a form's appearance: at runtime, you can set a control's Top or Left property to move the control off the visible surface of the containing form or control. A safe way to do this is to move the control to a position at twice the width or the height of the containing control or form. If you save the original position in the control's Tag property, you have an easy way to later move the control back to its starting position. For example, the following code hides the command button *cmdButton* when you click the button and redisplays the button when you click on the form:

```
Option Explicit
Private Sub cmdButton_Click()
    cmdButton.Tag = cmdButton.Left
    cmdButton.Left = Screen.Width * 2
End Sub

Private Sub Form_Click()
    cmdButton.Left = cmdButton.Tag
End Sub
```

You can position and size controls using the Move method. You typically use the ScaleWidth and ScaleHeight properties of the containing form or control with the Move method. This technique can accommodate a sizeable form and keep controls proportionally spaced as the user changes the form's size or shape. Add code in the form's Resize event to accomplish this.

NOTE

- "[Dear John, How Do I... Automatically Position a Form on the Screen?](#)" in Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)," for information about positioning forms
- "[Dear John, How Do I... Use a Tabbed Control?](#)" in Chapter 12 for information about using tabbed controls to make your forms more dynamic

Dear John, How Do I... Dynamically Customize the Menus?

With Visual Basic, you can edit menu objects in the Properties window just as you can edit the properties of the other controls in your application. This feature is easily overlooked, yet it makes the editing of menus a much more manageable task. (Note that you still need to create menu objects by using the Menu Editor.) To get to the menu properties, select the menu from the drop-down list at the top of the Properties window.

NOTE

Another useful method for working with menus is to use menu control arrays, which offer a powerful technique for expanding and shrinking your menus. This subject is covered in the Visual Basic documentation.

Here is a simple trick that isn't too well known, one that provides an easy-to-understand and easy-to-use technique for creating multiple sets of menus. If you set a menu's Visible property to *False*, the menu and all of its submenus will be hidden, both at runtime and in the development environment. So to create two unique File menus, for instance, name the topmost menu item *mnuFile1* for the first File menu and *mnuFile2* for the second File menu. The menus can both have the same Caption property (*File*, in this case), but they must have unique Name properties. The menu items for each File menu can also be entirely unique, or they can share some of the same captions. Just remember to keep each menu item's Name property unique.

Either in the development environment or at runtime, toggle the Visible properties of these two top-level menu items to make one visible and the other hidden. You can set up multiple File menus this way, again by toggling the Visible properties so that only one menu is visible at a time. Note that you don't need to toggle the Visible property for the items on each File menu because they all effectively become invisible when the top-level menu item is made invisible.

You can extend this concept to all the menus to make full use of this technique. You can set up multiple Edit menus or have menu items that come and go depending on the state of the application. You can easily swap out entire sets of menus with replacement sets by using this technique. Figure 13-4 and Figure 13-5 show two unique sets of menus on the same form, one displayed before and one after the Toggle Menus command button has been clicked.

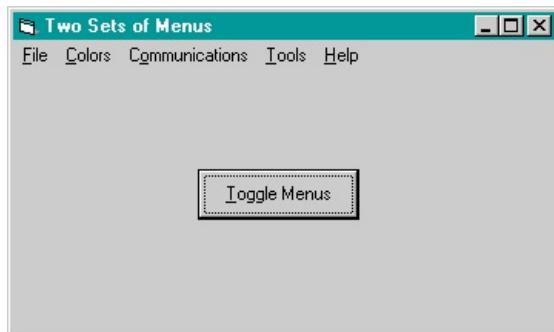


Figure 13-4. A form showing the first of two unique sets of menus.

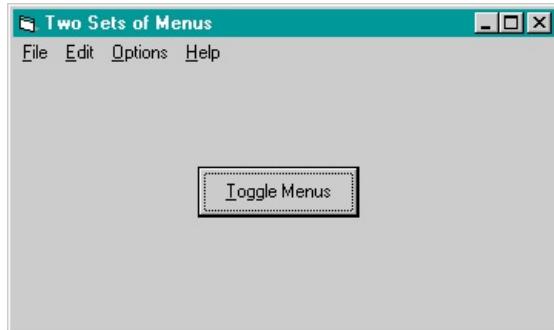


Figure 13-5. The same form showing the second set of menus.

Dear John, How Do I... Remove the Title Bar from a Form?

Before you can remove a title bar from a form at design time, you need to eliminate any menu items. Then you can use one of the following two techniques: Using the first technique, if you set the `ControlBox` property to `False`, delete any text from the `Caption` property, and set the `BorderStyle` property to `1 - Fixed Single`, the form will have a single border and no title bar. For the second technique, set the `BorderStyle` property to `0 - None`. The form will have no border and no title bar, even if any of the `ControlBox`, `MaxButton`, and `MinButton` properties are set to `True` or you have text in the `Caption` property. If you want a border around the form, you can draw it yourself.

SEE ALSO

- "[Dear John, How Do I... Add ToolTips?](#)" in Chapter 17, "User Assistance," for an example of the second technique

Dear John, How Do I... Create a Progress Indicator?

The easiest way to learn how to use the ProgressBar control included with the Windows Common Controls 6.0 (MSCOMMCTL.OCX) is to create a simple application and put it through its paces. The following short application creates a 3-minute timer, suitable for timing the boiling of eggs as well as demonstrating the ProgressBar control.

On a new form, add a Label control named *lblState*, a command button named *cmdStart*, a Timer control named *tmrTest*, and a ProgressBar control named *prgEgg*. Set the label's caption to *Ready to start*, and set its font size as desired. Set the Timer's Enabled property to *False* and its Interval property to 1000 milliseconds. Leave the ProgressBar control's properties set to their defaults.

Add the following code to the form to control the action:

```
Option Explicit
```

```
Private msngStartTime As Single
Private Const intSeconds As Integer = 180

Private Sub cmdStart_Click()
    prgEgg.Value = 0
    msngStartTime = 0
    tmrTest.Enabled = True
End Sub

Private Sub tmrTest_Timer()
    Dim sngPercent As Single
    If msngStartTime = 0! Then
        msngStartTime = Timer
    End If
    sngPercent = 100 * (Timer - msngStartTime) / intSeconds
    If sngPercent < 100 Then
        prgEgg.Value = sngPercent
        lblState.Caption = "CookingDear John, How Do I... "
    Else
        prgEgg.Value = 100
        lblState.Caption = "Done!"
        Beep
        tmrTest.Enabled = False
        prgEgg.Value = 0!
    End If
End Sub
```

Figure 13-6 shows the form at runtime, with my egg about two-thirds done.



Figure 13-6. Using a progress bar as a 3-minute egg timer.

You might want to temporarily change the value of the divisor from 180 (the number of seconds in 3 minutes) to something much smaller while you test this program. I've isolated this value in the constant *intSeconds* to make this easier to do. A value of 10, for example, will cause the ProgressBar control to fill in 10 seconds instead of 3 minutes. (I wouldn't advise you to eat an egg cooked for this length of time, however.)

You can learn a lot about the ProgressBar control by experimenting with this program. For instance, the

number of "chunks" inside the bar is adjusted by changing the height of the bar, an effect that's easier to understand if you try it for yourself instead of just reading about it.

Rolling Your Own Progress Indicator

The ProgressBar control is yet another of those great 32-bit controls that can be used in programs for Windows 95 and Windows NT. But don't despair if you're still stuck in the 16-bit programming world. It's easy to create your own progress bar by using a pair of nested PictureBox controls.

To see how similar this is to the previous example using the ProgressBar control, make these changes: Replace the ProgressBar control with a PictureBox control named *picProgress*. Draw a second PictureBox control inside the first, and name it *picFill*. Set the BackColor properties of these two controls as desired. I used dark blue for *picFill* and white for *picProgress*, but you can use any color. If you change the *picProgress* control's ScaleWidth property to 100 and its ScaleHeight property to 1, you can reuse the previous example's code with only a few modifications:

```
Option Explicit

Private msngStartTime As Single
Private Const intSeconds As Integer = 180

Private Sub cmdStart_Click()
    msngStartTime = 0
    tmrTest.Enabled = True
End Sub

Private Sub Form_Load()
    picFill.Move 0, 0, 0, 0
End Sub

Private Sub tmrTest_Timer()
    Dim sngPercent As Single
    If msngStartTime = 0! Then
        msngStartTime = Timer
    End If
    sngPercent = 100 * (Timer - msngStartTime) / intSeconds
    If sngPercent < 100 Then
        picFill.Move 0, 0, sngPercent, 1
        lblState.Caption = "CookingDear John, How Do I... "
    Else
        lblState.Caption = "Done!"
        Beep
        tmrTest.Enabled = False
        picFill.Move 0, 0, 0, 0
    End If
End Sub
```

This type of progress indicator fills with a solid color instead of using the "chunky" fill style, but the action and appearance are otherwise very similar.

Figure 13-7 shows the egg-timer program in action, again at about the soft-boiled stage.



Figure 13-7. A homegrown progress indicator in action.

Dear John, How Do I... Use the Slider Control?

The Slider control included in MSCOMCTL.OCX is similar to a Scrollbar control except that it has some enhancements that make it a better choice for allowing the user to input numeric values selected from a range. You might think of the scrollbar as a qualitative approach to selecting from a range (it provides visual feedback of an approximate nature) and of the slider as more of a quantitative control (it provides an exact value or a range of values from the range of choices).

An interesting and unique feature of the Slider control is its ability to select either a single value or a range of values. You select a range by setting the SelectRange property to *True* and manipulating the SelStart and SelLength properties to define the range. Microsoft suggests that you programmatically set the range properties when the user holds down the Shift key and moves the slider. The Visual Basic Books Online provides a good working example of this technique using the SelectRange property.

Figure 13-8 shows an imaginary database-filtering application in which the user can select, from a large list of all major cities, cities located in a range of latitudes. The Slider control simplifies this type of range selection, but it can also be used to select a single value from a range, as in setting the volume control of a multimedia device. The online help is the best source of information for the properties and methods of the Slider control.

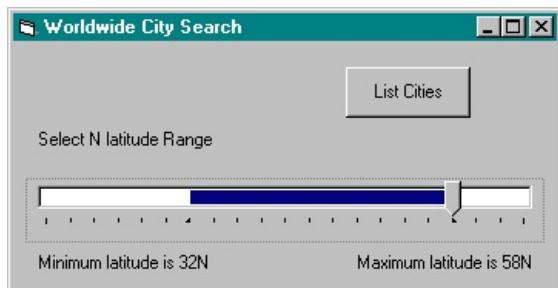


Figure 13-8. A Slider control being used to select a range of values.

Dear John, How Do I... Use the UpDown Control?

The UpDown control is a standard Windows 95 and Windows NT control that makes it easier to increment the value of an associated control. The UpDown control is included in the Windows Common Controls-2 6.0 (MSCOMCT2.OCX) file. Figure 13-9 shows a text box linked to an UpDown control.

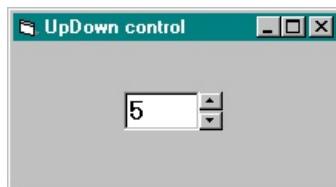


Figure 13-9. An UpDown control being used to increment or decrement the value of a text box.

The UpDown control replaces the scrollbar as a means of incrementing or decrementing a value. In the past, you had to write code to link changes in a scrollbar to the value of a text box. The UpDown control handles this linking automatically through settings on the Buddy tab of its Property Pages dialog box, as shown in Figure 13-10.

To link an UpDown control to the value of another control, follow these steps:

1. After adding the UpDown control to your Toolbox, draw the UpDown control and the control that you want to link to your form.
2. Select the UpDown control, and double-click Custom in the Properties window.
3. Select the Buddy tab of the Property Pages dialog box.
4. In the Buddy Control text box, type the name of the control you want to affect. Alternatively, you could check the AutoBuddy check box, which causes the UpDown control to automatically select the control in the previous tab order as its buddy control.
5. In the Buddy Property drop-down list, select the target control property that you want to affect. You will usually want to affect the Default or Value property of the target control. Notice that after you select a Buddy Property value, the SyncBuddy check box automatically appears checked, which indicates that the Value property of the UpDown control is synchronized with a property of the buddy control.
6. Click OK.

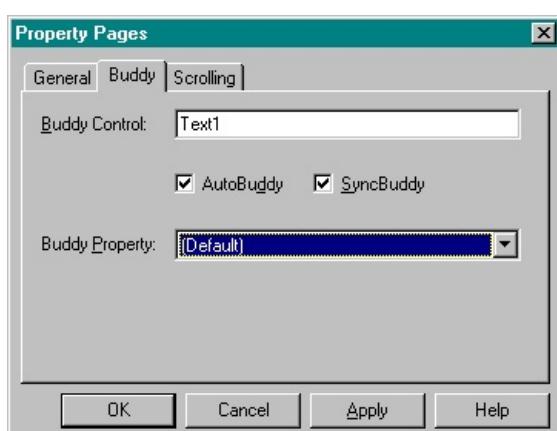


Figure 13-10. Setting the Buddy properties of an UpDown control to change the value of another control.

At runtime, the value of the buddy control is incremented or decremented as you click the UpDown control. Note that you might have to initialize the value of the control—for instance, the default value of *Text1* in a text box control remains *Text1* until you click the UpDown control.

You set the scroll range and rate of the UpDown control on the Scrolling tab of the Property Pages dialog

box, shown in Figure 13-11. The Wrap check box causes the buddy control value to roll over from the maximum to the minimum and vice versa, rather than stopping at the range limits.

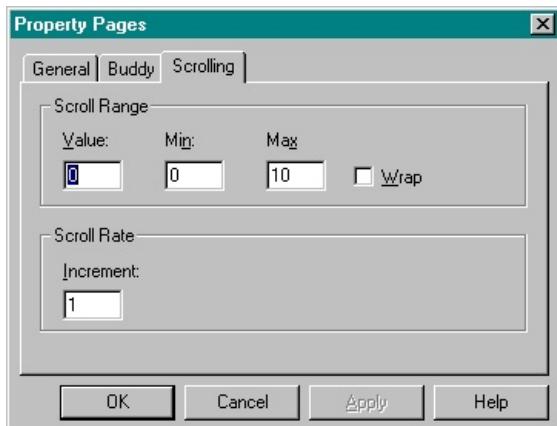


Figure 13-11. Setting the scroll range and rate of the UpDown control.

SEE ALSO

- The Blinker ActiveX control sample in Chapter 6, "[ActiveX Controls](#)," to see how the UpDown control can be used

Dear John, How Do I... Use the FlatScrollBar Controls?

The FlatScrollBar control is an enhanced version of the HScrollBar and VScrollBar controls designed to bring the appearance of your Visual Basic applications up to par with that of Internet Explorer. There's only one FlatScrollBar control because its new Orientation property lets you flip it into either the horizontal or vertical position, rather than requiring a separate control for each of these two positions.

New properties provide more control over the appearance of this type of scrollbar. As already mentioned, the Orientation property displays the FlatScrollBar in either a vertical or horizontal position. The Appearance property provides a flat two-dimensional appearance like that of Internet Explorer, the original three-dimensional appearance identical to the original scrollbars, or a special 2D mode where the arrows appear 3D only while the mouse pointer is moved across them. The Arrows property allows the selective disabling of the arrows at each end of the scrollbar.

A good way to learn how these new properties work is to experiment with them and directly observe the variety of behaviors and appearances they provide. Start with a fresh form, and add a FlatScrollBar control to it. (If the Toolbox doesn't display the FlatScrollBar control, you can add the control to your project by selecting Components from the Project menu and then checking Windows Common Controls-2 6.0 in the Components dialog box.) Add three command buttons, named cmdOrientation, cmdAppearance, and cmdArrows, and set their captions as shown in Figure 13-12. Add a Label control, named lblState, to display the current settings of the three properties we're investigating. The following code ties together all these controls.

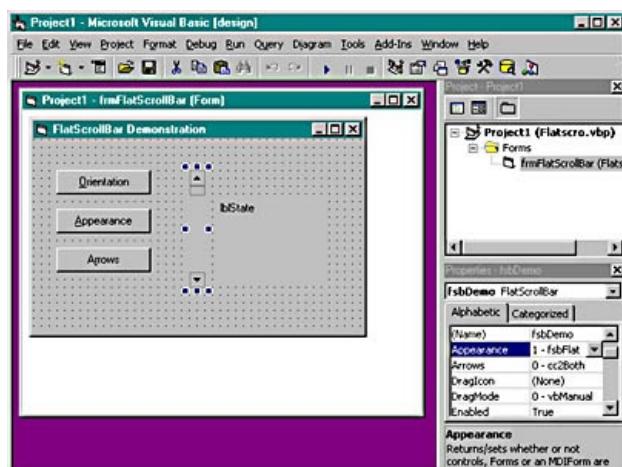


Figure 13-12. The FlatScrollBar demonstration form during development.

```
Option Explicit
```

```
Private Sub cmdAppearance_Click()
    fsbDemo.Appearance = (fsbDemo.Appearance + 1) Mod 3
    Display
End Sub

Private Sub cmdArrows_Click()
    fsbDemo.Arrows = (fsbDemo.Arrows + 1) Mod 3
    Display
End Sub

Private Sub cmdOrientation_Click()
    fsbDemo.Orientation = (fsbDemo.Orientation + 1) Mod 2
    Display
End Sub

Private Sub Display()
    lblState.Caption =
        "Orientation:" & fsbDemo.Orientation & vbCrLf & _
        "Appearance:" & fsbDemo.Appearance & vbCrLf & _
        "Arrows:" & fsbDemo.Arrows
End Sub
```

```
Private Sub Form_Load()
    Display
End Sub
```

Run the application and click the three buttons to cycle through the allowable settings of each of these FlatScrollBar properties. For example, Figure 13-13 shows the FlatScrollBar control's appearance when the form is displayed, and Figure 13-14 shows the control's alternate orientation after the Orientation button is clicked.

When you run the application, you might not notice the difference between an Appearance property setting of 1 and an Appearance property setting of 2—the FlatScrollBar control appears flat in either case. The difference between the two settings is apparent only when the mouse pointer moves over the control's arrows. With the Appearance property set to 1, nothing special happens, but with the Appearance property set to 2, the arrows will suddenly "pop out" or become three-dimensional in appearance as the mouse pointer moves over them.

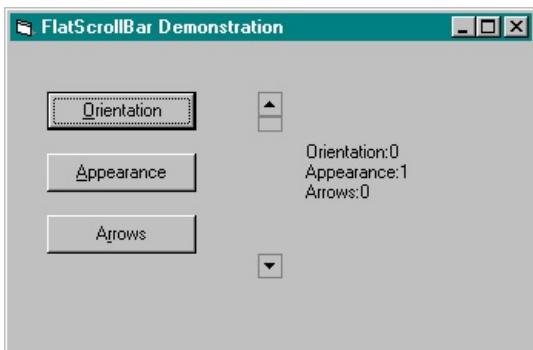


Figure 13-13. The FlatScrollBar control when the form is displayed.

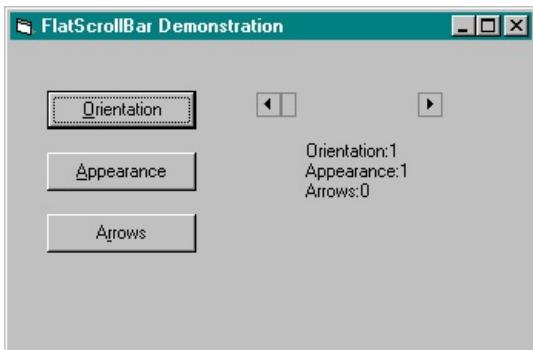


Figure 13-14. The FlatScrollBar control after the Orientation button is clicked.

Dear John, How Do I... Use the CoolBar Control?

The CoolBar control is designed to mimic the behavior of the toolbars in Internet Explorer. A CoolBar control is a container for a collection of Band objects, each of which is a complete little toolbar in itself. Each Band has a move handle on its left edge so that the user can move it around within the coolbar at runtime. If you've never tried to rearrange the Bands within Internet Explorer I suggest you do so now, so you'll better understand how the new CoolBar control works. Notice how, in Figure 13-15 and Figure 13-16 I've rearranged the menus and toolbars (Bands) within Internet Explorer to achieve a completely different order and appearance in my browser.



Figure 13-15. Internet Explorer's menu and toolbar Bands before rearrangement.



Figure 13-16. Internet Explorer's menu and toolbar Bands after some rearrangement.

The CoolBar control has a wide range of properties to allow complete customization. For example, any reasonable number of Bands can be added to a CoolBar control, and the Orientation property allows display of all the Bands in the standard horizontal configuration, or in a nonstandard vertically aligned configuration. The Bands have Index properties to identify each, but they also have a Position property that can change automatically to keep track of the order of the Bands within the CoolBar control as the user shuffles them. Each Band has a Caption property, which displays a text caption just to the right of the move handle. Many of these properties provide modifications beyond those apparent in the Internet Explorer, and experimentation with a CoolBar control is the only way to see how they work.

If you wish to display a tiled background image, either on the CoolBar or on each Band, there are properties that allow this. Some versions of the Internet Explorer, for example, have displayed an interesting but rather faint gray graphic behind the Bands comprising its menus and toolbars. The CoolBar control lets you mimic this appearance if you so desire.

The best way to learn all about the CoolBar control is to draw one on a Visual Basic form and then experiment with the various properties as you read about them in the Visual Basic online help. Be sure to use the Custom property in the Properties window to efficiently interact with many of the CoolBar's properties at design time.

Chapter Fourteen

Graphics Techniques

Visual Basic provides a rich set of graphics tools, including, for instance, the PaintPicture method, which is a user-friendly wrapper around the popular BitBlt Windows API function. You'll find PaintPicture used in several places in this chapter—it's a real workhorse for efficient manipulation of graphics images. You'll also find that I've used BitBlt directly, which is the fastest way to create animation. Several other useful API functions are demonstrated in this chapter, such as those that allow you to create a rubber band selection rectangle, draw polygons efficiently, and perform other graphics magic. Often the best way to implement API functions is to encapsulate them within an object—a technique I've used throughout this chapter.

Dear John, How Do I... Calculate a Color Constant from RGB, HSV, or HSL Values?

In Visual Basic, a color is indicated by a single number, the RGB color value. The bit pattern in this number is composed of three values corresponding to color intensity levels: one for red, one for green, and one for blue (thus the acronym, RGB), each ranging from 0 through 255. The RGB function lets you combine the three intensity levels into the desired color number, but there are no built-in functions for extracting the three intensity levels from a given RGB color value. The code following defines a class that allows you to extract values for red, green, and blue from the combined color value. Add this code to a class module, and name the class *RGB*.

```
'RGB.CLS
Option Explicit

Private mlnColor As Long

'~~~Color
Property Let Color(lngColor As Long)
    mlnColor = lngColor
End Property

'~~~Red
Property Get Red() As Byte
    Red = mlnColor And &HFF
End Property

'~~~Green
Property Get Green() As Byte
    Green = (mlnColor \ &H100) And &HFF
End Property

'~~~Blue
Property Get Blue() As Byte
    Blue = (mlnColor \ &H10000) And &HFF
End Property
```

To keep this class simple I've defined the Color property as write-only and the three component color properties as read-only. As an exercise, you might want to make all properties readable and writable.

The following code demonstrates how to use the RGB object to extract the red, green, and blue color components from a color value:

```
Option Explicit

Private Sub Form_Click()
    Dim rgbTest As New RGB
    Dim lColor As Long
    'Combine R,G, and B to create a known color value
    lColor = RGB(17, 53, 220)
    Print "Color:", lColor
    'Use RGB object to extract component colors
    rgbTest.Color = lColor
    Print "Red:", rgbTest.Red
    Print "Green:", rgbTest.Green
    Print "Blue:", rgbTest.Blue
End Sub
```

In addition to the RGB color model, two other common ways to classify colors are by hue, saturation, and value (or HSV) and by hue, saturation, and luminosity (or HSL). For some people, these other models are more intuitive, and for certain types of graphics these values are definitely easier to work with. For example, the colors required for a sunset scene might be easier to describe as a group of red colors that vary in value but remain constant in hue and saturation. Hue represents the relative position of a color in the spectrum and, in the HSV system, corresponds to the angle of the color on a color wheel. The range for the hue values is 0 through 360 (360 degrees forming a complete circle). Saturation specifies the

purity of the color. The saturation value is a percentage, ranging from 0 (no color) through 100 (the pure color, as specified by the hue value). Value specifies the brightness of the color and is also a percentage, ranging from 0 (black) through 100 (white). Here is a useful class for converting between RGB and HSV values:

```
'HSV.CLS
Option Explicit

`RGB color properties
Private mintRed As Integer
Private mintGreen As Integer
Private mintBlue As Integer

`HSV color properties
Private msngHue As Single
Private msngSaturation As Single
Private msngValue As Single

`Keep track of implied conversion
Private mintCalc As Integer
Private Const RGB2HSV = 1
Private Const HSV2RGB = 2

`~~~ Hue
Property Let Hue(intHue As Integer)
    msngHue = intHue
    mintCalc = HSV2RGB
End Property
Property Get Hue() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Hue = msngHue
End Property

`~~~ Saturation
Property Let Saturation(intSaturation As Integer)
    msngSaturation = intSaturation
    mintCalc = HSV2RGB
End Property

Property Get Saturation() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Saturation = msngSaturation
End Property

`~~~ Value
Property Let Value(intValue As Integer)
    msngValue = intValue
    mintCalc = HSV2RGB
End Property
Property Get Value() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Value = msngValue
End Property

`~~~ Red
Property Let Red(intRed As Integer)
    mintRed = intRed
    mintCalc = RGB2HSV
End Property
Property Get Red() As Integer
    If mintCalc = HSV2RGB Then CalcRGB
    Red = mintRed
End Property
```

```

`~~~ Green
Property Let Green(intGreen As Integer)
    mintGreen = intGreen
    mintCalc = RGB2HSV
End Property
Property Get Green() As Integer
    If mintCalc = HSV2RGB Then CalcRGB
    Green = mintGreen
End Property

`~~~ Blue
Property Let Blue(intBlue As Integer)
    mintBlue = intBlue
    mintCalc = RGB2HSV
End Property
Property Get Blue() As Integer
    If mintCalc = HSV2RGB Then CalcRGB
    Blue = mintBlue
End Property

`Converts RGB to HSV
Private Sub CalcHSV()
    Dim sngRed As Single
    Dim sngGreen As Single
    Dim sngBlue As Single
    Dim sngMx As Single
    Dim sngMn As Single
    Dim sngDelta As Single
    Dim sngVa As Single
    Dim sngSa As Single
    Dim sngRc As Single
    Dim sngGc As Single
    Dim sngBc As Single
    sngRed = mintRed / 255
    sngGreen = mintGreen / 255
    sngBlue = mintBlue / 255
    sngMx = sngRed
    If sngGreen > sngMx Then sngMx = sngGreen
    If sngBlue > sngMx Then sngMx = sngBlue
    sngMn = sngRed
    If sngGreen < sngMn Then sngMn = sngGreen
    If sngBlue < sngMn Then sngMn = sngBlue
    sngDelta = sngMx - sngMn
    sngVa = sngMx
    If sngMx Then
        sngSa = sngDelta / sngMx
    Else
        sngSa = 0
    End If
    If sngSa = 0 Then
        msngHue = 0
    Else
        sngRc = (sngMx - sngRed) / sngDelta
        sngGc = (sngMx - sngGreen) / sngDelta
        sngBc = (sngMx - sngBlue) / sngDelta
        Select Case sngMx
        Case sngRed
            msngHue = sngBc - sngGc
        Case sngGreen
            msngHue = 2 + sngRc - sngBc
        Case sngBlue
            msngHue = 4 + sngGc - sngRc
        End Select
    End If
End Sub

```

```
    msngHue = msngHue * 60
    If msngHue < 0 Then msngHue = msngHue + 360
End If

    msngSaturation = sngSa * 100
    msngValue = sngVa * 100
    mintCalc = 0
End Sub

`Converts HSV to RGB
Private Sub CalcRGB()
    Dim sngSaturation As Single
    Dim sngValue As Single
    Dim sngHue As Single
    Dim intI As Integer
    Dim sngF As Single
    Dim sngP As Single
    Dim sngQ As Single
    Dim sngT As Single
    Dim sngRed As Single
    Dim sngGreen As Single
    Dim sngBlue As Single
    sngSaturation = msngSaturation / 100
    sngValue = msngValue / 100
    If msngSaturation = 0 Then
        sngRed = sngValue
        sngGreen = sngValue
        sngBlue = sngValue
    Else
        sngHue = msngHue / 60
        If sngHue = 6 Then sngHue = 0
        intI = Int(sngHue)
        sngF = sngHue - intI
        sngP = sngValue * (1! - sngSaturation)
        sngQ = sngValue * (1! - (sngSaturation * sngF))
        sngT = sngValue * (1! - (sngSaturation * (1! - sngF)))
        Select Case intI
            Case 0
                sngRed = sngValue
                sngGreen = sngT
                sngBlue = sngP
            Case 1
                sngRed = sngQ
                sngGreen = sngValue
                sngBlue = sngP
            Case 2
                sngRed = sngP
                sngGreen = sngValue
                sngBlue = sngT
            Case 3
                sngRed = sngP
                sngGreen = sngQ
                sngBlue = sngValue
            Case 4
                sngRed = sngT
                sngGreen = sngP
                sngBlue = sngValue
            Case 5
                sngRed = sngValue
                sngGreen = sngP
                sngBlue = sngQ
        End Select
    End If
```

```

mintRed = Int(255.9999 * sngRed)
mintGreen = Int(255.9999 * sngGreen)
mintBlue = Int(255.9999 * sngBlue)
mintCalc = 0
End Sub

```

The HSL model is used in the Microsoft Paint program to select custom colors from a full palette. This model is very similar to HSV, except that the luminosity component in the HSL model is handled in a slightly different mathematical way than is the value component in HSV. Here's the class that provides conversion between the RGB and HSL color models:

```

`HSL.CLS
Option Explicit

`RGB color properties
Private mintRed As Integer
Private mintGreen As Integer
Private mintBlue As Integer

`HSL color properties
Private msngHue As Single
Private msngSaturation As Single
Private msngLuminosity As Single
`Keep track of implied conversion
Private mintCalc As Integer
Private Const RGB2HSL = 1
Private Const HSL2RGB = 2

`~~~ Hue
Property Let Hue(intHue As Integer)
    msngHue = (intHue / 240!) * 360!
    mintCalc = HSL2RGB
End Property
Property Get Hue() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Hue = (msngHue / 360!) * 240!
End Property

`~~~ Saturation
Property Let Saturation(intSaturation As Integer)
    msngSaturation = intSaturation / 240!
    mintCalc = HSL2RGB
End Property
Property Get Saturation() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Saturation = msngSaturation * 240!
End Property

`~~~ Luminosity
Property Let Luminosity(intLuminosity As Integer)
    msngLuminosity = intLuminosity / 240!
    mintCalc = HSL2RGB
End Property
Property Get Luminosity() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Luminosity = msngLuminosity * 240!
End Property

`~~~ Red
Property Let Red(intRed As Integer)
    mintRed = intRed
    mintCalc = RGB2HSL
End Property
Property Get Red() As Integer

```

```
- If mintCalc = HSL2RGB Then CalcRGB
  Red = mintRed
End Property

`~~~ Green
Property Let Green(intGreen As Integer)
  mintGreen = intGreen
  mintCalc = RGB2HSL
End Property

Property Get Green() As Integer
  If mintCalc = HSL2RGB Then CalcRGB
  Green = mintGreen
End Property

`~~~ Blue
Property Let Blue(intBlue As Integer)
  mintBlue = intBlue
  mintCalc = RGB2HSL
End Property
Property Get Blue() As Integer
  If mintCalc = HSL2RGB Then CalcRGB
  Blue = mintBlue
End Property

Private Sub CalcHSL()
  Dim sngMx As Single
  Dim sngMn As Single
  Dim sngDelta As Single
  Dim sngPctRed As Single
  Dim sngPctGrn As Single
  Dim sngPctBlu As Single
  sngPctRed = mintRed / 255
  sngPctGrn = mintGreen / 255
  sngPctBlu = mintBlue / 255
  sngMx = sngMaxOf(sngMaxOf(sngPctRed, sngPctGrn), sngPctBlu)
  sngMn = sngMinOf(sngMinOf(sngPctRed, sngPctGrn), sngPctBlu)
  sngDelta = sngMx - sngMn
  msngLuminosity = (sngMx + sngMn) / 2
  If sngMx = sngMn Then
    msngSaturation = 0
  Else
    msngSaturation = 1
  End If
  If msngLuminosity <= 0.5 Then
    If msngSaturation > 0 Then
      msngSaturation = sngDelta / (sngMx + sngMn)
    End If
  Else
    If msngSaturation > 0 Then
      msngSaturation = sngDelta / (2 - sngMx - sngMn)
    End If
  End If

  If msngSaturation Then
    If sngPctRed = sngMx Then
      msngHue = (sngPctGrn - sngPctBlu) / sngDelta
    End If
    If sngPctGrn = sngMx Then
      msngHue = 2 + (sngPctBlu - sngPctRed) / sngDelta
    End If
    If sngPctBlu = sngMx Then
      msngHue = 4 + (sngPctRed - sngPctGrn) / sngDelta
    End If
  End If

```

```

        msngHue = msngHue * 60
    End If
    If msngHue < 0 Then msngHue = msngHue + 360
    mintCalc = 0
End Sub

Private Sub CalcRGB()
    Dim sngM1 As Single
    Dim sngM2 As Single
    Dim sngPctRed As Single
    Dim sngPctGrn As Single
    Dim sngPctBlu As Single
    If msngLuminosity <= 0.5 Then
        sngM2 = msngLuminosity * (1! + msngSaturation)
    Else
        sngM2 = (msngLuminosity + msngSaturation) -
            (msngLuminosity * msngSaturation)
    End If
    sngM1 = 2! * msngLuminosity - sngM2
    If msngSaturation = 0! Then
        sngPctRed = msngLuminosity
        sngPctGrn = msngLuminosity
        sngPctBlu = msngLuminosity
    Else
        sngPctRed = rgbVal(sngM1, sngM2, msngHue + 120!)
        sngPctGrn = rgbVal(sngM1, sngM2, msngHue)
        sngPctBlu = rgbVal(sngM1, sngM2, msngHue - 120!)
    End If
    mintRed = Int(255.9999 * sngPctRed)
    mintGreen = Int(255.9999 * sngPctGrn)
    mintBlue = Int(255.9999 * sngPctBlu)
    mintCalc = 0
End Sub

Private Function rgbVal(sngN1 As Single, sngN2 As Single, _
sngHue As Single) As Single
    If sngHue > 360 Then
        sngHue = sngHue - 360
    ElseIf sngHue < 0 Then
        sngHue = sngHue + 360
    End If
    If sngHue < 60 Then
        rgbVal = sngN1 + (sngN2 - sngN1) * sngHue / 60
    ElseIf sngHue < 180 Then
        rgbVal = sngN2
    ElseIf sngHue < 240 Then
        rgbVal = sngN1 + (sngN2 - sngN1) * (240 - sngHue) / 60
    Else
        rgbVal = sngN1
    End If
End Function

Private Function sngMaxOf(sngV1 As Single, sngV2 As Single) As Single
    sngMaxOf = IIf(sngV1 > sngV2, sngV1, sngV2)
End Function

Private Function sngMinOf(sngV1 As Single, sngV2 As Single) As Single
    sngMinOf = IIf(sngV1 < sngV2, sngV1, sngV2)
End Function

```

To use the HSV or HSL class in your own application, create an instance of it, set the Red/Green/Blue, the Hue/Saturation/Value, or the Hue/Saturation/Luminosity set of properties to known values, and then

read the properties desired for the other color model. The HSVHSL application in Chapter 29, "[Graphics](#)," provides a complete working example of the HSV and HSL objects.

To see the HSL color scheme in action, take a look at the colors available in Windows 95 and Windows NT 4. In the Control Panel, double-click the Display icon and select the Appearance tab of the Display Properties dialog box. Click the Color button, which is to the right of the Item drop-down list. In the list of colors that is displayed, click the Other button. The resulting Color dialog box lets you choose from a range of colors and shows integer values for RGB and HSL. The HSL color model that Microsoft chose uses integer values ranging from 0 through 240, so the HSL class presented here was developed to use the same ranges.

Dear John, How Do I... Convert Between Twips, Points, Pixels, Characters, Inches, Millimeters, and Centimeters?

A Form, a PictureBox, or a Printer object can be scaled using the ScaleMode property. You can scale these objects by custom units or by a close representation of twips, points, pixels, characters, inches, millimeters, or centimeters. I say "close representation" because for many systems Windows can only approximate these units for your display. When used with the Printer object and printed on a high-quality printer, these dimensional units are usually represented much more accurately. The following list shows the relationships between some of these units of measure:

- 1440 twips per inch
- 567 twips per centimeter
- 72 points per inch
- 2.54 centimeters per inch
- 10 millimeters per centimeter

The character unit is special in that it has one measurement in the horizontal direction and a different measurement in the vertical direction:

- 120 twips per character in the horizontal direction
- 240 twips per character in the vertical direction

Two very useful Visual Basic properties, shown in the following table, help you determine the number of twips per pixel for an object. This value can vary widely based on the actual pixel resolution of your display. Again, the number of pixels in the horizontal direction might not be the same number as in the vertical direction, so there are two similar properties for the two directions.

Property	Return Value
TwipsPerPixelX	Twips per pixel in the horizontal direction
TwipsPerPixelY	Twips per pixel in the vertical direction

By combining these properties and the relationships defined above, you can easily convert between any of the various units of measurement.

Dear John, How Do I... Create One of Those Backgrounds That Fade from Blue to Black?

The following code paints the background of a form with boxes of varying shades of blue, from bright blue to black. The trickiest part is getting a continuous and smooth fade effect that works in 256-color mode as well as in the high-color (16-bit) and true-color (24-bit) modes. In 256-color mode, dithering is required for the smooth transition from bright blue to black. Visual Basic's Line method does not allow dithered colors for a straight line, but it does allow a box to be filled with a dithered "solid" color. To accomplish this, use the following procedure to change the form's DrawStyle property to *vbInvisible* and the form's ScaleMode property to *vbPixels*. The DrawStyle property determines the line style, and setting it to *vbInvisible* prevents a black border from being drawn around each blue box. Setting the ScaleMode property to *vbPixels* lets us calculate the dimensions for each box in pixels with no round-off errors; this prevents overlap or blank spaces between the boxes.

```
Option Explicit
```

```
Private Sub Form_Paint()
    Dim lngY As Long
    Dim lngScaleHeight As Long
    Dim lngScaleWidth As Long
    ScaleMode = vbPixels
    lngScaleHeight = ScaleHeight
    lngScaleWidth = ScaleWidth
    DrawStyle = vbInvisible
    FillStyle = vbFSSolid
    For lngY = 0 To lngScaleHeight
        FillColor = RGB(0, 0, 255 - (lngY * 255) \ lngScaleHeight)
        Line (-1, lngY - 1)-(lngScaleWidth, lngY + 1), , B
    Next lngY
End Sub
```

This procedure fills the form with shades of blue, no matter what size the form is. To create a dramatic full-screen backdrop, set the form's BorderStyle property to 0 - *None* and its WindowState property to 2 - *Maximized*.

There's a lot of room for you to experiment with this procedure. The FillColor calculation can be modified to produce shades of red instead of blue, for instance. Or you might consider reversing the colors so that bright blue is at the bottom and black is at the top. Figure 14-1 demonstrates the fading effect.

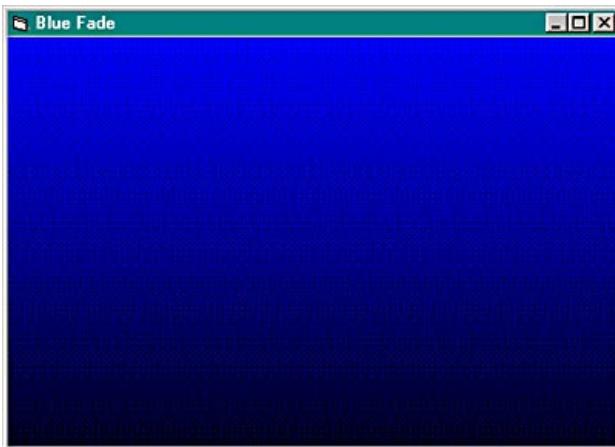


Figure 14-1. A form that fades from blue to black.

SEE ALSO

- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for a demonstration of a fading screen

Dear John, How Do I... Create a Rubber Band Selection Rectangle?

The DrawFocusRect Windows API function is great for drawing a rubber band selection rectangle. The following code demonstrates its use. Create a new form, set its AutoRedraw property to *True*, and add these lines of code to try it out. When you run the program, a selection rectangle is created as you hold down the left mouse button and drag the mouse. When you release the button, the selection rectangle is replaced by a permanent red rectangle to indicate the final selection area.

```
Option Explicit

Private Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type

Private Declare Function DrawFocusRect _
Lib "user32" (
    ByVal hdc As Long,
    lpRect As RECT
) As Long

Private FocusRec As RECT
Private sngX1 As Single
Private sngY1 As Single
Private sngX2 As Single
Private sngY2 As Single

Private Sub Form_Load()
    'Use units expected by the API function
    Me.ScaleMode = vbPixels
End Sub

Private Sub Form_MouseDown( _
    Button As Integer,
    Shift As Integer,
    X As Single,
    Y As Single
)
    'Be sure left mouse button is used
    If (Button And vbLeftButton) = 0 Then Exit Sub
    'Set starting corner of box
    sngX1 = X
    sngY1 = Y
End Sub

Private Sub FormMouseMove( _
    Button As Integer,
    Shift As Integer,
    X As Single,
    Y As Single
)
    'Be sure left mouse button is pressed
    If (Button And vbLeftButton) = 0 Then Exit Sub
    'Erase focus rectangle if it exists
    If (sngX2 <> 0) Or (sngY2 <> 0) Then
        DrawFocusRect Me.hdc, FocusRec
    End If

    'Update coordinates
    sngX2 = X
    sngY2 = Y

```

```

`Update rectangle
FocusRec.Left = sngX1
FocusRec.Top = sngY1
FocusRec.Right = sngX2
FocusRec.Bottom = sngY2
`Adjust rectangle if reversed
If sngY2 < sngY1 Then Swap FocusRec.Top, FocusRec.Bottom
If sngX2 < sngX1 Then Swap FocusRec.Left, FocusRec.Right
`Draw focus rectangle
DrawFocusRect Me.hdc, FocusRec
Refresh
End Sub

Private Sub Form_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single _
)
    `Be sure left mouse button is pressed
    If (Button And vbLeftButton) = 0 Then Exit Sub
    `Erase focus rectangle if it exists
    If FocusRec.Right Or FocusRec.Bottom Then
        DrawFocusRect Me.hdc, FocusRec
    End If
    `Draw indicated rectangle in red
    Line (sngX1, sngY1)-(sngX2, sngY2), vbRed, B
    `Zero the rectangle coordinates
    sngX1 = 0
    sngY1 = 0
    sngX2 = 0
    sngY2 = 0
End Sub

Private Sub Swap(vntA As Variant, vntB As Variant)
    Dim vntT As Variant
    vntT = vntA
    vntA = vntB
    vntB = vntT
End Sub

```

Figure 14-2 shows the program in action.

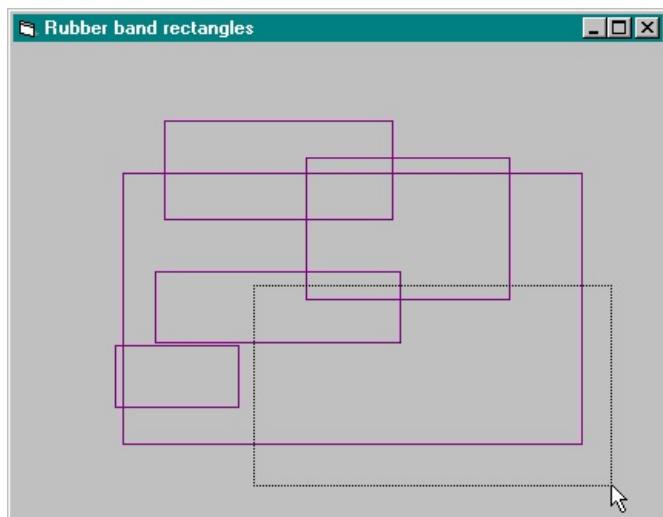


Figure 14-2. An interactive rubber band selection rectangle in action.

The DrawFocusRect Windows API function draws a dotted selection rectangle. As the mouse is dragged, your code must erase each selection rectangle before the next is drawn. Fortunately, the built-in Xor

action of the DrawFocusRect API function makes this easy to do. For example, if you call the DrawFocusRect function a second time using the same coordinates, the selection rectangle will be erased. In the Form_MouseMove event procedure above, you can see that the DrawFocusRect function is called twice, once to erase the previous rectangle and once to draw the next one.

Mouse actions in Visual Basic are separated into up, down, and move events. The rubber band selection program uses all three event-driven procedures. MouseDown indicates the start of a rectangle selection, MouseMove indicates the sizing of the rectangle while the mouse button is held down, and MouseUp indicates the completion of the selection.

In this example, only mouse actions on the form itself, and not on any controls it contains, can create a selection rectangle. However, you can draw selection rectangles on any picture box or form by adding the same code to its mouse event procedures. The preceding code will give you a good start on integrating selection rectangle capabilities into your own applications.

Dear John, How Do I... Create Graphics Hot Spots?

The Image control is an efficient tool with which to add rectangular hot spot regions to your graphics. Let's walk through an example to see how it's done.

I first loaded a graphic named WORLD.BMP into an Image control named *imgWorld*, and then I carefully drew four more Image controls on top of the image of the world. I named these *imgNAmerica*, *imgSAmerica*, *imgEurope*, and *imgAfrica*. Each rectangle covers the appropriate part of the world. I then added a Label control named *lblHotSpots* to the form.

Figure 14-3 shows a sample of WORLD.BMP with four Image controls.

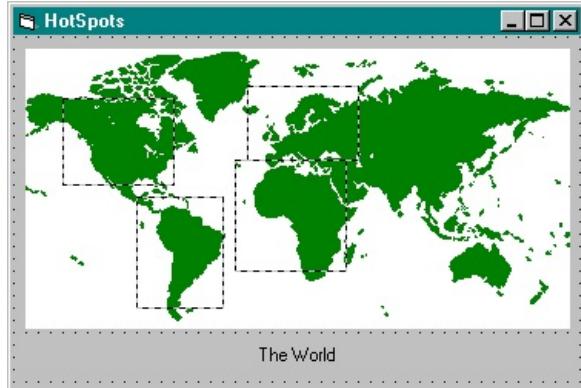


Figure 14-3. Hot spots shown during application development.

The following code completes the demonstration by displaying a label for each hot spot when the hot spot is clicked:

```
Option Explicit

Private Sub imgAfrica_Click()
    lblHotSpots.Caption = "Africa"
End Sub

Private Sub imgEurope_Click()
    lblHotSpots.Caption = "Europe"
End Sub

Private Sub imgNorthAmerica_Click()
    lblHotSpots.Caption = "North America"
End Sub

Private Sub imgSouthAmerica_Click()
    lblHotSpots.Caption = "South America"
End Sub

Private Sub imgWorld_Click()
    lblHotSpots.Caption = "The World"
End Sub
```

Figure 14-4 shows the result when the Africa hot spot rectangle is clicked.

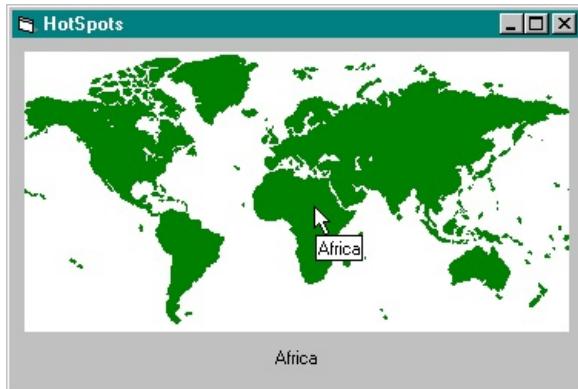


Figure 14-4. The Africa hot spot activated with a click.

The Image control outlines are visible during development but invisible at runtime. Because these hot spot Image controls are drawn on top of a larger image, click events are acted on by the hot spots rather than by the underlying image.

For a nice interactive touch, I added the same continent names to the ToolTips property of each Image control. This optional step provides a visual indication of each hot spot to users before they click it to select it.

SEE ALSO

- The VBClock application in Chapter 31, "[Date and Time](#)," for a demonstration of graphics hot spots

Dear John, How Do I... Draw a Polygon Quickly?

You can use the Line method to connect a sequence of points with straight lines and return to the starting point to draw a closed polygon. But the Polygon Windows API function is faster and has the advantage of being able to efficiently fill the interior of the polygon with a color. The following Polygon class greatly simplifies the task of drawing polygons by encapsulating the API function and its required data type:

```

`POLYGON.CLS
Option Explicit

Private Type POINTAPI
    x As Long
    y As Long
End Type

Private Declare Function Polygon _
Lib "gdi32" (
    ByVal hdc As Long,
    lpPoint As POINTAPI,
    ByVal nCount As Long
) As Long

`Module-level private variables
Private mobjDevice As Object
Private msngSX1 As Single
Private msngSY1 As Single
Private msngXRatio As Single
Private msngYRatio As Single
Private maPointArray() As POINTAPI

`~~~Device
Property Set Device(objDevice As Object)
    Dim sngSX2 As Single
    Dim sngSY2 As Single
    Dim sngPX2 As Single
    Dim sngPY2 As Single
    Dim intScaleMode As Integer
    Set mobjDevice = objDevice
    With mobjDevice
        `Grab current scaling parameters
        intScaleMode = .ScaleMode
        msngSX1 = .ScaleLeft
        msngSY1 = .ScaleTop
        sngSX2 = msngSX1 + .ScaleWidth
        sngSY2 = msngSY1 + .ScaleHeight
        `Temporarily set pixels mode
        .ScaleMode = vbPixels
        `Grab pixel scaling parameters
        sngPX2 = .ScaleWidth
        sngPY2 = .ScaleHeight
        `Reset user's original scale
        If intScaleMode = 0 Then
            mobjDevice.Scale (msngSX1, msngSY1)-(sngSX2, sngSY2)
        Else
            mobjDevice.ScaleMode = intScaleMode
        End If
        `Calculate scaling ratios just once
        msngXRatio = sngPX2 / (sngSX2 - msngSX1)
        msngYRatio = sngPY2 / (sngSY2 - msngSY1)
    End With
End Property

`~~~Point X,Y
Public Sub Point(sngX As Single, sngY As Single)

```

```

Dim lngN As Long
lngN = UBound(maPointArray) + 1
ReDim Preserve maPointArray(lngN)
maPointArray(lngN).x = XtoP(sngX)
maPointArray(lngN).y = YtoP(sngY)
End Sub

`~~~Draw
Public Sub Draw()
    Polygon mobjDevice.hdc, maPointArray(1), UBound(maPointArray)
    ReDim maPointArray(0)
End Sub

`Scales X value to pixel location
Private Function XtoP(sngX As Single) As Long
    XtoP = (sngX - msngSX1) * msngXRatio
End Function

`Scales Y value to pixel location
Private Function YtoP(sngY As Single) As Long
    YtoP = (sngY - msngSY1) * msngYRatio
End Function
`Initialization
Private Sub Class_Initialize()
    ReDim maPointArray(0)
End Sub

```

I've taken advantage of the behind-the-scenes way Visual Basic handles graphics to make the Polygon object easy to use. The Polygon API function always expects pixel parameters, whereas Visual Basic lets you scale your graphics output device in several different ways. Fortunately, Visual Basic doesn't use the API functions for mapping and scaling graphics at a low level; instead it scales user units to pixels just before calling the API functions for drawing lines, circles, and so on. The Polygon object performs the same type of scaling, which turns out to be an efficient way to allow the user to use Visual Basic's different scaling units. Notice that the scaling factors are calculated just once, in the Set Device property procedure. Then, with each coordinate of the polygon set in the Point method, the X and Y values are scaled to pixels using these factors. The only catch here is that the Device property should always be set immediately after any change to the device's scaling mode or scaling units.

Visual Basic uses standard graphical device contexts and brushes in ways that make it easy to accommodate Visual Basic's FillStyle, FillColor, and similar graphics settings when calling graphics API functions. In fact, nothing special needs to be done within the Polygon object because the Polygon API function automatically uses the output device's current settings. This means, for example, that your form can use the same Polygon object to draw polygons with different fill patterns and fill colors simply by changing the form's FillStyle and FillColor properties.

The following code demonstrates how the Polygon object is used in your applications. In this example, a random 17-point polygon is created and filled with a randomly selected solid color. To try it out, add this code to a new form that contains a PictureBox control named *picTest*. Be sure that the Polygon class is named *Polygon*. Run the program, and click the picture box to draw the polygons.

```

`POLYTEST.FRM
Option Explicit

Dim polyTest As New Polygon

Private Sub Form_Load()
    `Create unique polygon each time
    Randomize

    `Use any desired units and graphics settings
    With picTest
        .Move 0, 0, ScaleWidth, ScaleHeight
        .ScaleMode = vbInches
    End With
End Sub

```

```
.FillStyle = vbSolid
End With
End Sub

Private Sub picTest_Click()
    Dim intI As Integer
    'Connect picture box as polygon output device
    Set polyTest.Device = picTest
    With picTest
        'Clear output with each click
        .Cls
        'Build 17-point random polygon
        For intI = 1 To 17
            polyTest.Point Rnd * .ScaleWidth, Rnd * .ScaleHeight
        Next intI
        'Create unique fill color each time
        .FillColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
        'Draw polygon, filling the interior
        polyTest.Draw
    End With
End Sub
```

Figure 14-5 shows a 17-point polygon created using the Polygon object.

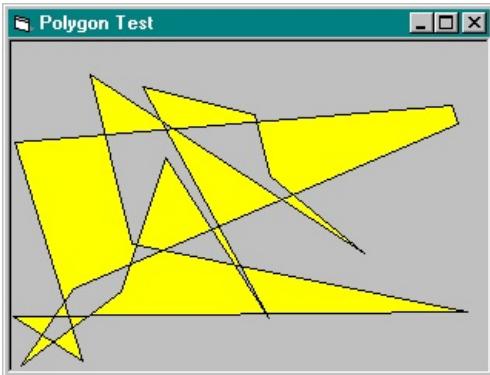


Figure 14-5. A multipointed polygon drawn using the *Polygon* object.

SEE ALSO

- "[Dear John, How Do I... Draw an Ellipse?](#)"
- "[Dear John, How Do I... Fill an Irregularly Shaped Area with a Color?](#)" later in this chapter

Dear John, How Do I... Draw an Ellipse?

Visual Basic's Circle method draws an ellipse using its center coordinate, radius, and aspect ratio. However, in many cases it's necessary to draw an ellipse within a bounding rectangle. Calculating the center coordinates, the radius, and the aspect ratio to precisely position an ellipse within such a bounding rectangle can be done, but the math is tricky and prone to errors.

Fortunately, there's a better way to draw ellipses within bounding rectangles: the Ellipse Windows API function. In fact, Visual Basic itself uses this API function to draw circles and ellipses, converting user information such as the center coordinates, the radius, and the aspect ratio to a bounding rectangle. The following Ellipse class encapsulates the Ellipse API function and calls it directly, making it easy for your applications to draw ellipses using their bounding rectangle dimensions:

```
' ELLIPSE.CLS
Option Explicit

Private Declare Function Ellipse _
Lib "gdi32" (
    ByVal hdc As Long, _
    ByVal X1 As Long, _
    ByVal Y1 As Long, _
    ByVal X2 As Long, _
    ByVal Y2 As Long _
) As Long

`Module-level private variables
Private mobjDevice As Object
Private msngSX1 As Single
Private msngSY1 As Single
Private msngXRatio As Single
Private msngYRatio As Single

`~~~Device
Property Set Device(objDevice As Object)
    Dim sngSX2 As Single
    Dim sngSY2 As Single
    Dim sngPX2 As Single
    Dim sngPY2 As Single
    Dim sngScaleMode As Integer
    Set mobjDevice = objDevice
    With mobjDevice
        `Grab current scaling parameters
        sngScaleMode = .ScaleMode
        msngSX1 = .ScaleLeft
        msngSY1 = .ScaleTop
        sngSX2 = msngSX1 + .ScaleWidth
        sngSY2 = msngSY1 + .ScaleHeight
        `Temporarily set pixels mode
        .ScaleMode = vbPixels
        `Grab pixel scaling parameters
        sngPX2 = .ScaleWidth
        sngPY2 = .ScaleHeight
        `Reset user's original scale
        If sngScaleMode = 0 Then
            mobjDevice.Scale (msngSX1, msngSY1)-(sngSX2, sngSY2)
        Else
            mobjDevice.ScaleMode = sngScaleMode
        End If
        `Calculate scaling ratios just once
        msngXRatio = sngPX2 / (sngSX2 - msngSX1)
        msngYRatio = sngPY2 / (sngSY2 - msngSY1)
    End With
End Property
```

```
'~~~Draw X1,Y1,X2,Y2
Public Sub Draw(
    sngX1 As Single, _
    sngY1 As Single, _
    sngX2 As Single, _
    sngY2 As Single _
)
    Ellipse mobjDevice.hdc, XtoP(sngX1), YtoP(sngY1), _
        XtoP(sngX2), YtoP(sngY2)
End Sub

`Scales X value to pixel location
Private Function XtoP(sngX As Single) As Long
    XtoP = (sngX - msngSX1) * msngXRatio
End Function

`Scales Y value to pixel location
Private Function YtoP(sngY As Single) As Long
    YtoP = (sngY - msngSY1) * msngYRatio
End Function
```

Notice that much of the code in the Ellipse class module is the same as that in the Polygon class described earlier in this chapter. In particular, the Device property performs the same scaling preparations. You can use the Ellipse (or the Polygon) class module as a template to create other graphics objects based on the many additional graphics API functions. Also, you might want to include these various API functions as methods in a single, all-encompassing Graphics class. The Graphics object would then have one Device property and multiple drawing methods, such as Polygon and Ellipse. I've chosen to place these API functions in separate class modules in order to focus clearly on the implementation of each object. If you plan to use only one or two of these new objects in a given application, these smaller, individualized graphics objects provide an efficient option.

The following code demonstrates the Ellipse object by drawing three nested ellipses, each just touching the edge of another ellipse. This is surprisingly difficult to accomplish using Visual Basic's Circle method, because of the way the aspect ratio scales ellipses. To try out the Ellipse object, add the following code to a form that contains a PictureBox control named *picTest*. Be sure that the Ellipse class is named *Ellipse*. Run the program, and click the picture box to draw the ellipses.

```
`ELLITEST
Option Explicit

Dim ellipseTest As New Ellipse

Private Sub picTest_Click()
    picTest.ScaleMode = vbCentimeters
    Set ellipseTest.Device = picTest
    ellipseTest.Draw 1, 1, 7, 4
    ellipseTest.Draw 2, 1, 6, 4
    ellipseTest.Draw 2, 2, 6, 3
End Sub
```

Figure 14-6 shows the three nested ellipses.

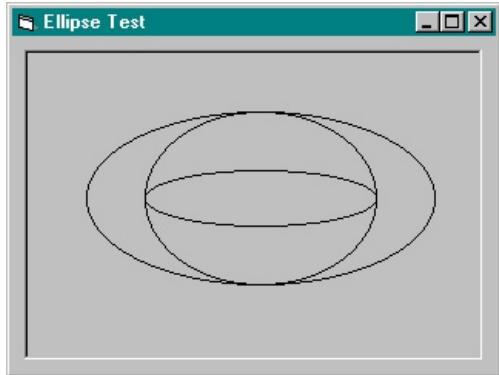


Figure 14-6. Nested ellipses defined by bounding rectangle coordinates.

SEE ALSO

- "[Dear John, How Do I... Draw a Polygon Quickly?](#)" earlier in this chapter

Dear John, How Do I... Fill an Irregularly Shaped Area with a Color?

The Polygon Windows API function described earlier in this chapter is great for filling an area with a color if you know the boundary coordinates of the area. Sometimes, however, you don't know the boundary coordinates but do know the boundary color, and you can use this information to fill the area. The FloodFill Windows API function simulates the old Paint command found in earlier versions of Basic, which floods an area bounded by a color.

The following class and sample code demonstrates the use of the FloodFill API function to fill the area formed by an overlapping circle and square with color:

```
'PAINT.CLS
Option Explicit
```

```
Private Declare Function FloodFill _  
Lib "gdi32" ( _  
    ByVal hdc As Long, _  
    ByVal x As Long, _  
    ByVal y As Long, _  
    ByVal crColor As Long _  
) As Long

`Module-level private variables  
Private mobjDevice As Object  
Private msngSX1 As Single  
Private msngSY1 As Single  
Private msngXRatio As Single  
Private msngYRatio As Single

Property Set Device(objDevice As Object)  
    Dim sngSX2 As Single  
    Dim sngSY2 As Single  
    Dim sngPX2 As Single  
    Dim sngPY2 As Single  
    Dim nScaleMode As Integer  
    Set mobjDevice = objDevice  
    With mobjDevice  
        `Grab current scaling parameters  
        nScaleMode = .ScaleMode  
        msngSX1 = .ScaleLeft  
        msngSY1 = .ScaleTop  
        sngSX2 = msngSX1 + .ScaleWidth  
        sngSY2 = msngSY1 + .ScaleHeight  
        `Temporarily set pixels mode  
        .ScaleMode = vbPixels  
        `Grab pixel scaling parameters  
        sngPX2 = .ScaleWidth  
        sngPY2 = .ScaleHeight  
        `Reset user's original scale  
        If nScaleMode = 0 Then  
            mobjDevice.Scale (msngSX1, msngSY1)-(sngSX2, sngSY2)  
        Else  
            mobjDevice.ScaleMode = nScaleMode  
        End If  
        `Calculate scaling ratios just once  
        msngXRatio = sngPX2 / (sngSX2 - msngSX1)  
        msngYRatio = sngPY2 / (sngSY2 - msngSY1)  
    End With  
End Property
```

```

`~~~Flood x,y
Public Sub Flood(sngX As Single, sngY As Single)
    FloodFill mobjDevice.hdc, XtoP(sngX), YtoP(sngY), _
        mobjDevice.ForeColor
End Sub

`Scales X value to pixel location
Private Function XtoP(sngX As Single) As Long
    XtoP = (sngX - msngSX1) * msngXRatio
End Function

`Scales Y value to pixel location
Private Function YtoP(sngY As Single) As Long
    YtoP = (sngY - msngSY1) * msngYRatio
End Function

```

The code that outlines the circle and the square in red and fills the overlapping area with solid green is as follows:

```

Option Explicit

Dim paintTest As New Paint

Private Sub picTest_Click()
    picTest.ScaleMode = vbInches
    `Draw overlapping box and circle in red
    picTest.Line (0.5, 0.5)-(2, 2), vbRed, B
    picTest.Circle (2, 1), 0.7, vbRed
    `Prepare to paint the overlapping area
    picTest.FillStyle = vbFSSolid      `Paint style
    picTest.FillColor = vbGreen       `Paint color
    picTest.ForeColor = vbRed         `Paint boundary color
    Set paintTest.Device = picTest
    paintTest.Flood 1.7, 0.9
    `Reset fill style to default
    picTest.FillStyle = vbFTransparent
End Sub

```

Notice that the color boundary is defined by the current ForeColor property and that the fill color itself is defined by the current FillStyle and FillColor properties. The paint fill starts at the specified xy-coordinates, filling all pixels with the given fill color and using the given fill style, until pixels with the ForeColor boundary color are reached. This can be very handy for coloring irregularly shaped areas, such as the overlapping circle and square in this example.

To test this example, you will need to name the Paint class *Paint* and add the previous code to a form that contains a PictureBox control named *picTest*. Figure 14-7 shows the results of this painting process when the picture box is clicked.

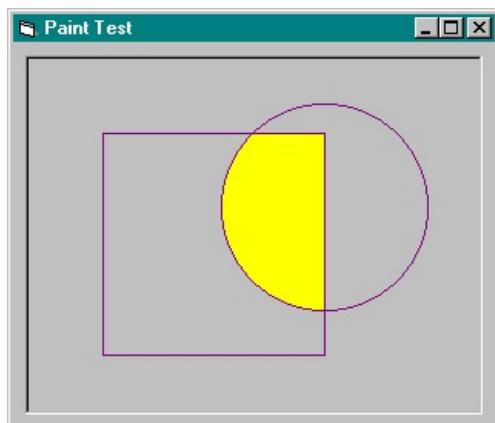


Figure 14-7. An irregularly shaped area painted using the *Paint* object.

SEE ALSO

- "[Dear John, How Do I... Draw a Polygon Quickly?](#)" earlier in this chapter

Dear John, How Do I... Rotate a Bitmap?

The 32-bit function PlgBlt is designed to help you create a generalized polygon transformation of a picture. But although it's been documented in Windows 95, the function has not yet been implemented. Nothing happens when I try to call the PlgBlt function, which means that it's still just a stub function, to be implemented later. Stay tuned—this function should be much more efficient than the technique I'll show you here.

The code below will rotate a picture any number of degrees while copying it, pixel by pixel, from one picture box to another. The code is slow, so you'll probably want to use it to rotate individual images while you are building a set of images for an application instead of using it to rotate the images in an application while the application is running.

The Point and PSet methods are used to read and write the pixels, and this short demonstration provides a good example of their use. Be sure to set the ScaleMode to *vbPixels*, as shown in the following code, before using these functions on graphics images.

To try this technique, start a new project and add two picture boxes named *picOne* and *picTwo* and a command button named *cmdRotate* to the form. Assign a bitmap picture to the Picture property of *picOne*, and size *picOne* to display the bitmap. Only the part of the bitmap that is displayed in the picture box will be copied. Size *picTwo* appropriately, and add the code below to the form. The code sets the rotation angle to 45 degrees, but you can set it to any angle. When you run the program, click the command button to start the rotating and copying process.

```
Option Explicit
```

```
Const PI = 3.14159265358979
Const ANGLE = 45
```

```
Private Sub cmdRotate_Click()
    Dim intX As Integer
    Dim intY As Integer
    Dim intX1 As Integer
    Dim intY1 As Integer
    Dim dblX2 As Double
    Dim dblY2 As Double
    Dim dblX3 As Double
    Dim dblY3 As Double
    Dim dblThetaDeg As Double
    Dim dblThetaRad As Double
    'Initialize rotation angle
    dblThetaDeg = ANGLE
    'Compute angle in radians
    dblThetaRad = dblThetaDeg * PI / 180
    'Set scale modes to pixels
    picOne.ScaleMode = vbPixels
    picTwo.ScaleMode = vbPixels
    For intX = 0 To picTwo.ScaleWidth
        intX1 = intX - picTwo.ScaleWidth \ 2
        For intY = 0 To picTwo.ScaleHeight
            intY1 = intY - picTwo.ScaleHeight \ 2
            'Rotate picture by dblThetaRad
            dblX2 = intX1 * Cos(-dblThetaRad) + _
                    intY1 * Sin(-dblThetaRad)
            dblY2 = intY1 * Cos(-dblThetaRad) - _
                    intX1 * Sin(-dblThetaRad)
            'Translate to center of picture box
            dblX3 = dblX2 + picOne.ScaleWidth \ 2
            dblY3 = dblY2 + picOne.ScaleHeight \ 2
            'If data point is in picOne, set its color in picTwo
            If dblX3 > 0 And dblX3 < picOne.ScaleWidth - 1 And dblY3 > 0 And dblY3 < picOne.ScaleHeight - 1 Then
                picTwo.PSet (intX, intY), picOne.Point(dblX3, dblY3)
            End If
        Next intY
    Next intX
End Sub
```

```
    Next intY
    Next intX
End Sub
```

You might want to use the SavePicture statement to save the rotated image as a bitmap file. If you do so, be sure to set the AutoRedraw property of *picTwo* to *True*. The following statement shows an example of how to use the SavePicture statement:

```
SavePicture picTwo.Image, "C:\FINISHROT.BMP"
```

Figure 14-8 shows the demonstration program during development, and Figure 14-9 shows the picture after rotating 45 degrees.



Figure 14-8. A picture before rotation.



Figure 14-9. A picture plus a copy rotated 45 degrees.

Dear John, How Do I... Scroll a Graphics Image?

Visual Basic's PaintPicture method simplifies many graphics manipulation techniques. In the following code, for instance, I use PaintPicture to display a small window of a larger picture, with scrollbars to let the user smoothly scroll the image. As shown in Figure 14-10, a polyconic map of the western hemisphere appears in the full-size picture box, and a copy of part of that image appears in the smaller picture box.

NOTE

The full-size image need not be visible for this program to work. During the development process, you can load the source picture box with a large image and then set its Visible property to *False*. The small picture box will still display the scrollable picture.

To try this program, add two picture boxes of different sizes to a new form. Name the larger picture box *picOne* and the smaller one *picTwo*. Add a vertical scrollbar named *vsbScroll* and a horizontal scrollbar named *hsbScroll* to the form, as shown in Figure 14-10. Position these two scrollbar controls adjacent to *picTwo* because they will control the contents of this picture box. Load an image file into the Picture property of *picOne*, and size the picture box—only the part of the image displayed in the *picOne* picture box can be displayed in the *picTwo* picture box. The program scrolls this image in *picTwo* by copying a rectangular region from *picOne* using the PaintPicture method.

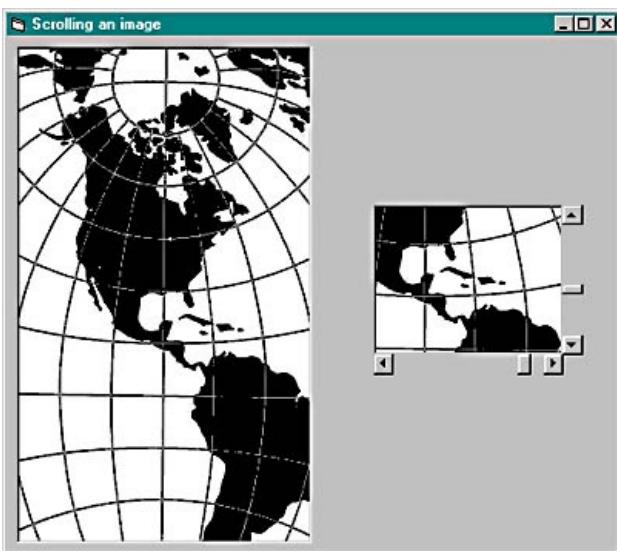


Figure 14-10. Scrolling a large image in a smaller picture box.

Add the following code to the form. Notice that the program triggers scrollbar Change events from within each scrollbar's Scroll event. This lets you drag the scrollbar box and smoothly scroll the image in real time.

```
Option Explicit
```

```
Private Sub Form_Load()
    hsbScroll.Max = picOne.ScaleWidth - picTwo.ScaleWidth
    hsbScroll.LargeChange = hsbScroll.Max \ 10
    hsbScroll.SmallChange = hsbScroll.Max \ 25
    vsbScroll.Max = picOne.ScaleHeight - picTwo.ScaleHeight
    vsbScroll.LargeChange = vsbScroll.Max \ 10
    vsbScroll.SmallChange = vsbScroll.Max \ 25
End Sub

Private Sub hsbScroll_Change()
    UpdatePicTwo
End Sub
```

```
Private Sub hsbScroll_Scroll()
    hsbScroll_Change
End Sub

Private Sub vsbScroll_Change()
    UpdatePicTwo
End Sub

Private Sub vsbScroll_Scroll()
    vsbScroll_Change
End Sub

Private Sub UpdatePicTwo()
    picTwo.PaintPicture picOne.Picture, 0, 0, -
        picTwo.ScaleWidth, picTwo.ScaleHeight, -
        hsbScroll.Value, vsbScroll.Value, -
        picTwo.ScaleWidth, picTwo.ScaleHeight, -
        vbSrcCopy
End Sub
```

Dear John, How Do I... Use BitBlt to Create Animation?

Visual Basic's PaintPicture method is a convenient and fairly fast method for moving irregularly shaped graphics objects around the screen without disturbing the background. PaintPicture is Visual Basic's equivalent of the BitBlt Windows API function. (Actually, after PaintPicture performs error checking and scaling from the current graphics units, it calls BitBlt.) The only drawback to PaintPicture is its slowness. By calling BitBlt directly, you avoid the overhead of the calculations that PaintPicture must perform before it in turn calls BitBlt. The result is an animated sequence created with BitBlt that is considerably faster, with smoother action—that's why the following demonstration program calls BitBlt directly.

The BitBlt API function quickly moves a rectangular block of pixels from a picture box or form to another picture box or form or to the Printer object. BitBlt requires an hDC (handle to a device context) for both the source and the destination of the image transfer. The PictureBox, Form, and Printer objects all provide an hDC property. Note, however, that with the AutoRedraw property set to *True*, the hDC property of a picture box points to the control's Image property, not to its Picture property. The image is a behind-the-scenes copy, stored in memory, of what you see on the screen; it's where the actual work is performed as you draw or use the BitBlt API function to manipulate the contents of a PictureBox control when AutoRedraw is *True*. The results are transferred from the image to the picture only when the system refreshes a picture box. This works well for our purposes, avoiding flicker and other problems.

The code below animates an image of an unidentified flying object (UFO), moving it across the standard Windows 95 cloud bitmap without disturbing this background image. This action requires three picture boxes: one containing the UFO's image, a second containing a mask to prepare the background for the UFO's image, and a third to temporarily hold the background that will be restored when the UFO moves on. Much more complicated objects can be animated using the same technique, but for now I have chosen a simple, small UFO shape.

```
'BITBLT
Option Explicit

Private Declare Function BitBlt _ 
Lib "gdi32" ( _
    ByVal hDestDC As Long, _
    ByVal x As Long, ByVal y As Long, _
    ByVal nWidth As Long, ByVal nHeight As Long, _
    ByVal hSrcDC As Long, _
    ByVal xSrc As Long, ByVal ySrc As Long, _
    ByVal dwRop As Long _ 
) As Long

Private Sub cmdAnimate_Click()
    Static lngX As Long
    Static lngY As Long
    Static lngW As Long
    Static lngH As Long
    Static blnBackSaved As Boolean
    Dim lngRtn As Long
    'Display hourglass pointer while busy
    Screen.MousePointer = vbHourglass
    'Provide starting location
    lngX = -picUfo.ScaleWidth
    lngY = picClouds.ScaleHeight
    'Save sizes in local variables once for speed
    lngW = picUfo.ScaleWidth
    lngH = picUfo.ScaleHeight
    'Loop to animate the UFO
    Do
        'Restore background unless this is first time object is drawn
        If blnBackSaved = True Then
            lngRtn = BitBlt(picClouds.hDC, lngX, lngY, lngW, lngH, _
                picBack.hDC, 0, 0, vbSrcCopy)
        'Stop UFO's motion when it gets to the edges
        If lngX > picClouds.ScaleWidth Then
```

```

        blnBackSaved = False

        picClouds.Refresh
        Exit Do
    End If
End If
`Move UFO to a new location
lngX = lngX + 1
    If lngX < 0.5 * picClouds.ScaleWidth _
        Or lngX > 0.8 * picClouds.ScaleWidth Then
        lngY = lngY - 1
    Else
        lngY = lngY + 1
    End If
`Save background at new location
lngRtn = BitBlt(picBack.hDC, 0, 0, lngW, lngH, _
    picClouds.hDC, lngX, lngY, vbSrcCopy)
blnBackSaved = True
`Apply mask
lngRtn = BitBlt(picClouds.hDC, lngX, lngY, lngW, lngH, _
    picUfoMask.hDC, 0, 0, vbSrcAnd)
`Draw UFO
lngRtn = BitBlt(picClouds.hDC, lngX, lngY, lngW, lngH, _
    picUfo.hDC, 0, 0, vbSrcPaint)
picClouds.Refresh
Loop
`Restore pointer
Screen.MousePointer = vbDefault
End Sub

```

To try out this code, start a new project and add to the form a command button named *cmdAnimate* and four PictureBox controls named *picClouds*, *picUfo*, *picUfoMask*, and *picBack*. Now you need three bitmaps: one large bitmap for the background (I used CLOUDS.BMP, which I found on my Windows 95 installation CD-ROM), one for the UFO, and one for the UFO mask. (I created the UFO and UFO mask bitmaps using Windows Paint.) Set the *picClouds* Picture property to CLOUDS.BMP or another background image, and set the AutoRedraw property to *True*. For the other three PictureBox controls, set the *picUfo* Picture property to the UFO bitmap, set the *picUfoMask* Picture property to the UFO mask bitmap, and set the *picBack* Picture property to the UFO bitmap. Size these three controls so that the three picture boxes are the same size, and position them on the form near the top so that you can follow the action when the program is run. For all four of the PictureBox controls, set the AutoSize property to *True* and the ScaleMode property to 3 - *Pixel*. Figure 14-11 following shows the form during the development phase.



Figure 14-11. The BitBlt Animation example during development.

The UFO and UFO mask bitmaps must be created in a special way. The UFO image here consists of a UFO drawn on a black background. You can use any color except black for an object that is to appear in front of the background. In this case, all the colored pixels that make up the UFO will show. Wherever there are black pixels in an image, however, the background will appear in the animated version. For example, if you want to change the UFO into a doughnut, draw a solid black circle in the center of a nonblack circle.

The UFO mask image is created using the following rule: wherever there are black pixels in the UFO bitmap, make them white in the mask, and wherever there are pixels of any color other than black in the UFO bitmap, make those pixels black in the mask. The mask is a negative image of the primary bitmap and should contain only black and white pixels.

Here's how these bitmaps work to create the animation. First the saved background is restored to erase the UFO at its current location. (Note that this step is skipped the first time the UFO is drawn.) The background at the UFO's next location is saved in the *picBack* picture box. This will be used to restore the background when the UFO moves again. At the new location, the mask image is placed over the background image using a Boolean And operation on a pixel-by-pixel basis. The last parameter in the BitBlt method determines this logical bit-mixing operation. An And operation displays black pixels wherever the applied mask is black and undisturbed background pixels wherever the mask is white. A solid black UFO shape will be displayed on the form at this point, but it will disappear with the next step, before you can see it.

The final step in updating the UFO image to a new location is to apply the primary UFO bitmap (the first picture box) using an Or bitwise operation. The nonblack pixels in the UFO appear where the background is blacked out, and the black pixels leave the background undisturbed. The result is a solid-looking UFO that appears to hover in front of the background scene.

The action is repeated using a loop, causing the UFO to glide across the form from left to right. The UFO is shown in midflight in Figure 14-12.



Figure 14-12. The BitBlt Animation program in progress, showing the UFO as it moves across the background.

If you are having difficulty getting the UFO to mask properly, the problem could be related to your video driver. Try changing to a different video driver to see whether you experience the same problem.

SEE ALSO

- The Lottery application in Chapter 29, "[Graphics](#)," for a demonstration of animation

Dear John, How Do I... Use Picture Objects for Animation?

Picture properties have been included in all previous versions of Visual Basic, and starting with version 4, Visual Basic has provided a companion Picture object, which greatly enhances the options you have for manipulating images. A Picture object is an independent entity that you can use to load and store images—you no longer have to use a PictureBox control or an Image control. When you want to access the images, it's easy to copy them from Picture objects to visible controls.

NOTE

In addition to the Picture object, Visual Basic provides an ImageList control that lets you load multiple images from a resource file. You might want to try the Picture object or the ImageList control to improve the speed and efficiency of your graphics displays.

Here's some example code for you to experiment with. This program creates an array of Picture objects and loads a sequence of images into each when the form loads. A Timer control named *tmrAnimate* is set to the minimum interval of 1 millisecond, the approximate rate at which it will assign the Picture objects sequentially to a PictureBox control named *picTest*. The result is a smooth, flicker-free animation of the images you've created in the sequence of bitmap files.

```
Option Explicit

Const NUMFRAMES = 15
Dim picArray(1 To NUMFRAMES) As Picture

Private Sub Form_Load()
    Dim strFile As String
    Dim intI As Integer
    For intI = 1 To NUMFRAMES
        strFile = App.Path & "\GLOBE" & _
            Format(intI) & ".BMP"
        Set picArray(intI) = LoadPicture(strFile)
    Next intI
End Sub

Private Sub tmrAnimate_Timer()
    Static intN As Integer
    intN = (intN Mod NUMFRAMES) + 1
    picTest.Picture = picArray(intN)
End Sub
```

I modified the AniGlobe application (discussed in Chapter 29, "Graphics") to create 15 globe images, and named them GLOBE1.BMP, GLOBE2.BMP, and so on, through GLOBE15.BMP. These 15 images are loaded into an array of 15 Picture objects, to be copied and displayed sequentially in a single PictureBox control. The *picTest* PictureBox control's AutoSize property should be set to *True* so that the frames of the animation will automatically fit. Figure 14-13 shows the animation frozen at about the middle of the sequence of frames.

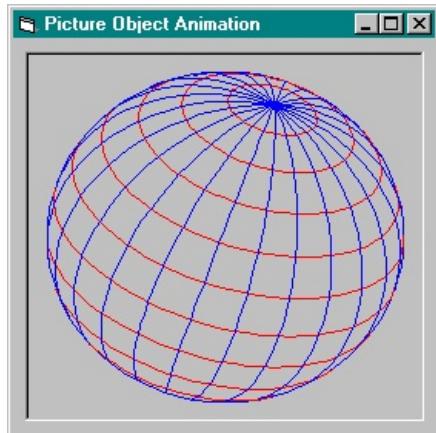


Figure 14-13. Simple animation sequence created using an array of picture objects.

Dear John, How Do I... Use the Animation Control?

Windows 95 introduced those tiny, animated Recycle Bins, searching flashlights, file copying, and similar system animations. The Animation control in Visual Basic makes it easy to add any of these animations to your own applications. The following code loads and runs two animations—one that sweeps the flashlight back and forth across a folder and one that zaps the contents of the Recycle Bin:

```
Option Explicit

Private Sub Form_Load()
    anmOne.Open App.Path & "\SEARCH.AVI"
    anmOne.Play
    anmTwo.Open App.Path & "\FILENUKE.AVI"
    anmTwo.Play
End Sub
```

To try out this code, add two Animation controls to a new form and name them *anmOne* and *anmTwo*. The Animation control, which is part of the Microsoft Windows Common Controls-2 6 (MSCOMCT2.OCX), enables you to control animation programmatically. Here I've used the Open method to load the animation clips, and I've used the Play method with none of its optional parameters, which results in the animations playing continuously until the form closes. Figure 14-14 shows these two animations in action.

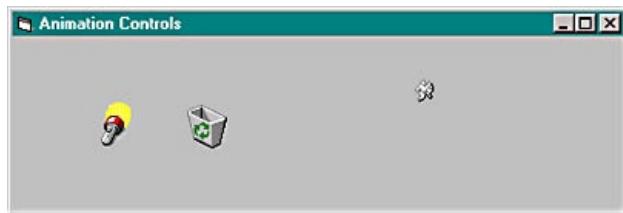


Figure 14-14. Simple system animations played using the Animation control.

Note that these animations run in their own threads, which means that your program can proceed with other processing.

Not all Audio Video Interleaved (AVI) files can be played using the Animation control. The animations must be silent clips; you'll get an error if you try to play an AVI file containing sound. Check the Visual Basic online help for more information about the fine details and options for the Animation control.

Dear John, How Do I... Position Text at an Exact Location in a Picture Box?

One advantage a picture box has over a text box or a label is that a picture box lets you print text at any location, with various fonts, in a variety of colors, and intermixed with graphics. You can change the font characteristics using the standard properties of the picture box, and you can position the text for printing at any location in the picture box by using a combination of the ScaleWidth and ScaleHeight properties and the TextWidth and TextHeight methods of the picture box.

The following code demonstrates how to center a string at a point and how to position a string so that it prints flush with the lower-right corner of the picture box. To try this code, add a PictureBox control named *picTest* to a new form, add the following code, run the application, and resize the form while it's running.

```
Option Explicit
```

```
Private Sub Form_Resize()
    Dim intX As Integer
    Dim intY As Integer
    Dim strA As String
    'Reposition the picture box
    picTest.Move 0, 0, ScaleWidth, ScaleHeight
    'Erase previous contents of picture box
    picTest.Cls
    'Determine center of picture box
    intX = picTest.ScaleWidth \ 2
    intY = picTest.ScaleHeight \ 2
    'Draw circle at center for reference
    picTest.Circle (intX, intY), 700
    'Print string centered in picture box
    strA = "CENTER"
    picTest.CurrentX = intX - picTest.TextWidth(strA) \ 2
    picTest.CurrentY = intY - picTest.TextHeight(strA) \ 2
    picTest.Print strA
    'Determine lower-right corner of picture box
    intX = picTest.ScaleWidth
    intY = picTest.ScaleHeight
    'Print string at lower-right corner
    strA = "Lower-right cornerDear John, How Do I... "
    picTest.CurrentX = intX - picTest.TextWidth(strA)
    picTest.CurrentY = intY - picTest.TextHeight(strA)
    picTest.Print strA
End Sub
```

Notice that the TextWidth method returns the effective length of the entire string, taking into account the current font settings, the number of characters in the string, and the proportional spacing for those characters. For this reason, always pass the exact string to be printed to this method just before printing and after any font properties have been set.

Figure 14-15 shows the result of this code.

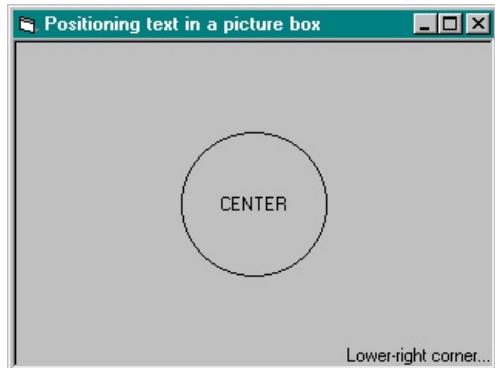


Figure 14-15. Printing text at exact locations in a picture box.

The above example shows the general technique for exact text placement. By extrapolating from this

example, you can print superscript characters, label graph axes, and perform similar tasks.

Dear John, How Do I... Scale a Font Infinitely?

In earlier versions of Visual Basic, we were limited to a certain fixed set of font sizes, even for the "infinitely scalable" TrueType fonts. Now it's easy to set the Size property of the Font object of graphical output devices to any font size for TrueType and PostScript fonts. To do so, add the following code on a blank form and set the form's Font property to any TrueType font:

```
Option Explicit

Private Sub Form_Click()
    Dim sngSize As Single
    sngSize = 1
    Do
        sngSize = sngSize * 1.2
        Me.Font = "Garamond"
        Me.Font.Size = sngSize
        Me.Print "Garamond - "; sngSize
    Loop Until sngSize > 100!
End Sub
```

Figure 14-16 shows the result of running this code.

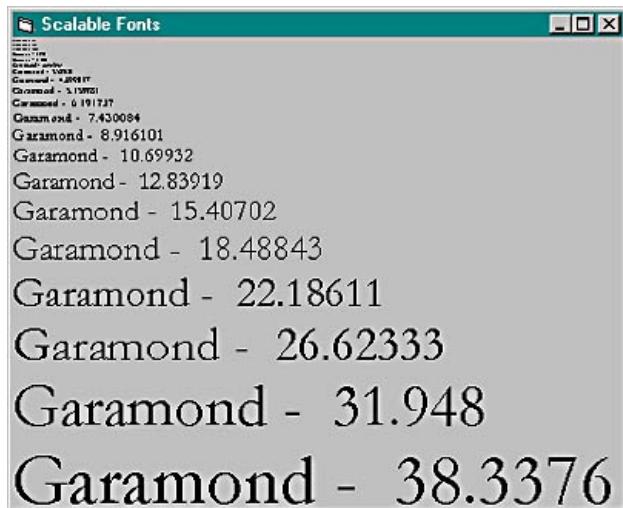


Figure 14-16. A range of font sizes displayed for the Garamond TrueType font when the form is clicked.

Dear John, How Do I... Rotate Text to Any Angle?

Visual Basic doesn't let you directly rotate a font to any angle, but with the help of a few Windows API function calls we can still get the job done. The following Rotator class defines an object that encapsulates five API functions, a LOGFONT data type, and a couple of required constants that, working together, let you easily rotate fonts on your screen or in your printed output:

```
'ROTATOR.CLS
Option Explicit

`API constants
Private Const LF_FACESIZE = 32
Private Const LOGPIXELSY = 90

Private Type LOGFONT
    lfHeight As Long
    lfWidth As Long
    lfEscapement As Long
    lfOrientation As Long
    lfWeight As Long
    lfItalic As Byte
    lfUnderline As Byte
    lsngStrikeOut As Byte
    lfCharSet As Byte
    lfOutPrecision As Byte
    lfClipPrecision As Byte
    lfQuality As Byte
    lsngPitchAndFamily As Byte
    lfFaceName(LF_FACESIZE - 1) As Byte
End Type

Private Declare Function SelectObject _
Lib "gdi32" (
    ByVal hdc As Long,
    ByVal hObject As Long
) As Long

Private Declare Function DeleteObject _
Lib "gdi32" (
    ByVal hObject As Long
) As Long

Private Declare Function CreateFontIndirect _
Lib "gdi32" Alias "CreateFontIndirectA" (
    lpLogFont As LOGFONT
) As Long

Private Declare Function TextOut _
Lib "gdi32" Alias "TextOutA" (
    ByVal hdc As Long,
    ByVal x As Long,
    ByVal y As Long,
    ByVal lpString As String,
    ByVal nCount As Long
) As Long

Private Declare Function GetDeviceCaps _
Lib "gdi32" (
    ByVal hdc As Long,
    ByVal intIndex As Long
) As Long

`Module-level private variables
Private mobjDevice As Object
```

```

Private msngSX1 As Single
Private msngSY1 As Single
Private msngXRatio As Single
Private msngYRatio As Single
Private mlfFont As LOGFONT
Private mintAngle As Integer

`~~~Angle
Property Let Angle(intAngle As Integer)
    mintAngle = intAngle
End Property
Property Get Angle() As Integer
    Angle = mintAngle
End Property

`~~~Label
Public Sub Label(sText As String)
    Dim lngFont As Long
    Dim lngOldFont As Long
    Dim lngRes As Long
    Dim bytBuf() As Byte
    Dim intI As Integer
    Dim strFontName As String
    `Prepare font name, decoding from Unicode
    strFontName = mobjDevice.Font.Name
    bytBuf = StrConv(strFontName & Chr$(0), vbFromUnicode)
    For intI = 0 To UBound(bytBuf)
        mlfFont.lfFaceName(intI) = bytBuf(intI)
    Next intI
    `Convert known font size to required units
    mlfFont.lfHeight = mobjDevice.Font.Size *
        GetDeviceCaps(mobjDevice.hdc, LOGPIXELSY) \ 72
    `Set italic or not
    If mobjDevice.Font.Italic = True Then
        mlfFont.lfItalic = 1
    Else
        mlfFont.lfItalic = 0
    End If
    `Set underline or not
    If mobjDevice.Font.Underline = True Then
        mlfFont.lfUnderline = 1
    Else
        mlfFont.lfUnderline = 0
    End If
    `Set strikethrough or not
    If mobjDevice.Font.Strikethrough = True Then
        mlfFont.lsngStrikeOut = 1
    Else
        mlfFont.lsngStrikeOut = 0
    End If
    `Set bold or not (use font's weight)
    mlfFont.lfWeight = mobjDevice.Font.Weight
    `Set font rotation angle
    mlfFont.lfEscapement = CLng(mintAngle * 10#)
    mlfFont.lfOrientation = mlfFont.lfEscapement
    `Build temporary new font and output the string
    lngFont = CreateFontIndirect(mlfFont)
    lngOldFont = SelectObject(mobjDevice.hdc, lngFont)
    lngRes = TextOut(mobjDevice.hdc, XtoP(mobjDevice.CurrentX), _
        Ytop(mobjDevice.CurrentY), sText, Len(sText))
    lngFont = SelectObject(mobjDevice.hdc, lngOldFont)
    DeleteObject lngFont

```

```

End Sub

`~~~Device
Property Set Device(objDevice As Object)
    Dim sngSX2 As Single
    Dim sngSY2 As Single
    Dim sngPX2 As Single
    Dim sngPY2 As Single
    Dim intScaleMode As Integer
    Set mobjDevice = objDevice
    With mobjDevice
        `Grab current scaling parameters
        intScaleMode = .ScaleMode
        msngSX1 = .ScaleLeft
        msngSY1 = .ScaleTop
        sngSX2 = msngSX1 + .ScaleWidth
        sngSY2 = msngSY1 + .ScaleHeight
        `Temporarily set pixels mode
        .ScaleMode = vbPixels

        `Grab pixel scaling parameters
        sngPX2 = .ScaleWidth
        sngPY2 = .ScaleHeight
        `Reset user's original scale
        If intScaleMode = 0 Then
            mobjDevice.Scale (msngSX1, msngSY1)-(sngSX2, sngSY2)
        Else
            mobjDevice.ScaleMode = intScaleMode
        End If
        `Calculate scaling ratios just once
        msngXRatio = sngPX2 / (sngSX2 - msngSX1)
        msngYRatio = sngPY2 / (sngSY2 - msngSY1)
    End With
End Property

`Scales X value to pixel location
Private Function XtoP(sngX As Single) As Long
    XtoP = (sngX - msngSX1) * msngXRatio
End Function

`Scales Y value to pixel location
Private Function YtoP(sngY As Single) As Long
    YtoP = (sngY - msngSY1) * msngYRatio
End Function

```

Even though a lot of code and complicated details are encapsulated in the Rotator object, it's very easy to use this object in your own applications. The following code shows the steps required. An instance of the Rotator object is created, a PictureBox control named *picTest* is set as the output device for the object, the angle for the output is set in the Angle property, and a string is passed to the Label method. Notice that the font name and size and the printing location (CurrentX and CurrentY) are set in the picture box's various font properties; the Rotator object uses these current settings to complete the output. For illustration purposes, I've commented out the Bold, Italic, Underline, Strikethrough, and Weight property settings, but you should experiment with these properties to see how they affect the rotated text output.

```

Option Explicit

Dim rotTest As New Rotator
Private Sub picTest_Click()
    Dim intA As Integer
    `Prepare the font in the picture box
    picTest.Scale (-1, -1)-(1, 1)
    With picTest

```

```
.CurrentX = 0
.CurrentY = 0
With .Font
    .Name = "Courier New"
    .Size = 11
    .Bold = True
    .Italic = True
    .Strikethrough = True
    .Underline = True
    .Weight = 1000
End With
End With
`Connect Rotator object to the picture box
Set rotTest.Device = picTest
`Label strings at a variety of angles
For intA = 10 To 359 Step 15
    rotTest.Angle = intA
    rotTest.Label Space(4) & picTest.Font.Name & Str(intA)
Next intA
End Sub
```

Figure 14-17 shows the output of this code.

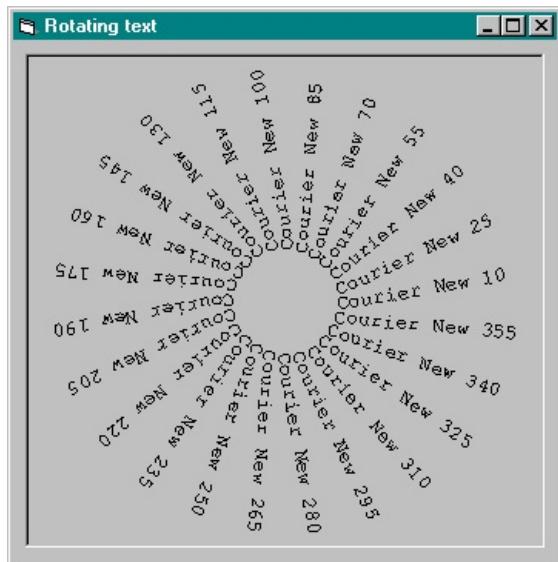


Figure 14-17. Text printed at a variety of angles using the Rotator object.

Dear John, How Do I... Use Multiple Fonts in a Picture Box?

The PictureBox control has a full set of properties for font characteristics. Unlike the TextBox and Label controls, the PictureBox control lets you change these properties on the fly, without affecting any text already drawn in the picture box. Simply reset the font properties, print the text, and then move on to print other text using any other combination of these properties. Figure 14-18 shows a few different fonts, all drawn in one picture box.

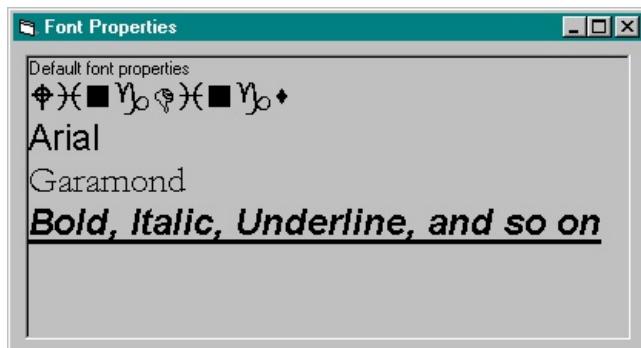


Figure 14-18. A creative combination of text styles in one picture box.

For backward compatibility, picture boxes support the font properties from earlier versions of Visual Basic, such as `FontName` and `FontSize`. The `Font` property, however, is an object in itself, and it now provides a set of its own properties. Look at the following sample code, which I used to create Figure 14-18, and notice how I set these properties by using the `Font` object. Keep in mind that you can create a `Font` object as an independent entity and assign the entire set of its properties to any object having a `Font` property in one command. For more details, see the Visual Basic online help for `Font` objects.

```
Option Explicit
```

```
Private Sub picTest_Click()
    picTest.Print "Default font properties"
    picTest.Font.Name = "WingDings"
    picTest.Font.Size = 18
    picTest.Print "WingDings"
    picTest.Font.Name = "Arial"
    picTest.Print "Arial"
    picTest.Font.Name = "Garamond"
    picTest.Print "Garamond"
    picTest.Font.Bold = True
    picTest.Font.Italic = True
    picTest.Font.Underline = True
    picTest.Font.Name = "Arial"
    picTest.Print "Bold, Italic, Underline, and so on"
End Sub
```

To try this code, start a new project and add a PictureBox control named `picTest` to the form. Add the code to the form, and at runtime click on the picture box to display the different formats of text in the picture box.

NOTE

For some purposes, Visual Basic's RichTextBox control also provides excellent multifont capabilities. Although the ability to position text at an exact location is not part of the RichTextBox control's properties and methods, you can left align, right align, and center text. You can also use multiple fonts within the same control. Select Microsoft Rich Textbox Control 6 in the Components dialog box to add this control to your Toolbox. Check it out.

SEE ALSO

- Chapter 21, "[Text Box and Rich Text Box Tricks](#)," for more information about the RichTextBox control

Chapter Fifteen

File I/O

Visual Basic provides several efficient techniques that you can use to read and write files. This chapter describes some of the most useful techniques for file I/O (input/output), along with several functions that people tend to overlook.

Dear John, How Do I... Rename, Copy, or Delete a File Efficiently?

Visual Basic has three statements that were specifically designed for renaming, copying, and deleting files. Use the Name statement to rename a file, the FileCopy statement to copy a file, and the Kill statement to delete a file. The following block of code demonstrates all three of these important file manipulation statements in action. When you click on the form, this program creates a simple text file named FILEA, renames it FILEB, copies FILEB to create FILEC.TXT in the directory in which the program resides, displays the contents of FILEC.TXT, and then deletes both FILEB and FILEC.TXT.

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim strA As String
    Dim intFileNum As Integer
    'Create a small file
    intFileNum = FreeFile
    Open "FILEA" For Output As #intFileNum
    Print #intFileNum, "This is a testDear John, How Do I... "
    Close #intFileNum
    'Rename file
    Name "FILEA" As "FILEB"
    'Copy file
    FileCopy "FILEB", App.Path & "\FILEC.TXT"
    'Read and display resulting file
    Open App.Path & "\FILEC.TXT" For Input As #intFileNum
    Line Input #intFileNum, strA
    Close #intFileNum
    MsgBox strA, vbOKOnly, "Contents of the Renamed and Copied File"
    'Delete files
    Kill "FILEB"
    Kill App.Path & "\FILEC.TXT"
End Sub
```

As shown in Figure 15-1, the contents of the copied file are displayed in a message box, verifying the correct operation of the demonstrated statements. When you click OK, FILEB and FILEC.TXT will be deleted.



Figure 15-1. Contents of a renamed and copied file.

Dear John, How Do I... Work with Directories and Paths?

Visual Basic has several statements that mimic their equivalent MS-DOS commands but don't require your program to shell out to MS-DOS or do any other fancy gymnastics. These statements are discussed in the sections that follow.

MkDir, ChDir, and RmDir

The MkDir statement creates a new directory, ChDir lets you change the current working directory, and RmDir deletes a directory. If you know how to use the commands of the same name at the MS-DOS prompt, you're all set to use them in your Visual Basic applications. Check the Visual Basic online help for more details about these statements.

CurDir and App.Path

Your application can determine the current working directory for any drive by using the CurDir function. Often, however, it's more useful to know the directory that the application itself resides in rather than the current directory. The Path property of the App object provides this information. For example, with the following lines of code, your application can determine both the current working directory and the directory that the application resides in. The application directory is usually a good location at which to read and write application-specific data files.

```
strCurDirectory = CurDir
strAppDirectory = App.Path
```

Dir

The Dir function is a powerful means of locating files or directories on any drive. The Visual Basic online help describes this function in detail. However, I want to point out that if you call the Dir function with parameters to specify the path or filename pattern and then call the Dir function again without parameters, the function will return the next file that matches the previously specified parameters from the directory currently specified to be searched. This makes possible the sequential collection of all the files in a directory. The following code demonstrates this important feature. This code lists all normal files (hidden files, system files, and directory files are not included) in all directories on the C drive and saves them in a text file named FILETREE.TXT. Add these two procedures to a new form, and run the program to create FILETREE.TXT in the current working directory. The program will automatically unload when it has finished.

```
Option Explicit
```

```
Sub RecurseTree(CurrentPath As String)
    Dim intN As Integer, intDirectory As Integer
    Dim strFileName As String, strDirectoryList() As String
    'First list all normal files in this directory
    strFileName = Dir(CurrentPath)
    Do While strFileName <> ""
        Print #1, CurrentPath & strFileName
        strFileName = Dir
    Loop
    'Next build temporary list of subdirectories
    strFileName = LCase(Dir(CurrentPath, vbDirectory))
    Do While strFileName <> ""
        'Ignore current directory, parent directory, and
        'Windows NT page file
        If strFileName <> ".." And strFileName <> ".." And _
            strFileName <> "pagefile.sys" Then
            'Ignore nondirectories

            If GetAttr(CurrentPath & strFileName) _
                And vbDirectory Then
                intDirectory = intDirectory + 1
                ReDim Preserve strDirectoryList(intDirectory)
                strDirectoryList(intDirectory) = CurrentPath
```

```

        & strFileName
    End If
End If
strFileName = Dir
`Process other events
DoEvents
Loop
`Recursively process each directory
For intN = 1 To intDirectory
    RecurseTree strDirectoryList(intN) & "\"
Next intN
End Sub

Private Sub Form_Load()
    Dim strStartPath As String
    Me.Show
    Print "WorkingDear John, How Do I... "
    Me.MousePointer = vbHourglass
    strStartPath = "C:\"
    Open "C:\FILETREE.TXT" For Output As #1
    RecurseTree strStartPath
    Close #1
    Me.MousePointer = vbDefault
    Unload Me
End Sub

```

This program also illustrates the use of recursive procedures in Visual Basic. In this example, the nesting depth of the recursive calls to `RecurseTree` is determined by the nested depth of your subdirectories.

You might wonder why `RecurseTree` builds a dynamically allocated array of subdirectory names instead of simply calling itself each time a subdirectory is encountered. It turns out that the `Dir` function call, when passed with no parameters, keeps track of the file and path it has most recently found in preparation for the next time `Dir` is called, regardless of where the function call originated. When it returns from a nested, recursive call, the `Dir` function will have lost track of its position in the current directory. The string array list of subdirectories, however, is updated accurately with each nested call because at each level a new, local copy of the string array variable is built. This is one of those annoying software details that will drive you up the wall until you realize what's happening!

After you have run this short program, look for a new file on your computer named `FILETREE.TXT` that contains a list of all files in all directories on your C drive. (Be patient—the program might take a while to finish.)

Figure 15-2 shows an example of `FILETREE.TXT` in WordPad.

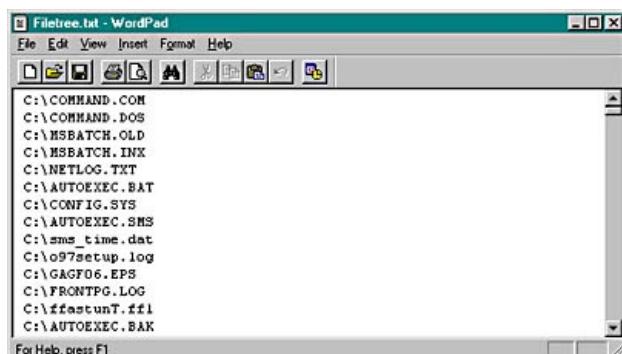


Figure 15-2. The `FILETREE.TXT` file in WordPad.

Dear John, How Do I... Perform Fast File I/O?

The best way to increase the speed of your file I/O operations is to use binary file access whenever possible. You can open any file for binary access, even if it's a text file.

Also, to increase file processing speed, remember to use the FileCopy command whenever it's appropriate, instead of reading and writing from one file to another.

SEE ALSO

- "[Dear John, How Do I... Rename, Copy, or Delete a File Efficiently?](#)" earlier in this chapter for information about working with files
- "[Dear John, How Do I... Work with Binary Files?](#)" for information about binary file access

Dear John, How Do I... Work with Binary Files?

Binary file operations in Visual Basic are fast. Whenever possible, I read and write files using the binary file Get and Put statements. Even if this requires extra processing of the data once the data is loaded into variables, the whole process usually remains faster than using the older Input, Output, and other such functions that have been around since the earliest versions of Basic. Today, with user-defined type (UDT) data structures and very large strings (such as those in 32-bit operating systems), binary file access makes more sense than ever. Let's take a look at a few examples.

UDT Data Structures

UDT data structures are read from and written to binary files in one fell swoop. For example, the following code writes the same personnel data record to a text file named EMPLOYEE.TXT 10 times when the command button *cmdPut* is clicked. When the command button *cmdGet* is clicked, the personnel data is read from the EMPLOYEE.TXT file.

```
Option Explicit

Private Type Employee
    FirstName As String * 20
    MiddleInitial As String * 1
    LastName As String * 20
    Age As Byte
    Retired As Boolean
    Street As String * 30
    City As String * 20
    State As String * 2
    Comments As String * 200
End Type

Private Sub cmdPut_Click()
    Dim Emp As Employee
    Dim intN
    Open App.Path & "\EMPLOYEE.TXT" For Binary As #1
    For intN = 1 To 10
        Emp.Age = 14
        Emp.City = "Redmond"
        Emp.Comments = "He is a smart guy."
        Emp.FirstName = "Willy"
        Emp.LastName = "Doors"
        Emp.MiddleInitial = "G"
        Emp.Retired = False
        Emp.State = "WA"
        Emp.Street = "One Macrohard Way"
        Put #1, , Emp
    Next intN
    Close #1
End Sub

Private Sub cmdGet_Click()
    Dim Emp(1 To 10) As Employee
    Dim intN As Integer
    Open App.Path & "\EMPLOYEE.TXT" For Binary As #1
    For intN = 1 To 10
        Get #1, , Emp(intN)
        Print Emp(intN).FirstName
    Next intN
    Close #1
End Sub
```

Figure 15-3 shows an example of the output after the Put Data and Get Data buttons are clicked.

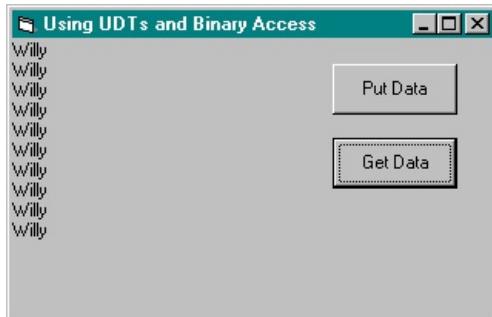


Figure 15-3. Using the binary access mode to write a UDT data structure to, and read a UDT data structure from, a text file.

Notice that the EMPLOYEE.TXT file is accessed 10 times in this sample program to read in 10 data records, but that this procedure actually reads 2960 bytes from the file. Each record in the Employee UDT data structure contains 296 bytes of data, so the entire structure is read from the file each time the Get statement is executed.

WARNING

As a general rule, you should always read binary file data using the same UDT data structure that was used to write the data. Starting with Visual Basic 4, data layout in memory and in binary files has become very tricky and unpredictable because of internal changes necessary for the global programming goals of Unicode and because of the need to make 32-bit applications run faster.

The 32-bit version of Visual Basic internally manipulates all string data, even within UDT data structures, as 2-bytes-per-character Unicode data. When performing file I/O, Visual Basic converts all Unicode strings to 1-byte-per-character ANSI strings, which completely changes the overall size of UDT data structures that contain strings.

UDT data structure items are double word-aligned to achieve the most efficient I/O throughput possible in 32-bit operating systems. This extra padding also changes the exact layout of the data within a UDT data structure.

Be careful!

Strings

A fast and flexible method of retrieving a chunk of data from a file is to open the data file in binary mode and then use the Get statement to place the entire chunk of data in a string variable. To retrieve a given number of bytes from a file, first set the size of the string. The Get statement will read bytes until the string is filled, so if the string is one character long, 1 byte will be read from the file. Similarly, if the string is 10,000 bytes long, 10,000 bytes from the file will be loaded. The following code demonstrates this technique using any generic text file named TEST.TXT:

```
Option Explicit

Private Sub Form_Click()
    Dim strA As String * 1
    Open App.Path & "\TEST.TXT" For Binary As #1
    Get #1, , strA
    Print strA
    Close #1
End Sub
Private Sub Text1_Click()
    Dim strB As String
    Open App.Path & "\TEST.TXT" For Binary As #1
    strB = Space(LOF(1))
    Get #1, , strB
End Sub
```

```
Text1.Text = strB
Close #1
End Sub
```

In the Form_Click event procedure, a fixed-length string of 1 byte is used to get 1 byte from the TEST.TXT file. In the Text1_Click event procedure, the string *strB* is filled with spaces to set the size of the variable to the size of the file. The Get statement is then used with the *strB* variable to input a chunk of data from the TEST.TXT data file that is equal in length to the string variable *strB*. This results in the variable *strB* being assigned the entire contents of the file.

NOTE

This isn't supposed to work! Microsoft warns us that Unicode character conversions "might" cause binary file data loaded into strings to become corrupted. However, testing the above technique with files containing all 256 possible byte values revealed nothing unexpected. Each byte of the file is assumed to be an ANSI character and when it is loaded into a string, each byte is padded with a binary byte value of 0 to make it a Unicode character. Everything works as it has in earlier versions of Visual Basic, but in future versions everything might not work that way. The solution is to start using binary arrays now for this type of work instead of strings. Read on!

Note that strings were limited to approximately 65,535 bytes in size in earlier, 16-bit versions of Windows and are now limited to approximately 2 billion bytes in Windows NT and Windows 95. This means, in most cases, that you can load the entire file into a single string using one Get statement. Just be sure to size the string to the file's size before calling Get by using the LOF function, as shown in the previous code.

SEE ALSO

- The Secret application in Chapter 34, "[Advanced Applications](#)," for a demonstration of binary file I/O

Byte Arrays

One way to process each byte of a file is to load the file into a string (as shown in the preceding section) and then use the ASC function to convert each character of the string to its ASCII numeric value. However, with the introduction of Visual Basic's Byte variable type, byte processing became much more efficient.

In the following code, I've created a dynamic Byte array that can be dimensioned to match a file's size. You can load the entire file into the array by passing the Byte array variable to the Get statement, as shown here:

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim intN As Integer
    Open App.Path & "\TEST.TXT" For Binary As #1
    ReDim bytBuf(1 To LOF(1)) As Byte
    Get #1, , bytBuf()
    For intN = LBound(bytBuf) To UBound(bytBuf)
        Print Chr(bytBuf(intN));
    Next intN
    Close #1
End Sub
```

The Connection Between Strings and Byte Arrays

The preceding example hints at the connection between strings and Byte arrays. But there's much more to this duality, so I'll describe some of the details you'll need to know to work effectively with the Byte array type.

Byte arrays will be used more and more often for the kinds of byte manipulations we have become used to accomplishing with strings. To ease this transition, you can plop a Byte array into the middle of many commands and functions that, in the past, would work only with string parameters. For example, as shown in the following code, you can assign a string to a Byte array and a Byte array to a string:

```
Option Explicit

Private Sub Form_Click()
    Dim intN As Integer
    Dim strA As String, strC As String
    Dim bytB() As Byte
    strA = "ABC"
    bytB() = strA
    `Displays 65 0 66 0 67 0
    For intN = LBound(bytB) To UBound(bytB)
        Print bytB(intN);
    Next intN
    Print
    strC = bytB()
    `Displays ABC
    Print strC
    `Notice actual size of string
    Print strA, Len(strA), LenB(strA)
End Sub
```

Figure 15-4 shows an example of the output when the form is clicked.

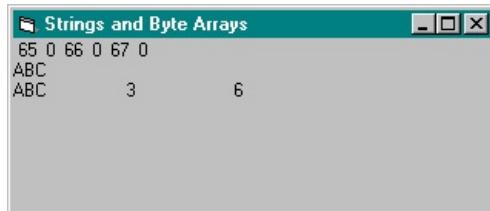


Figure 15-4. Working with strings and Byte arrays.

Notice that I've declared the Byte array *bytB()* as a dynamic array. In general, you should always declare your Byte arrays as dynamic when assigning strings to them. Visual Basic will then be able to automatically expand or collapse the size of the array to match the length of the string, just as it does for dynamic strings.

I suggest that you take the time to run the preceding code and study its output carefully. It will help you understand how string variables are maintained in memory as Unicode characters. The 3-character string ABC is shuffled from the string *strA* into the Byte array *bytB()*, and then all the bytes in *bytB()* are printed out. Instead of the three expected byte values of 65, 66, and 67 (ASCII representations of A, B, and C), we find that the Byte array actually contains six byte values. This is because the three-character string is maintained in memory in a 6-byte Unicode format and the contents of this string are shuffled straight across, with no conversion back to its 3-byte ANSI string representation upon assignment to the Byte array. Understanding this can help you clearly visualize how Unicode strings are manipulated in memory by Visual Basic.

The 6-byte array is then assigned to a second string, *strC*. Toward the end of the code listing, I added a Print command to display *strA*, its length in number of characters, and its length in actual byte count. *Len* and *LenB*, respectively, provide this information.

The StrConv Function

StrConv is a handy conversion function that lets you control the conversion of string data to and from

Unicode representation. The code below demonstrates the StrConv function as it forces a 6-byte Unicode string to transfer its characters to a 3-byte ANSI array and then shows how you can use StrConv to transfer a 3-byte ANSI array into a 6-byte Unicode string.

```
Option Explicit
```

```
Private Sub Form_Click()
    Dim intN As Integer
    Dim strA As String
    Dim bytB() As Byte
    strA = "ABC"
    bytB() = StrConv(strA, vbFromUnicode)
    `Displays 65 66 67
    For intN = LBound(bytB) To UBound(bytB)
        Print bytB(intN);
    Next intN
    Print
    strA = bytB()
    `This displays a question mark
    Print strA
    strA = StrConv(bytB(), vbUnicode)
    `Displays ABC 3 6
    Print strA, Len(strA), LenB(strA)
End Sub
```

Figure 15-5 shows an example of the output when the form is clicked.

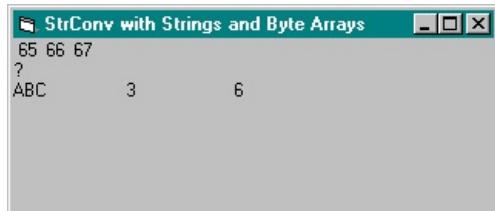


Figure 15-5. Using the StrConv function with strings and Byte arrays.

NOTE

What happens if you assign a Byte array to a string when the array bytes are not padded with extra 0 bytes, so as to be in proper internal Unicode format? The bytes transfer into the string, but Visual Basic doesn't know what the heck to do with those strange characters when you try to print the string. You'll get a question mark instead of whatever you might be expecting.

The constants *vbFromUnicode* and *vbUnicode* are just two of several constants defined for the StrConv function. Check the Visual Basic online help for more information about other handy uses for this versatile function.

By understanding the ways in which Byte arrays work hand in hand with strings and how you can control the ANSI/Unicode conversions by using the StrConv function, you will master the fast and efficient capabilities of binary file I/O.

Dear John, How Do I... Use the Visual Basic File System Objects?

The Scripting runtime library (SCRRUN.DLL) lets you create, copy, and delete folders and files through the FileSystemObject object. To use the FileSystemObject object in your application, follow these steps:

1. From the Project menu, choose References. Visual Basic displays the References dialog box.
2. Select the check box next to Microsoft Scripting Runtime, and click OK.
3. In your code, create an instance of the FileSystemObject object, then use the object's properties and methods to perform storage-related tasks.

The FileSystemObject object provides four subordinate objects that allow you to do the following tasks:

Uses for Objects Provided by FileSystemObject

Collection	Object	Task
Drives	Drive	Get information about disk drives on the system and access subordinate objects.
Folders	Folder	Create, copy, move, and delete folders; get Temporary, Windows, and System folders; navigate among folders.
Files	File	Create, copy, move, and delete files; get and change file attributes.
N/A	TextStream	Write text data to a file; read text data from a file.

The file system objects have a relatively flat organization. For instance, you can create a temporary text file with a few lines of code:

```
Sub WriteTempFile()
    Dim fsysTemp As New FileSystemObject
    Dim tstrTemp As TextStream
    Set tstrTemp = fsysTemp.CreateTextFile(fsysTemp.GetTempName)
    tstrTemp.Write "This is some temp text."
End Sub
```

One limitation of file system objects is that they do not include methods for reading and writing binary files. As its name implies, the TextStream object is designed for reading and writing text—either one line at a time or as an entire file.

The following sections use the File System Demo (FILESYS.VBP) sample application to show how to work with drives, folders, and files using the FileSystemObject object. Figure 15-6 shows the File System Demo in action.

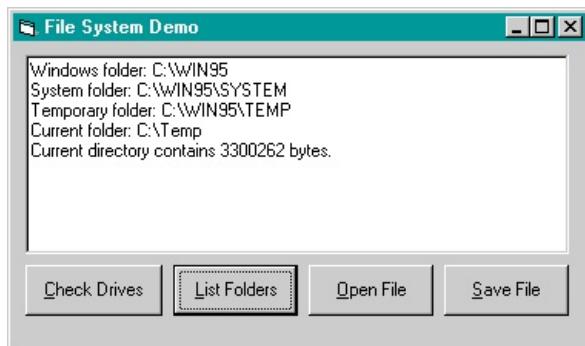


Figure 15-6. The File System Demo application shows how to work with drives, folders, and files through the FileSystemObject object.

Working with Drives

The FileSystemObject object's Drives method lets you check the status of each drive on the user's

system. Before querying a drive, however, you should check its `IsReady` property to make sure the drive is installed and contains a disk. The following code creates a new `FileSystemObject` object, then iterates through the `Drives` collection, displaying the free disk space on the drives that are ready.

```
'Create a new FileSystemObject object
Dim mfsysObject As New Scripting.FileSystemObject

Private Sub cmdCheckDrives_Click()
    'Declare a Drive object
    Dim drvItem As Drive
    'Add headers to text box
    txtData = "Drive" & vbTab & "Free space" & vbCrLf
    'Change mouse pointer. This can take a while
    MousePointer = vbHourglass
    'Check each drive
    For Each drvItem In mfsysObject.Drives
        'Update text box
        DoEvents
        'If drive is ready, you can get free space
        If drvItem.IsReady Then
            txtData = txtData & _
                drvItem.DriveLetter & vbTab & _
                drvItem.FreeSpace & vbCrLf
        'Otherwise, display "Drive not ready."
        Else
            txtData = txtData & _
                drvItem.DriveLetter & vbTab & _
                "Not Ready." & vbCrLf
        End If
    Next drvItem
    'Go back to the normal mouse pointer
    MousePointer = vbDefault
End Sub
```

The `MousePointer` and `DoEvents` statements above are a nod to the fact that checking disk drive status can take a little while—particularly with removable disks, CD-ROMs, and network drives. If you want to list only the fixed drives on the system, you can check the `DriveType` property, as shown here:

```
'Speed things up!
If drvItem.DriveType = Fixed Then
    'Check free space. . .
End If
```

Be careful not to combine testing the `DriveType` with the `IsReady` property in the same `If` statement, since both tests will be executed and the result won't be any faster than before.

Working with Folders

The `FileSystemObject` object is handy for getting the names of the Windows, System, and Temporary folders without making Windows API calls in your own code. The `FileSystemObject` object doesn't provide a method for getting the current directory, since that's already included in Visual Basic as the `CurDir` function. The following code displays information about some of the folders on the user's system:

```
Sub cmdListFolders_Click()
    Dim fldObject As Folder
    'Display folder information
    txtData = "Windows folder: " & _
        mfsysObject.GetSpecialFolder(WindowsFolder) & vbCrLf & _
        "System folder: " & _
        mfsysObject.GetSpecialFolder(SystemFolder) & vbCrLf & _
        "Temporary folder: " & _
        mfsysObject.GetSpecialFolder(TemporaryFolder) & vbCrLf & _
        "Current folder: " & CurDir & vbCrLf
```

```

`Get the current folder object
Set fldObject = mfsysObject.GetFolder(CurDir)
`Display some information about it
txtData = txtData & "Current directory contains " &
    fldObject.Size & " bytes."
End Sub

```

Notice that the GetFolder method returns an object reference. Once you've got a reference to a Folder object, you can copy, move, or delete the folder using the Folder object's methods. For example, the following lines of code move the current directory to the root and then back where it belongs:

```

`Moves the current directory up to the root, then back
Sub MoveFolder()
    Dim fldCurrent As Folder
    `Get the current folder
    Set fldCurrent = mfsysObject.GetFolder(CurDir)
    `Move the folder to the root
    fldCurrent.Move "\"
    `Move the folder back where it belongs
    fldCurrent.Move CurDir
End Sub

```

You can step through the preceding code to verify that the folder moves to the root directory and back. The Move, Copy, and Delete methods are so fast and powerful that they are a little scary to work with!

Working with Files

The key to using the FileSystemObject object to create and modify files is understanding the difference between the File and TextStream objects. You use the File object to get and modify the location and attributes of the file. You use the TextStream object to read and write text from the file.

The following code uses the Common Dialog control to get the name of a file to open, then displays the text from that file in a text box:

```

Sub cmdOpenFile_Click()
    `Declare a text stream object
    Dim tstrOpen As TextStream
    Dim strFileName As String
    `Display the Save common dialog
    dlgFile.ShowOpen
    strFileName = dlgFile.FileName
    `Check if a filename was specified
    If strFileName = "" Then Exit Sub
    `Check if the file already exists
    If Not mfsysObject.FileExists(strFileName) Then
        Dim intCreate As Integer
        intCreate = MsgBox("File not found. Create it?", vbYesNo)
        If intCreate = vbNo Then
            Exit Sub
        End If
    End If
    `Open a text stream
    Set tstrOpen = mfsysObject.OpenTextFile(strFileName, _
        ForReading, True)
    `Check if file is zero length
    If tstrOpen.AtEndOfStream Then
        `Clear text box, but don't read -- file is
        `zero length
        txtData = ""
    Else
        `Display the text stream
        txtData = tstrOpen.ReadAll
    End If
    `Close the stream

```

```
tstrOpen.Close
End Sub
```

One interesting aspect of the preceding code is the use of the FileExists method. Many of us are used to testing whether a file exists by calling the Dir function with the file's name. The FileExists method does essentially the same thing, but is a whole lot clearer when reviewing the code.

Another interesting detail is the use of AtEndOfStream after the file has been opened to check if the file contains any data to read. It is important to perform this test before reading from the file so that you avoid errors trying to read an empty or newly created file.

The code to save the file is very similar, as shown here:

```
Sub cmdSaveFile_Click()
    'Declare a text stream object
    Dim tstrSave As TextStream
    Dim strFileName As String
    'Display the Save common dialog
    dlgFile.ShowSave
    strFileName = dlgFile.FileName
    'Check if a filename was specified
    If strFileName = "" Then Exit Sub
    'Check if the file already exists
    If mfsysObject.FileExists(strFileName) Then
        Dim intOverwrite As Integer
        'Prompt before overwriting an existing file
        intOverwrite = MsgBox("File already exists. " & _
            "Overwrite it?", vbYesNo)
        'If the user chose No, exit this procedure
        If intOverwrite = vbNo Then
            Exit Sub
        End If
    End If
    'Open a text stream
    Set tstrSave = mfsysObject.OpenTextFile(strFileName, _
        ForWriting, True)
    'Save the text stream
    tstrSave.Write txtData
    'Close the stream
    tstrSave.Close
End Sub
```

When working with the TextStream and File objects, you often need to keep the filename around. The TextStream object doesn't provide a way to get at the file's name, and the easiest way to get a reference to a File object is through the GetFile method, which requires a filename argument. The following code shows how you must preserve a temporary filename to be able to delete the file when you're done:

```
'Creates sample temp file
Sub WriteTempFile()
    Dim fsysTemp As New FileSystemObject
    Dim tstrTemp As TextStream
    Dim strTempName As String
    'Get a temporary filename
    strTempName = fsysTemp.GetTempName
    'Create a text stream
    Set tstrTemp = fsysTemp.CreateTextFile(strTempName)
    'Write to the stream
    tstrTemp.Write "This is some temp text."
    'Close it
    tstrTemp.Close
    'Delete the file
    fsysTemp.GetFile(strTempName).Delete
End Sub
```

Chapter Sixteen

The Registry

In 32-bit Windows 95 and Windows NT, the *Registry* replaces the initialization (INI) files used by most 16-bit Windows-based applications to keep track of information between sessions. The Registry is a system-wide database designed to let all 32-bit applications keep track of application-specific data between runs of the application. The transition from using INI files to using the Registry is easy because Visual Basic provides a handful of functions that simplify the reading and writing of the Registry. Typical information stored in the Registry includes user preferences for colors, fonts, window positions, and the like. This chapter explains and demonstrates these Registry functions.

Dear John, How Do I... Read and Write to the Registry?

Visual Basic provides four functions that are used for reading and writing to the Registry: GetSetting, SaveSetting, DeleteSetting, and GetAllSettings. The first three functions are, in my opinion, the most important.

As indicated by their names, these functions get or save (read or write) string data (settings) from or to the Registry and delete any settings no longer used. GetAllSettings is a special function that lets you get all settings from a section in one call. This is an extension of the GetSetting function, so I'll leave it up to you to read the Visual Basic online help for more information about this particular function.

To demonstrate the first three functions, I've created a Registry class and a small test application that lets you get and save settings in the Registry. The object defined by the Registry class has properties corresponding to the parameters for the Registry functions as well as methods to get, save, and delete Registry entries based on the property values. The four string properties are Appname, Section, Key, and Setting. The code below defines all of the Registry object's properties and methods.

```
'REGISTRY.CLS
Option Explicit

Private mstrAppname As String
Private mstrSection As String
Private mstrKey As String
Private mstrSetting As String

'~~~AppName
Public Property Get Appname() As String
    Appname = mstrAppname
End Property
Public Property Let Appname(ByVal strAppName As String)
    mstrAppname = strAppName
End Property

'~~~Section
Public Property Get Section() As String
    Section = mstrSection
End Property
Public Property Let Section(ByVal strSection As String)
    mstrSection = strSection
End Property

'~~~Key
Public Property Get Key() As String
    Key = mstrKey
End Property
Public Property Let Key(ByVal strKey As String)
    mstrKey = strKey
End Property

'~~~Setting
Public Property Get Setting() As String
    Setting = mstrSetting
End Property
Public Property Let Setting(ByVal strSetting As String)
    mstrSetting = strSetting
End Property

'~~~RegGet
Public Sub RegGet()
    mstrSetting = GetSetting(mstrAppname, mstrSection, mstrKey)
End Sub

'~~~RegSave
Public Sub RegSave()
```

```

    SaveSetting mstrAppname, mstrSection, mstrKey, mstrSetting
End Sub

`~~~RegDel
Public Sub RegDel()
    DeleteSetting mstrAppname, mstrSection, mstrKey
End Sub

```

To try out an instance of this new Registry object, start a Standard EXE application and add a class module to it. Change the class Name property to *Registry*, add the previous code to it, and save the file as REGISTRY.CLS. Modify the application's startup form as follows: Change the form's Name property to *Registry_Form1*, and save this form using the same name. (I saved the project itself using the name REGISTRY_TEST.VBP.) Add two text boxes, named *txtKey* and *txtSetting*, and three command buttons, named *cmdSave*, *cmdGet*, and *cmdDel*. If you want, add labels above the TextBox controls and edit the various Text and Caption properties, as shown in Figure 16-1. Add this code to the form:

```

Option Explicit

Dim mregTest As New Registry

Private Sub cmdSave_Click()
    mregTest.Key = txtKey.Text
    mregTest.Setting = txtSetting.Text
    mregTest.RegSave
End Sub

Private Sub cmdGet_Click()
    mregTest.Key = txtKey.Text
    mregTest.RegGet
    txtSetting.Text = mregTest.Setting
End Sub

Private Sub cmdDel_Click()
    mregTest.Key = txtKey.Text
    mregTest.RegDel
End Sub

Private Sub Form_Load()
    mregTest.Appname = App.Title
    mregTest.Section = "Testing"
End Sub

```

Figure 16-1 shows the form during development.

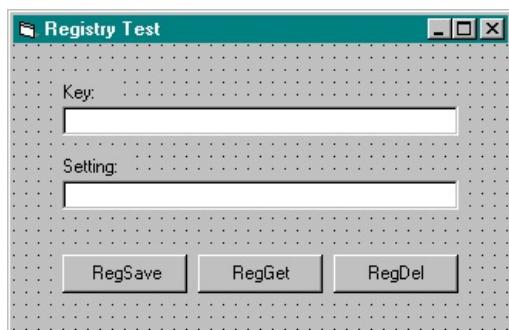


Figure 16-1. The Registry Test form at design time.

Notice in the Form_Load event procedure that I've assigned the Title property of the test program's App object to the Appname property of the *regTest* object, and I've hardwired a Section string, *Testing*. You can, of course, change these properties as desired.

At runtime, type a string in the Key text box. To save a setting for your key, type a string in the Setting text box and click the RegSave button. To access a previously stored setting, enter the identifying key string

and click the RegGet button. Click RegDel to delete the entry for the given key from the Registry.

Figure 16-2 shows the test application at runtime.

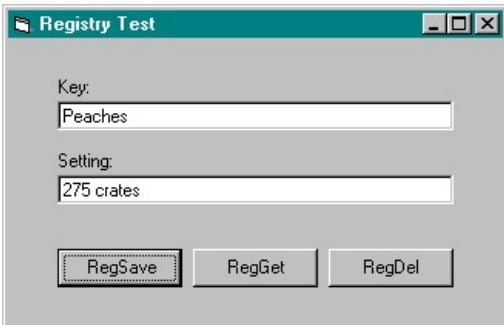


Figure 16-2. The Registry Test form at runtime.

A couple of notes about the Registry object's internal code are in order. The RegGet method calls the GetSetting function, which has an optional parameter named *default* that I chose not to use here. If you provide a string for this parameter, the same string is returned by the function if the indicated key is not found in the Registry. If you don't provide a string for this parameter, an empty string is returned if nothing is found, and this scenario works just fine for my object.

The best way to see exactly how the Registry is modified as you save and get settings is to run the Registry Editor application included in Windows 95 and Windows NT 4. Click the Start button on the Windows taskbar, choose Run, type *Regedit*, and click OK. As shown in Figure 16-3, these Visual Basic functions always save and get settings from the HKEY_CURRENT_USER\Software\VB and VBA Program Settings section of the Registry, and this is where you should always look for them. Notice that the Appname and Section properties further define the location of saved settings. In this example, the Appname property is *Registry_Test*, and the Section property is *Testing*. These show up as part of the tree structure on the left of the Registry Editor's display. On the right side of the display, you'll find the specific Key and Setting properties. In this example, I entered settings for the three keys: Apples, Oranges, and Peaches. While you are experimenting with the Registry_Test sample application, be sure to refresh the Registry Editor's display after you save or delete keys, either by pressing F5 or by choosing Refresh from the View menu.

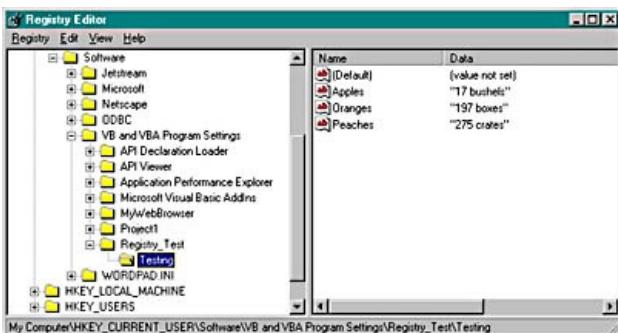


Figure 16-3. The contents of the Registry as displayed by the Registry Editor.

SEE ALSO

- The MySaver application in Chapter 29, "[Graphics](#)," and the VBClock application in Chapter 31, "[Date and Time](#)," for a demonstration of the Registry

Dear John, How Do I... Remember the State of an Application?

The answer to this question is just a continuation of the previous section. Use a Registry object's RegSave method to save the current state of your application, and use RegGet to recall the state. An excellent place to put this code is in the Form_Unload and Form_Load event procedures. With this arrangement, when the form is loaded but before it is actually drawn onto the screen, the previous size, shape, location, color, and any other detail of the state of the application can be restored. Similarly, immediately before the form is unloaded, the current state can be written for the next time the application is run.

SEE ALSO

- "[Dear John, How Do I... Read and Write to the Registry?](#)" earlier in this chapter for information about manipulating the Registry
- The MySaver application in Chapter 29, "[Graphics](#)," and the VBClock application in Chapter 31, "[Date and Time](#)," for a demonstration of the use of the Registry to save and restore an application's state

Dear John, How Do I... Associate a File Type with an Application?

The Registry stores information about how Windows responds to events on the desktop and in the Windows Explorer. For instance, if you double-click on a desktop file named MYFILE.TXT, Windows opens the file in the Notepad application. This association between files with the extension TXT and NOTEPAD.EXE is set in the Registry.

To examine the Registry entry for a file type, first locate the file type in the Registry Editor. You can find file types under the HKEY_CLASSES_ROOT key. Select the file type branch on the left side of the Registry Editor, and choose Export Registry File from the Registry menu. The Export Registry File dialog box is displayed, as shown in Figure 16-4. Enter a name for the file, verify that the Selected Branch option in the Export Range section is selected, and then click Save.

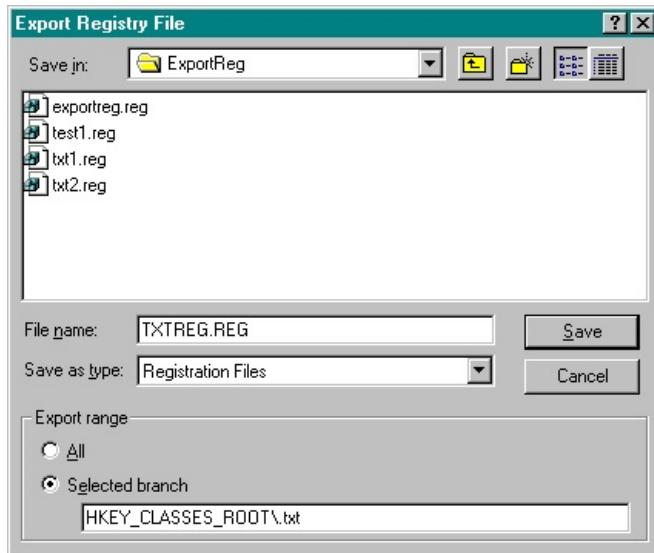


Figure 16-4. Exporting a branch in the Registry.

The Export Registry File command creates a text file with an REG extension. The following text shows the results of exporting the .txt and txtfile branches of the HKEY_CLASSES_ROOT key and merging them into one REG file. (Note that depending on how your system is configured, you might obtain different results.)

REGEDIT4

```
[HKEY_CLASSES_ROOT\.txt]
"Content Type"="text/plain"
@="txtfile"

[HKEY_CLASSES_ROOT\.txt\ShellNew]
"NullFile="""

[HKEY_CLASSES_ROOT\txtfile]
@="Text Document"
[HKEY_CLASSES_ROOT\txtfile\DefaultIcon]
@="c:\windows\SYSTEM\shell32.dll,-152"

[HKEY_CLASSES_ROOT\txtfile\shell]

[HKEY_CLASSES_ROOT\txtfile\shell\open]

[HKEY_CLASSES_ROOT\txtfile\shell\open\command]
@="c:\windows\NOTEPAD.EXE %1"
```

You can modify the exported Registry file to create your own file type associations. For example, the REG file below creates an association between the file type MYA and MYAPP.EXE.

REGEDIT4

```
[HKEY_CLASSES_ROOT\mya]
"Content Type"="VBRegSample"
@="MyApp"

[HKEY_CLASSES_ROOT\mya\ShellNew]
"NullFile"=""

[HKEY_CLASSES_ROOT\MyApp]
@="My File"

[HKEY_CLASSES_ROOT\MyApp\DefaultIcon]
@="C:\VB6 Workshop\Samples\Chapter 16\MyApp.EXE, 0"

[HKEY_CLASSES_ROOT\MyApp\shell]

[HKEY_CLASSES_ROOT\MyApp\shell\open]

[HKEY_CLASSES_ROOT\MyApp\shell\open\command]
@="C:\VB6 Workshop\Samples\Chapter 16\MyApp.EXE %1"
```

The items enclosed in square brackets are the Registry keys. The Registry keys establish the structure of the Registry data. The items to the right of the equal signs are the Registry data. An @ character in front of the equal sign indicates that the data is a default value. The first entry for the file type MYA tells Windows that the information for the application is found under the key [HKEY_CLASSES_ROOT\MyApp].

The following table describes some of the more important Registry keys.

Key	Description
\DefaultIcon	Indicates the file that contains the icon to display for this file type in the Windows shell. The second argument is the index of the icon in the file's resource list. Visual Basic applications seem to support displaying only the first icon (index 0).
\Shell\Open\Command	The command line executed when the user opens a file of this type from the Windows shell. The %1 argument is the filename passed to the application's command line interface.
\Shell\Print\Command	The command line executed when the user prints a file of this type from the Windows shell. The %1 argument is the filename passed to the application's command line interface.

To add these entries to the Registry, run Regedit with the registration filename as an argument. Add a /s switch to run Regedit silently so that the user doesn't see the process. This command line imports the MYAPP.REG file into the Registry:

```
RegEdit /s MyApp.Reg
```

NOTE

It is always a good idea to make a backup of the Registry before making modifications to it.

Figure 16-5 shows the .mya and MyApp branches in the Registry after MYAPP.REG has been imported.

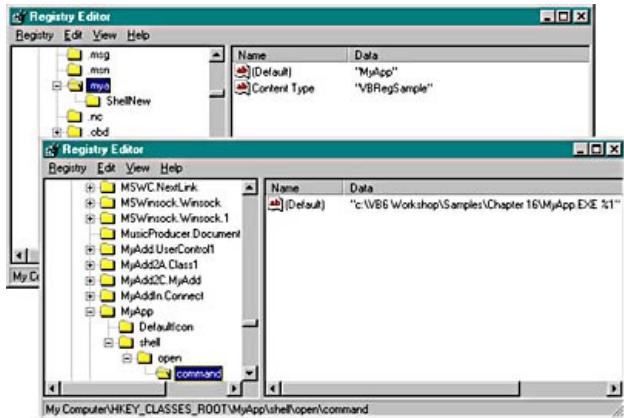


Figure 16-5. Portions of the Registry after MYAPP.REG has been imported.

Retrieving Command Line Arguments

Now that you've seen how your application's Open, Print, and other command data can be entered in the Registry, let's take a look at how your application can grab and use command line arguments. Applications that respond to Open, Print, and other commands from the Windows shell can accept command line arguments. Using the Visual Basic Command function, the code shown below shows how to retrieve the command line arguments.

```
Dim mstrFile As String

Private Sub Form_Load()
    'Get command line argument
    mstrFile = Command()
    'If there was an argument, do something
    If Len(mstrFile) Then
        OpenFile 'Or take other action
    End If
End Sub
```

See the Visual Basic online help for the Command function for an example of how to parse multiple arguments from the command line.

SEE ALSO

- *Inside the Microsoft Windows 98 Registry* by Günter Born (Microsoft Press, 1998) to learn more about the Registry

Chapter Seventeen

User Assistance

The tools that help users work with your applications form a smooth, unbroken chain - from the first icon seen, all the way through the last screen. ToolTips, status bars, Wizards, and Help of all kinds are the keys to making your applications seem natural and understandable. Some of these tools are built into Visual Basic, but others are not so obvious. For example, the Microsoft Help Workshop (HCW.EXE) is squirreled away in the \COMMONTOOLS directory on the Visual Studio CD-ROM, and the Microsoft HTML Help Workshop (HHW.EXE) is available from a self-extracting ZIP file named HTMLHELP.EXE in the \HTMLHELP directory on the Visual Studio CD-ROM.

In this chapter, I cover Visual Basic's built-in features and guide you through the process of building WinHelp and the new HTML-format Help. I haven't included full instructions for creating help files—that's beyond the scope of a single chapter. Besides, the Help Workshop and the HTML Help Workshop include some pretty good assistance of their own. I also explain how to integrate help into your Visual Basic application. These techniques have never been gathered in one place—until now that is!

Dear John, How Do I... Add ToolTips?

ToolTips are verbal descriptions of graphical interface elements such as toolbar buttons, command buttons, and status bar graphics. ToolTips are displayed when the user places the mouse pointer over an object and leaves it there for a period of time, as shown in Figure 17-1.

All the Visual Basic intrinsic controls containing visual elements have a `ToolTipText` property. Simply add a description of the control in the control's `ToolTipText` property, and the appropriate ToolTip is displayed when the user leaves the mouse pointer over the control for a short period of time.

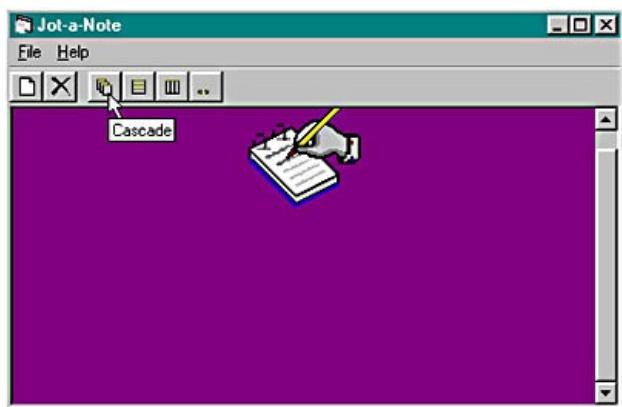


Figure 17-1. A *ToolTip* at runtime.

Most ActiveX controls provide a `ToolTipText` property, but ActiveX controls that contain subordinate elements (such as Toolbar controls and StatusBar controls) let you add ToolTips through the Property Pages dialog box. For example, to add ToolTips to buttons in a toolbar, follow these steps:

1. Draw a Toolbar control on a form.
2. In the Properties window, double-click the Custom property. Visual Basic displays the Toolbar control's Property Pages dialog box.
3. Click the Buttons tab of the dialog box.
4. Click Insert Button to add a button to the toolbar.
5. Type the ToolTip text for the button in the `ToolTipText` text box.
6. Click OK when you have finished adding buttons to the toolbar.

Adding ToolTips to panels in a status bar is done in a similar fashion.

SEE ALSO

- The Jot application in Chapter 32, "[Databases](#)," for a demonstration of ToolTips
- Chapter 13, "[The Visual Interface](#)," for instructions on creating Toolbar controls

Dear John, How Do I... Add a Status Display to My Application?

Use the StatusBar control, available in the Microsoft Windows Common Controls 6.0 (MSCOMMCTL.OCX), to add a status bar to your program. A StatusBar control creates a window, usually across the bottom of your form, containing up to 16 Panel objects. Panel objects have a number of properties that let you display text or predefined data, such as an automatically updated time and date. You can combine pictures with your text in each panel, too. You can set the number of panels and their properties on the various tabs in the Property Pages dialog box. To access this dialog box, right-click a positioned StatusBar control and choose Properties from the pop-up menu, or select the positioned StatusBar control, select Custom in the Properties window, and click the displayed ellipsis button.

Figure 17-2 shows a sample form with three StatusBar controls added, one with its alignment set to the bottom of the form and two set to align at the top of the form. The Visual Basic online help is a good reference for the many properties of the StatusBar control and its associated Panel objects.

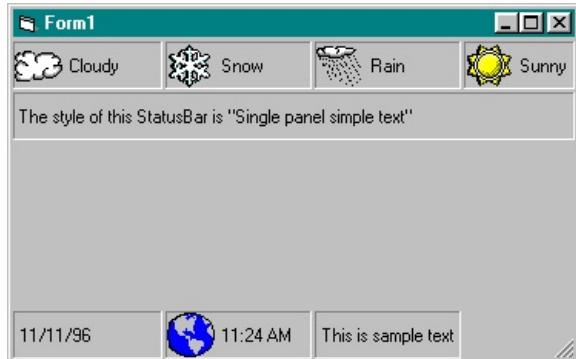


Figure 17-2. Three sample StatusBar controls in action.

SEE ALSO

- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for an example of the use of a status bar

Dear John, How Do I... Display a Tip of the Day at Startup?

The "Tip of the Day" feature in Microsoft Office applications helps new users learn about shortcuts and ways to accomplish work more quickly. Tips are handy for new users who might not think to look for easier ways to accomplish some tasks.

Visual Basic includes a template for creating Tip of the Day forms. Figure 17-3 shows the Tip of the Day sample application in action.

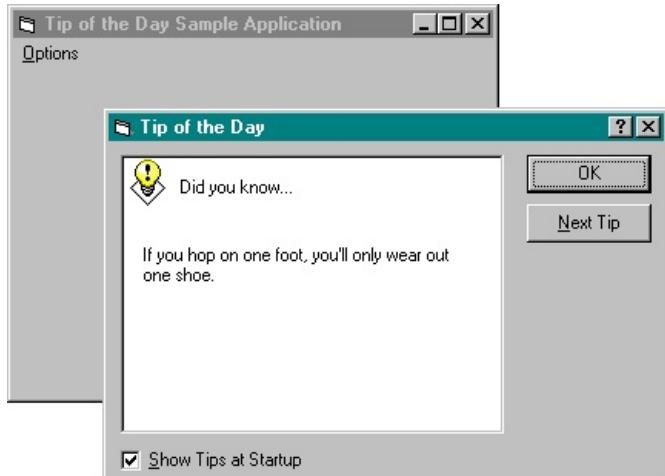


Figure 17-3. The Tip of the Day sample application (*TipOfDay.VBP*) demonstrates displaying a Tip of the Day.

To add a Tip of the Day form to your application, follow these steps:

1. From the Project menu, choose Add Form. Visual Basic displays the Add Form dialog box.
2. Select the Tip of the Day form template, and click Open. Visual Basic adds a form based on the frmTip.frm template to your project.
3. Add code to show the Tip of the Day form after the application has started.
4. Write a tip file named TIPOFDAY.TXT, and save it in the project directory.

The Tip of the Day form requires no additional code, other than the lines displaying the form after the application is started. Still, here are a few things you might consider:

- The Tip of Day form stores the setting of the Show Tips At Startup check box in the Registry in the entry for the application's executable file name (App.EXEName) so that the user can turn off the Tip of the Day. You should add an option within your application that lets users turn Tip of the Day back on if they change their minds.
- Since frmTip can unload itself, you'll need to check for errors when you use the Show method. Alternatively, you can check the Registry before attempting to load frmTip.
- If you don't want TIPOFDAY.TXT to change or be accidentally deleted, you can store the tips in a resource file that is compiled with the application.

The TipOfDay.VBP sample application included on the companion CD-ROM demonstrates each of these three points.

Dear John, How Do I... Walk Users Through Tasks Using Wizards?

Wizards lead users through complicated tasks step by step. It's often better to provide a wizard for a common, complex task than to write a bunch of how-to documentation. Any complicated, linear task is a candidate for a wizard, especially do-once-and-forget tasks such as configuring an application or setting up a database.

Visual Basic includes a Wizard Manager Add-In that simplifies creating wizards. The Wizard Manager generates a template for creating a wizard. The template includes code in a standard module that lets you use the wizard as a Visual Basic add-in. You can delete those elements, or simply ignore them, and use the generated template for any type of wizard you want to create.

The VBCal Wizard (Figure 17-4) demonstrates how to use multiple forms to create a wizard application.

Wizards may be separate executables or may be part of a larger application. So, what makes something a wizard? Quite simply, a wizard is an application that has the features listed below.

An Introduction that describes the task to be completed The Introduction might also describe any prerequisites the user might need to complete the task.

Separate steps for each stage of the task Only one step is displayed at a time and users must be able to navigate between steps in both directions. Any entered data is validated before moving to the next step.

A Finished display at the wizard's conclusion This lets users know they have successfully completed the tasks.



Figure 17-4. The VBCal.Wizard (VBCal.VBP) displays Introduction, Step, and Finished screens.

SEE ALSO

- The VBCal application in Chapter 31, "[Date and Time](#)," for a full description of the VBCal application

Dear John, How Do I... Create a WinHelp File?

The Help Workshop is a huge improvement over the old command line Help compiler. For one thing, the Help Workshop handles larger source files without a problem—the old Help compiler ran out of memory when working with graphics files over a certain size. The Help Workshop also handles long filenames and has a graphical user interface, shown in Figure 17-5.

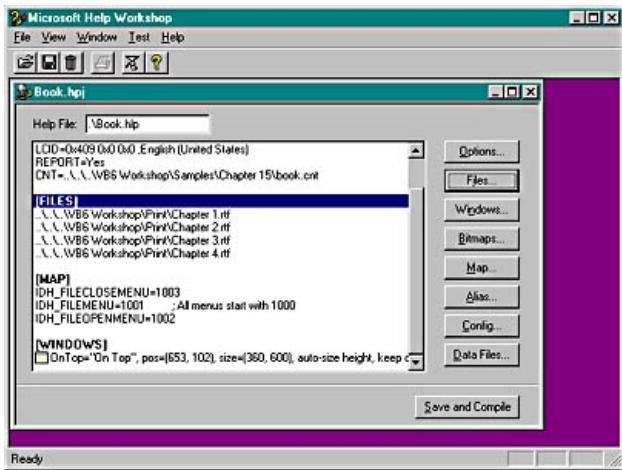


Figure 17-5. The Microsoft Help Workshop (HCW.EXE).

The Help Workshop creates a help project file (*.HPJ), which lists the source document files used to create the help file. The source documents must be saved in rich-text format (RTF) and can be created using Microsoft Word or any other word processing application that supports RTF files. The Help Workshop launches the compiler (HCRTF.EXE) and displays compiler errors, warnings, and messages once compilation is complete, as shown in Figure 17-6.

Help Topic Basics

Within a help source file, topics are separated by manual page breaks. You use footnotes to set a topic's title, search keywords, and the unique topic ID used by the Help system to link that topic to other topics. You create links to other topics using underlining and hidden text. Figure 17-7 shows a help topic with its various parts.

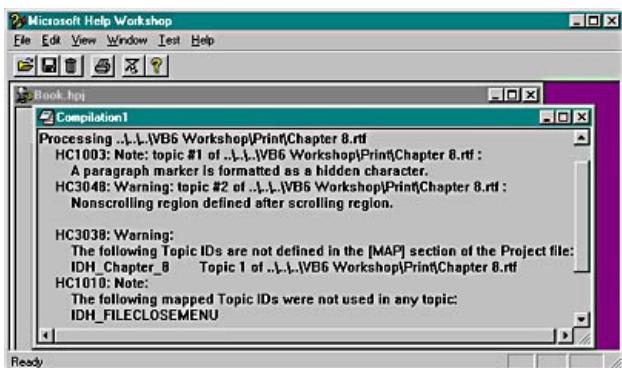


Figure 17-6. The Help Workshop displays errors, warnings, and messages after compilation.

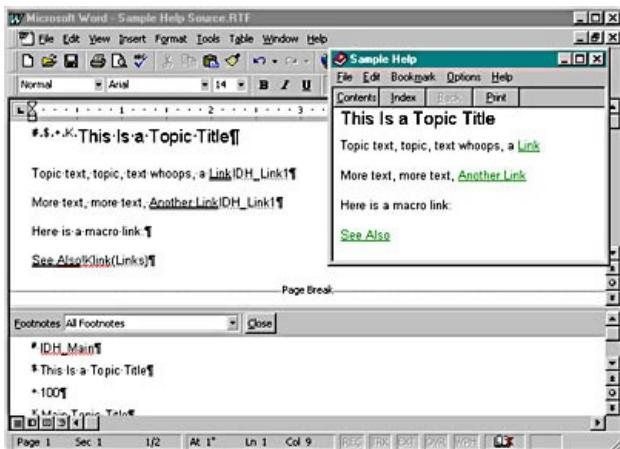


Figure 17-7. The anatomy of a help topic.

The footnotes and formatting that you can add to a help topic are listed in the table below.

Help Topic Footnotes and Formatting

Footnotes/Formatting	Meaning
\$	Topic title.
#	Topic ID; used to link to this topic from other topics.
K	Search keywords. These keywords will be listed in the help file's Index for this topic; multiple words are separated by semicolons. Can be used with KLink macro to create keyword links.
+	Browse sequence. The number entered for this footnote determines the order of the topics in the help file.
A	Associated keyword. Similar to a K footnote, but the footnote text doesn't appear in the Index. These footnotes are used with the ALink macro to create keyword links.
!	Macro. Help macros entered in the footnote text are run when the topic is displayed.
Double underline	Link to another topic. Links must be followed by hidden text listing the topic ID of the topic to jump to.
Single underline	Pop-up link; used for definitions and other short topics.
Hidden text	Topic ID of link. This ID must match the # footnote of a topic in the help project, or an error will occur during compilation.

Creating a Project File

A project file is a text file created using the Help Workshop that contains information about the files to be included in the help file and other settings. A project file has an HPJ extension. To create a new project file, choose New from the File menu to display the New dialog box. Select Help Project, and click OK. In the Project File Name dialog box, select a location, enter a name for the new project file with an HPJ extension, and then click the Save button. Figure 17-8 shows the project window that is displayed.

The Help Workshop project window contains various buttons that will display dialog boxes and allow you to configure your help file. If you click the Files button, you can add any topic files you created to the project file.



Figure 17-8. The Help Workshop project window.

Compiling and Testing a Help File

Compiling a help file is a process in which the topic files, graphics, and project file are used to create a help file. You compile a help file by clicking the Save And Compile button in the project window, which launches the compiler (HCRTF.EXE). After compilation, any errors or warnings will be displayed. To test the new help file, select Run WinHelp from the File menu and then click the View Help button in the View Help File dialog box.

Mapping Topic IDs

You enter topic IDs in your help file as strings. For example, you might type *IDH_FileMenu* in the # footnote for the help topic explaining the File menu. In your application, you enter the topic ID as a number. For example, you might type 1001 in the HelpContextID property for the File menu. The Help Workshop matches the name to the number in the [MAP] section of the project file, as shown here:

```
[MAP]
IDH_FILEMENU=1001           ; All menus start with 1000
IDH_FILEOPENMENU=1002
IDH_FILECLOSEMENU=1003
```

Keeping this [MAP] section in sync with your application is essential. Some tools, such as HelpBreeze, generate a BAS module with these values as constants. It is also fairly easy to do this with enumerated constants (Enums):

```
Enum HelpContext
    IDH_FILEMENU=1001           ^ All menus start with 1000
    IDH_FILEOPENMENU=1002
    IDH_FILECLOSEMENU=1003
End Enum
```

Using Full-Text Search and Table of Contents

Two new help features appeared with Windows 95 and Windows NT: full-text search and table of contents. The full-text search capability is included with the 32-bit Help Viewer (WINHLP32.EXE). Even help files compiled for Windows 3.1 support this feature when viewed using WINHLP32.EXE.

Windows automatically builds a database of words in the help file for use with Find using the Find Setup Wizard, shown in Figure 17-9.

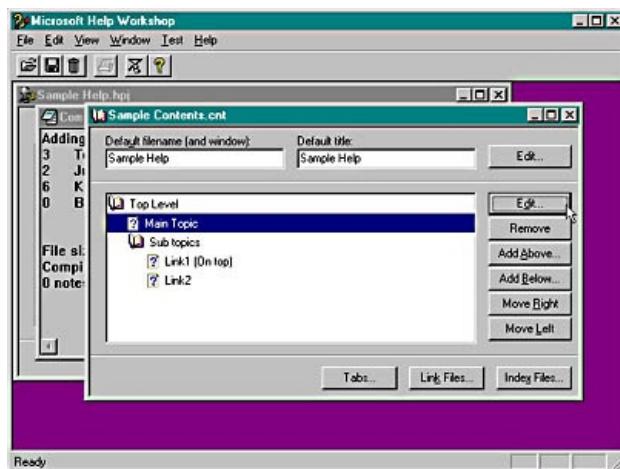
**Figure 17-9.** The Find Setup Wizard.

The Help Contents list is built from a text file listed in the Options section of the project file. The contents text file has a simple format for each line, as shown in the text below.

```
; From HCW.CNT
1 Getting Started      ; Top-level topic, not a link
2 What is Help Workshop?=BAS_THE_HELP_COMPILER ; Second-level topic
2 Help Workshop components=SDK_FILES_DOCS
2 Notational Conventions=SDK_NOTE_CONVENTION
1 What's New in Version 4.0
2 WinHelp 4.0
3 Redesigned User Interface ; Third-level topic
4 New Help Topics dialog box=NEW_WINHELP_INTERFACE
4 New context menu=NEW_WINHELP_CONTEXT_MENUS
4 New Options menu=NEW_WINHELP_OPTIONS_MENU
4 New annotation capabilities=NEW_ANNOTATION
4 Improved copying=NEW_WINHELP_COPY
4 New printing options=NEW_WINHELP_PRINTING
4 New ESC key function=NEW_KEY
4 Background color override=NEW_COLOR
```

You can maintain the contents file by opening the source file (CNT) in the Help Workshop. This lets you edit the file as it would appear in Help, as shown in Figure 17-10.

You use the File New command to create a new contents file from the Help Workshop.

**Figure 17-10.** A sample contents file.

Creating Help Windows

Your help file can display topics in specific Help windows defined in the Help Workshop. To define a Help window in the Help Workshop, click the Windows button in the Help project window and complete the dialog box shown in Figure 17-11.



Figure 17-11. The *Window Properties* dialog box is used to define *Help windows*.

To display a topic in the window defined in Figure 17-11, type >*OnTop* after the topic ID in a link's hidden text. Clicking that link will then display the topic in the *OnTop* window. You add window names to the entries in the contents file the same way; just type >*OnTop* after the topic ID of the topic. Adding a > footnote to a topic displays the topic in the custom window when the user selects the topic from the Help Index.

Using Macros

Macros perform special tasks in the Help system. You can run a macro when a topic displays, when the user clicks a button, or when the user clicks a link. To add a macro to a link, precede the macro name with a ! footnote in the hidden text of the link. For example, *See Also!*!KLink(*API*, *WinHelp*) runs the KLink macro to display a list of topics containing the keywords *API* or *WinHelp* when the user clicks See Also, as shown in Figure 17-12.

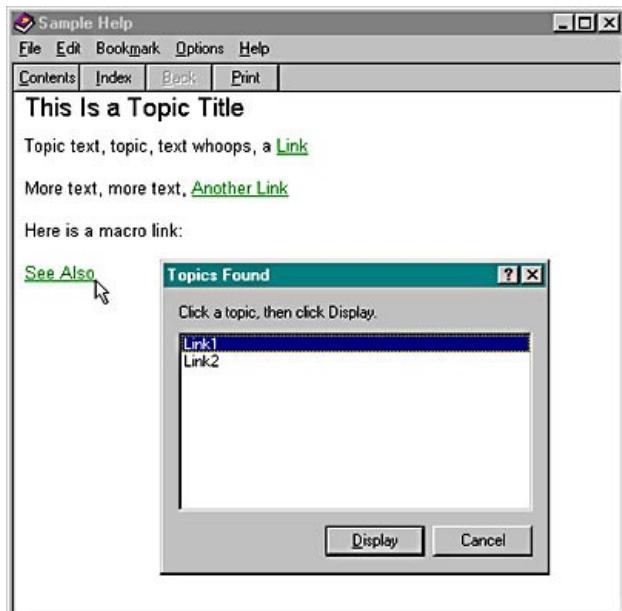


Figure 17-12. The *KLink* macro displays a list of topics that contain specific keywords in the *Topics Found* dialog box.

A list of the help macros you can use can be found in the Help Workshop Help file HCW.HLP.

SEE ALSO

- *Microsoft Windows 95 Help Authoring Kit* (Microsoft Press, 1995) for more information about Windows Help and using the Help Workshop



Dear John, How Do I... Use the WinHelp API Function to Add Help Files to My Projects?

You can activate a help file using a direct Windows API call, which is the tried-and-true, old-fashioned way to access help. The function is named WinHelp, and you have to add a declaration for it in your application, as shown in the following example. In this sample code, the Contents topic of the Windows Notepad main help file pops up when the form is clicked on.

```
Option Explicit

Private Declare Function WinHelp _
Lib "user32" Alias "WinHelpA" ( _
    ByVal hWnd As Long, _
    ByVal lpHelpFile As String, _
    ByVal wCommand As Long, _
    ByVal dwData As Long _
) As Long

Const HELP_TAB = &HF

Private Sub Form_Click()
    Dim lngX
    lngX = WinHelp(hWnd, "notepad.hlp", HELP_TAB, 0)
End Sub
```

The *hWnd* parameter is set to the *hWnd* property of your application's form. The *lpHelpFile* parameter is a string that contains the name of the help file to be activated. The *wCommand* parameter is one of the predefined constants for controlling this function. (Most of these constants are listed under the MSComDlg library in the Object Browser, after the CommonDialog control is added to the Toolbox; in the Visual Basic online help, under Help Constants; and in the table at the end of this section.) The *dwData* parameter can take on several types of values, depending on the value of *wCommand*. In particular, if *wCommand* is set to *cdlHelpContext*, *dwData*'s value determines which help topic is displayed.

To demonstrate how this works, the following WinHelp function activates the Windows Notepad help file and then displays a specific topic. (You will need to add the Microsoft Common Dialog control to your Toolbox.)

```
Private Sub Form_Click()
    Dim lngX
    lngX = WinHelp(hWnd, "notepad.hlp", cdlHelpContext, 1001)
End Sub
```

The header and footer Help topic will be displayed when the help file starts because I've set the *wCommand* parameter to the *cdlHelpContext* constant and the *dwData* parameter to the header and footer topic number. (Some help authoring tools, such as RoboHelp, automate the creation of a BAS file that contains constants for all topics—this is a great time-saver.)

There are several other variations on the function call that you might find useful. Each of the constants listed in the table below, when passed in the *wCommand* parameter, causes a specific action to be carried out by the WinHelp function. (Most of these constants are defined by Visual Basic for the CommonDialog control, hence the *cdl* prefixes.) For example, to display help information about how to use the Help system itself, use the constant *cdlHelpHelpOnHelp*.

Constant	Value	Description
<i>cdlHelpContext</i>	1	Displays help for a particular topic.
<i>cdlHelpQuit</i>	2	Closes the specified help file.
<i>cdlHelpIndex</i>	3	Displays the index of the specified help file.
<i>HELP_FINDER</i>	11	Displays the Help Topics dialog box with the last selected tab displayed. You must define this constant in your application because it is not included in the intrinsic constants.

<i>HELP_TAB</i>	15	Displays the Help Topics dialog box with the tab index specified by <i>dwData</i> selected. (The Contents tab is 0, the Index tab is _2, and the Find tab is _1.) You must define this constant in your application because it is not included in the intrinsic constants.
<i>cdlHelpContents</i>	3	Displays the Contents topic in the current help file. This constant is provided for compatibility with earlier versions of help; new applications should display the Help Topics dialog box by using <i>HELP_FINDER</i> or <i>HELP_TAB</i> .
<i>cdlHelpHelpOnHelp</i>	4	Displays help for using the Help application itself.
<i>cdlHelpSetIndex</i>	5	Sets the current index for multi-index help.
<i>cdlHelpSetContents</i>	5	Designates a specific topic as the Contents topic.
<i>cdlHelpContextPopup</i>	8	Displays a topic identified by a context number.
<i>cdlHelpForceFile</i>	9	Creates a help file that displays text in only one font.
<i>cdlHelpKey</i>	257	Displays help for a particular keyword.
<i>cdlHelpCommandHelp</i>	258	Displays help for a particular command.
<i>cdlHelpPartialKey</i>	261	Calls the search engine in Windows Help.

These constants are predefined for the CommonDialog control. You can copy them into your code by using the Object Browser, or you can enable the CommonDialog control in your application, in which case all *cdl* constants will be automatically available. To do so, choose Components from the Project menu and check the Microsoft Common Dialog Control 6.0 check box. The constants are also discussed in the description of the HelpCommand property in the Visual Basic online help.

SEE ALSO

- The Lottery application in Chapter 29, "[Graphics](#)," for a demonstration of the WinHelp function

Dear John, How Do I... Add Context-Sensitive F1 Help to My Projects?

Forms, menu items, and most controls have a HelpContextID property that provides a context-sensitive jump to a specific help topic when the F1 key is pressed. The HelpFile property of the App object sets the path and name of the help file for the entire application, and the control with the focus determines which help topic is activated when F1 is pressed. If a control's HelpContextID property is set to 0 (the default), the containing control or form is checked for a nonzero HelpContextID value. Finally, if the form's HelpContextID is set to 0, the help file's Contents topic is activated as the default.

This scheme works well for accessing context-sensitive help, which is activated when the F1 key is pressed, but how can we activate the help file programmatically without actually pressing the F1 key? Well, here's a slick trick that makes the HelpContextID property much more valuable: simply code your program to send an F1 keypress to your application using the SendKeys statement. The SendKeys statement tells Windows to send keypresses to the window that currently has the focus, so the program responds as though the F1 key had been pressed.

The following code fragments demonstrate this technique:

```
Private Sub Form_Load()
    App.HelpFile = "notepad.hlp"
End Sub

Private Sub cmdHelp_Click()
    cmdHelp.HelpContextID = 1001
    SendKeys "{F1}"
End Sub
```

The help file to be accessed is set in the Form_Load event procedure, but you can change the path and filename at any point in your program if you want to access multiple help files. In the cmdHelp_Click event procedure, you activate a specific topic in the help file by setting the HelpContextID property to the context number of the desired topic and then sending the F1 keypress using the SendKeys statement. These HelpFile and HelpContextID properties can be set at runtime, as shown above, or you can set them at design time.

Menu items also have a HelpContextID property, but the designated topic in the help file will be activated only if the menu item is highlighted and the F1 key is manually pressed. If you click on the menu item and set up a SendKeys command in the menu event procedure, as shown in the following code, the topic that will be activated will not be determined by the menu item's HelpContextID value but by the HelpContextID value for whatever control currently has the focus:

```
Private Sub mnuHelp_Click()
    SendKeys "{F1}"
End Sub
```

This happens because the menu vanishes as soon as it is clicked on and the focus is returned to the control before the SendKeys command has time to send the F1 keypress. Using a menu command to get context-sensitive help for a control works well, but the behavior seems a little strange until you figure out the sequence of events.

Dear John, How Do I... Use the CommonDialog Control to Add Help Files to My Projects?

The CommonDialog control provides a powerful and flexible way to access help files. Add a CommonDialog control to your form, set its relevant help file properties, and use the ShowHelp method to activate the Help system. It's that easy—you don't even need to use the control to display a dialog box. For example, the following code activates help for Windows Help itself when the *cmdHelpContents* command button is clicked:

```
Private Sub cmdHelpContents_Click()
    dlgCommon.HelpCommand = cdlHelpOnHelp
    dlgCommon.ShowHelp
End Sub
```

NOTE

Setting the HelpCommand property to *cdlHelpContents* displays the first topic in a help file if Windows can't locate the help contents file.

Here's an example that shows the activation of a specific help file topic when a menu item is clicked:

```
Private Sub mnuHelp_Click()
    dlgCommon.HelpFile = "notepad.hlp"
    dlgCommon.HelpCommand = cdlHelpContext
    dlgCommon.HelpContext = 1001
    dlgCommon.ShowHelp
End Sub
```

Dear John, How Do I... Add WhatsThisHelp to a Form?

Windows 95 provides a few twists to the way help files work. One of the nice features is called WhatsThisHelp. A form that has this feature displays a question-mark button on the title bar that, when clicked, changes the mouse pointer to a special question-mark symbol. Clicking a control on the form with this special pointer then activates a pop-up topic specific to the object clicked.

It's easy to add WhatsThisHelp to your Visual Basic form and its controls. Start by setting the form's WhatsThisButton and WhatsThisHelp properties to *True*. The form's BorderStyle property must also be changed from its default setting before you'll see the question-mark box on the title bar. Set BorderStyle to either 1 - *Fixed Single* or 3 - *Fixed Dialog*. Figure 17-13 shows a form with the question-mark title bar button.

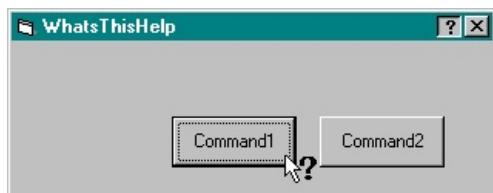


Figure 17-13. A form showing the question-mark button that activates WhatsThisHelp.

To determine exactly which help file will be activated, you must set the HelpFile property of the App object to the path and name of the help file. You can do this either at design time or at runtime. To set the help file at design time, choose your project's Properties from the Project menu, select the General tab of the Project Properties dialog box, and enter the name of the appropriate help file, including the full path to the file, in the Help File Name text box. To set the help file at runtime, assign the pathname and the filename to the App.HelpFile property.

Finally, to define the topic in the help file that will pop up when a given control is clicked, set that control's WhatsThisHelpID property to the topic's ID number.

WhatsThisMode

There's one other way you can use WhatsThisHelp. You can programmatically activate the special question-mark mouse pointer by using the WhatsThisMode method of the form. This is a useful technique when you want to add the WhatsThisHelp feature to a form that has its BorderStyle property set to 2 - *Sizable*. Figure 17-14 shows a form set up to activate WhatsThisHelp when the Get Help command button is clicked.

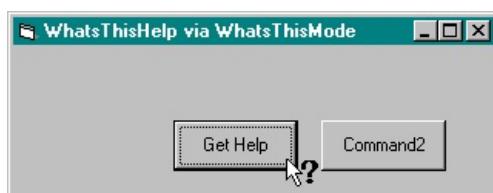


Figure 17-14. A form showing how WhatsThisHelp can be activated using the WhatsThisMode method.

When the Get Help button is clicked, the mouse pointer changes to the special question mark, and the WhatsThisHelp state operates in exactly the same way as when the title bar question-mark button was clicked in the previous example. To use this WhatsThisHelp technique, all I had to do was set the form's WhatsThisHelp property to *True* and add one line to the command button's Click event. Here's the code:

```
Private Sub cmdGetHelp_Click()
    WhatsThisMode
End Sub
```

Dear John, How Do I... Create HTML Help?

If you're deploying a new application, you should consider authoring the Help system in Microsoft's new HTML Help format. All of the Visual Studio help is written as HTML Help, as is all the help for the new Windows 98 operating system. Although WINHLP32.EXE will continue to work on Windows 98 and future Microsoft operating systems, it's likely that WINHLP32.EXE won't receive any additional enhancements. The focus of Microsoft's new help development is HTML Help. Figure 17-15 shows a sample of HTML Help as viewed in the HTML Help browser.

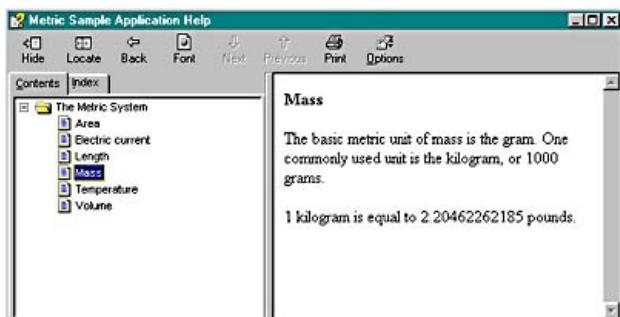


Figure 17-15. A sample HTML Help file viewed in the HTML Help browser.

You can create HTML Help using the Microsoft HTML Help Workshop, which is included in a self-extracting ZIP file named HTMLHELP.EXE in the \HTMLHELP directory of the Visual Studio CD-ROM.

The HTML Help Workshop includes the following components:

HTML Help Workshop (HHW.EXE) Lets you create and edit contents, index, and HTML files.

HTML Help Image Editor (FLASH.EXE) Captures screen shots and lets you crop, edit, and transform images.

HTML Help Authoring Guide (HTMLHELP.CHM) Provides full documentation for the HTML Help Workshop and associated components.

HTML Help Engine (HH.EXE) Displays compiled HTML Help files (.CHM). Compiled files are compressed, single-file versions of the HTML Help source files (.HTM).

HTML Help Converter DLLs Converts WinHelp project files to HTML Help projects.

HTML Help ActiveX Control (HHCTRL.OCX) Provides HTML Help table of contents, index, related topics, pop-up topics, window control, context-sensitivity, and other Help features within an HTML file.

HTML Help Java Applet Control (HHCTRL.CLASS) Provides HTML Help table of contents, index, and related topic capabilities to Java scripts within an HTML file.

HTML Help Authoring DLL Provides additional information about table of contents entries.

Converting WinHelp Projects to HTML Help

The HTML Help Workshop includes a wizard to help you convert existing WinHelp to HTML Help projects. To start the HTML Help Workshop New Project Wizard, choose Open from the File menu and specify a WinHelp project filename (.HPJ). Figure 17-16 shows the Introduction screen of the New Project Wizard.

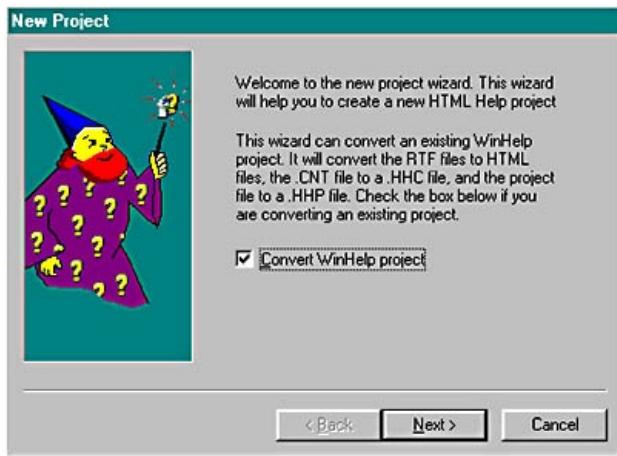


Figure 17-16. Simply open an .HPJ file in the HTML Help Workshop to convert a WinHelp project to HTML Help.

The New Project Wizard automatically converts the WinHelp project's files to HTML equivalents as shown in the following table:

HTML Conversions by New Project Wizard

File description	WinHelp file type	HTML Help file type
Project file	.HPJ	.HHP
Contents file	.CNT	.HHC
Index file	None (KLinks in source file)	.HHK
Source files	.RTF	.HTM (one for each topic)

The conversion process handles most standard WinHelp elements smoothly. However, not all WinHelp macros and segmented hypergraphics are converted in the source files. You will probably need to do other significant housekeeping before you can use your converted HTML Help file in an application. The HTML Help Authoring Guide (HTMLHELP.CHM) contains a list of conversion issues in the ReadMe topic.

Using the HTML Help Control

The HTML Help control (HHCTRL.OCX) lets you create pop-up menus and dialog boxes for See Also references within your Help topics, as shown in Figure 17-17.

The HTML code for the page shown in Figure 17-17 can be found in the file HTMLCTRL.HTM on the companion CD-ROM included with this book. The file is also shown below for your convenience:

```

<HTML>
<HEAD>
<TITLE>HTML Help Control (HHCtrl.OCX)</TITLE>
<STYLE>
BODY { background: white }
</STYLE>
</HEAD>

<BODY TOPMARGIN=12 LEFTMARGIN=10>
<P>Use the HTML Help control to create buttons that display
pop-up menus that display links to other topics, as shown
below:</P>

<P>
<OBJECT ID=hhctrl TYPE="application/x-oleobject"
        CLASSID="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11"
        CODEBASE="file:///htmlhelp/files/hhctrl.ocx#Version=4,0,0,21"
        WIDTH=32
        HEIGHT=32

```

```

HSPACE=4
>
<PARAM NAME="Command" VALUE="Related Topics, MENU">
<PARAM NAME="Button" VALUE="Bitmap:help.bmp">
<PARAM NAME="Item1" VALUE="Topic1;..\topics\topic1.htm">
<PARAM NAME="Item2" VALUE="Topic2;..\topics\topic2.htm">
<PARAM NAME="Item3" VALUE="Topic3;..\topics\topic3.htm">
<PARAM NAME="Item4" VALUE="Topic4;..\topics\topic4.htm">
</OBJECT>
</P>

<P>The HTML Help control can also display a dialog box containing
the See Also links, as shown here:</P>
</P>
<P>
<OBJECT ID=hhctrl TYPE="application/x-oleobject"
        CLASSID="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11"
        CODEBASE="file:///htmlhelp/files/hhctrl.ocx#Version=4,0,0,21"
        WIDTH=32
        HEIGHT=32
        ALIGN=BOTTOM
        HSPACE=4
        >
        <PARAM NAME="Command" VALUE="Related Topics">
        <PARAM NAME="Button" VALUE="Text:See AlsoDear John, How Do I... ">
        <PARAM NAME="Item1" VALUE="Topic1;..\topics\topic1.htm">
        <PARAM NAME="Item2" VALUE="Topic2;..\topics\topic2.htm">
        <PARAM NAME="Item3" VALUE="Topic3;..\topics\topic3.htm">
        <PARAM NAME="Item4" VALUE="Topic4;..\topics\topic4.htm">
</OBJECT>
</P>

</BODY>
</HTML>

```

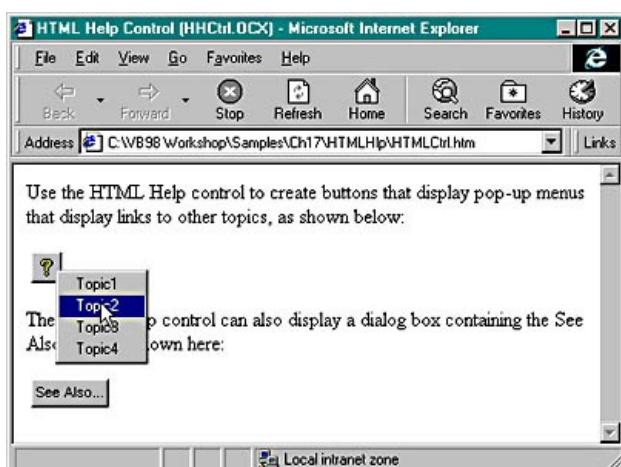


Figure 17-17. Use the HTML Help control on HTML or DHTML pages to display links to related topics.

The preceding HTML code uses the Command parameter to determine whether the control displays a pop-up menu (VALUE="Related Topics, MENU") or a dialog box (VALUE="Related Topics"). The value of the Button parameter determines whether to display a bitmap, an icon, or text on the button face of the control.

The HTML Help control is for use on HTML and DHTML pages only. You can't load the control in Visual Basic for use on forms or in other containers. Trying to do so results in an "Invalid control" error.

SEE ALSO

- The Microsoft Web site <http://www.microsoft.com/workshop/author/html/help> for updates to the Microsoft HTML Help Workshop and related articles
- *The Official Guide to Microsoft HTML Help* by WexTech Systems, Microsoft Press, 1998

Chapter Eighteen

Security

Legal issues surrounding software development and software ownership rights can get thorny. One simple device used to prove authorship of an application is a hidden credits dialog box, sometimes referred to as an *Easter egg*. Another handy technique is to embed encrypted messages that declare authorship. Both of these methods are described in this chapter.

Passwords are an integral part of many applications for which security is an issue. It's easy to create a password entry dialog box in Visual Basic; I'll show you how in this chapter.

Dear John, How Do I... Add a Hidden Credits Screen?

Sometimes they're called Easter eggs; sometimes they're not called anything at all. Many applications have a hidden, undocumented feature that lets the authors put on an impressive little show. Creatively designed Easter eggs, such as the example in Figure 18-1, can be a lot of fun.

Another purpose of these Easter eggs is to provide some legal protection for the author. If there's ever a disagreement as to who is the original creator of a piece of software, the real author can duly impress everyone by clicking here or there to open a dialog box that will prove authorship. Wouldn't that be fun to do in a court of law someday?

A straightforward approach is to create a special-purpose form to display whatever information you want on your hidden credits screen. The real trick is in deciding how such a form will be activated. I can offer general concepts and a specific example, but there's a lot of room for creativity in determining exactly how to activate your Easter egg.



Figure 18-1. A sample Easter egg (semisecret hidden credits screen).

It's easy to detect where the mouse is clicked on a form or control. The `MouseDown` and `MouseUp` events provide `x` and `y` parameters that tell you exactly where the mouse pointer is located when the mouse is clicked. Likewise, it's easy to determine the state of the shift keys (`Shift`, `Ctrl`, and `Alt`) and which mouse button or buttons are pressed. All of this information is passed as parameters to your `MouseUp` or `MouseDown` event procedure. Thus, for example, your application can check the status of the shift keys and the mouse buttons and then activate the Easter egg only if the right mouse button is pressed while the `Shift` key is held down and while the mouse pointer is located within 1 centimeter of the upper-right corner of a specific picture box.

Alternatively, by using static variables in your `MouseUp` or `MouseDown` event procedure, you can detect a specific sequence of left and right button clicks or watch for something like five clicks in less than 2 seconds.

In a similar way, you can secretly monitor a sequence of specific keypresses by setting the form's `KeyPreview` property to `True` and using the `KeyDown` event to keep track of recent keypresses. Let's look at some sample code to see how you might secretly detect the keypress sequence *EGG*. I've created a simple class module named *Egg* to handle the key code checking. The defined *Egg* object has just one write-only property, named *Char*. The object will display a message box if the key codes for *E*, *G*, and *G* are set in this property in exactly that sequence.

```
Option Explicit

Private mstrKeyPhrase As String * 3

`~~~.Char
Property Let Char(intKey As Integer)
    Select Case intKey
        Case vbKeyE: mstrKeyPhrase = Mid$(mstrKeyPhrase, 2) & "E"
        Case vbKeyG: mstrKeyPhrase = Mid$(mstrKeyPhrase, 2) & "G"
        Case Else: mstrKeyPhrase = ""
    End Select
    If mstrKeyPhrase = "EGG" Then EasterEgg
End Property
```

```
End Property
```

```
Private Sub EasterEgg()
    MsgBox "JC and JW were here!!!!"
End Sub
```

To try out the Egg object, add the following code to the main form:

```
Option Explicit

Dim eggTest As New Egg

Sub Form_Load()
    Me.KeyPreview = True
End Sub

Sub Form_KeyDown(intKeyCode As Integer, intShift As Integer)
    eggTest.Char = intKeyCode
End Sub
```

The Form_Load event procedure sets the form's KeyPreview property to *True* so that all keypresses can be checked, no matter which control on the form has the focus. The private EasterEgg procedure in the Egg object displays a simple message box when *EGG* is typed. You'll want to enhance this program for your own use.

The Form_KeyDown event procedure is activated whenever the form, or any control on the form, has the focus and a key is pressed. The *eggTest* instance of the Egg object accumulates the three most recent keypresses by monitoring its Char property, and only if the pattern *EGG* is detected is the private EasterEgg procedure called.

SEE ALSO

- The Dialogs application in Chapter 34, "[Advanced Applications](#)," for a hidden message screen that shows more detail and provides a working example of the activation of an Easter egg by a pattern of mouse clicks

Dear John, How Do I... Create a Password Dialog Box?

In the earliest versions of Visual Basic, you had to jump through hoops and get out the smoke and mirrors to create a password dialog box, but in recent editions of Visual Basic this is an easy task.

The goal is to create a dialog box that displays asterisks, or some other chosen character, when the user types a secret password. The asterisks provide visual feedback about the number of characters typed, without giving away the actual password to anyone lurking within eyesight of the screen. The program must keep track of the actual characters typed by the user, of course, so that the password can be verified. A password dialog box is shown in Figure 18-2.

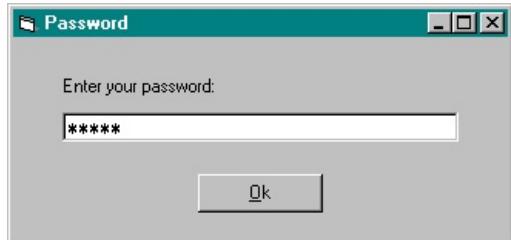


Figure 18-2. A typical password dialog box.

A TextBox control is usually set up so that the user can enter text from the keyboard. The TextBox control has a property named PasswordChar. Set this property to the character that you want to use to hide the password as it is entered. Notice that the Text property will contain the actual characters entered, even though they won't be displayed.

The following code illustrates the basic technique. Add a text box named *txtPassword* and a command button named *cmdOK* to a form. The *cmdOK_Click* event procedure checks for a password match on *sesame*, but you can modify the code to check for matches with any string you want.

```
Option Explicit
```

```
Private Sub cmdOK_Click()
    If txtPassword.Text <> "sesame" Then
        MsgBox "Incorrect password", vbCritical
    Else
        MsgBox "Okay! Dear John, How Do I... Correct password"
    End If
End Sub

Private Sub Form_Load()
    txtPassword.PasswordChar = "*"
    txtPassword.Text = ""
End Sub
```

SEE ALSO

- The Secret application in Chapter 34, "[Advanced Applications](#)," for a more complete demonstration of this topic

Dear John, How Do I... Encrypt a Password or Other Text?

Cipher techniques range from simplistic to extremely complex and secure. In most cases, you don't need or really want the level of security required by the National Security Agency; you just don't want the user to scan through your EXE file or a data file to discover copyright strings or other sensitive information. In fact, you've got to be careful about powerful ciphers, especially if there's any chance that your software will be shipped to other countries. The same laws that cover the shipping of munitions overseas apply to the exportation of strong ciphers! The following technique provides an ASCII-to-ASCII cipher suitable for hiding sensitive information from virtually all of the curious people out there. It's probably not secure enough to keep out the most determined hacker, though.

About the Cipher Class

The Cipher class module defines an object to encrypt and decrypt one string at a time. The two properties and three methods of this object are described here:

- **KeyString** This write-only property (there's a Property Let procedure but no corresponding Property Get) sets the string used to define a key for unique encryption or decryption. When this property is set, Visual Basic's internal random number generator is set to a unique starting seed value based on each and every character of the key and the order in which the characters appear in the key.

NOTE

To initialize Visual Basic's random number generator to repeat a given sequence, you must use the Rnd function with an optional, negative value parameter and then call the Randomize statement with a value. For example, *Randomize (Rnd(-1.23))* will initialize the generator so that the same sequence of random numbers will be generated each time Rnd is called.

- **Text** This property holds the text to be encrypted or decrypted. For example, you might set this property to a readable string, call the following methods to encrypt and stretch the string, and then save the resulting value of this property to a protected file, where it would be unreadable. Later you'd load the Text property with the encrypted string from the file, set the KeyString property to what was used to encrypt the string, call the methods to shrink and decrypt the string, and then display the Text property contents.
- **DoXor** This method processes the string contents of the Text property by applying the exclusive-or operator to each byte of the string with the next pseudorandom byte in the sequence defined by the KeyString property. This is a reversible process—a second processing of the string using the same key string-defined sequence of pseudorandom bytes returns the string to its original state. This technique is at the heart of many encryption algorithms.
- **Stretch** This method was added to the Cipher object, along with the corresponding Shrink method, to convert any string to a printable, displayable string. In other words, a string that can contain nonprintable or nondisplayable binary byte values is converted to a slightly longer string in which all characters of the string are guaranteed to be displayable and printable. Consider, for instance, a string containing 10 tab characters. (Tab characters have a binary byte value of 9.) If you try to print or display such a string, you'll end up not seeing anything, except perhaps a big gap in your output. When you perform an exclusive-or operation on the bytes of a string using all possible pseudorandom byte values, some of the resulting bytes will be tabs, some will be carriage returns, and some will be stranger yet, such as one of the "graphics" characters. The Stretch method borrows bits from every group of three characters to form a fourth character, and all four characters are then mapped to a range of character byte values that are all printable and displayable. The resulting string can be stored in the Registry or in an INI file, printed on paper, or included in e-mail sent over the Internet.
- **Shrink** This method undoes what the Stretch method does to a string, converting a string containing printable, displayable characters to one that might contain any of the possible set of 256 character byte values.

NOTE

For those interested in such things, I've implemented the Stretch and Shrink methods using an algorithm almost identical to the algorithm in Uuencode. I say "almost" because I've used a different offset value to map the stretched string characters in order to avoid using the space character.

Putting the Cipher Object to Work

The Cipher class uses a simple exclusive-or algorithm based on Visual Basic's random number generator to encrypt your text and uses the same algorithm to decrypt the text. Any string is used as the key, and each unique key will produce a unique encryption.

The Cipher class code listing shows how these properties and methods are implemented. To demonstrate its use, add the following code to a form containing two text boxes, two labels, and two command buttons:

Option Explicit

```
Private Sub cmdEncrypt_Click()
    Dim cipherTest As New Cipher
    cipherTest.KeyString = txtKey.Text
    cipherTest.Text = txtClear.Text
    cipherTest.DoXor
    cipherTest.Stretch
    txtEncrypted.Text = cipherTest.Text
End Sub

Private Sub cmdDecrypt_Click()
    Dim cipherTest As New Cipher
    cipherTest.KeyString = txtKey.Text
    cipherTest.Text = txtEncrypted.Text
    cipherTest.Shrink
    cipherTest.DoXor
    txtDecrypted.Text = cipherTest.Text
End Sub
```

This code assumes that *txtClear* contains the original unencrypted text (sometimes called clear text), *txtKey* contains the key string, *txtEncrypted* contains the encrypted version of the clear text, and *txtDecrypted* contains the resulting text after it has been decrypted. Clicking *cmdEncrypt* performs the encryption, and clicking *cmdDecrypt* performs the decryption. Notice that the encrypted result is approximately 4/3 the length of the clear text—the effect of the Stretch method.

Figure 18-3 shows an example of the Cipher object in action.

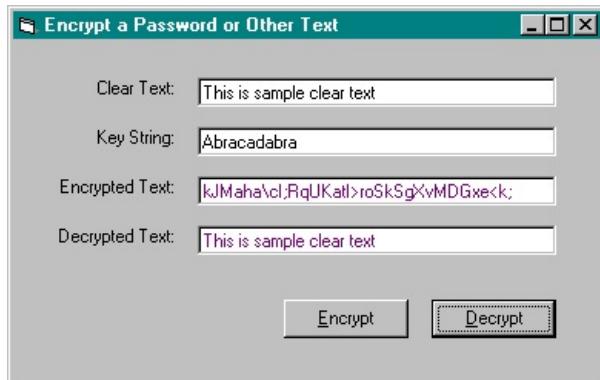


Figure 18-3. A Cipher object being used to encrypt and decrypt text.

The following should provide you with enough code to implement a simple string ciphering object in your

own applications:

```

`CIPHER.CLS
Option Explicit

Private mstrKey As String
Private mstrText As String

`~~~.KeyString
`A string (key) used in encryption and decryption
Public Property Let KeyString(strKey As String)
    mstrKey = strKey
    Initialize
End Property

`~~~.Text
`Write text to be encrypted or decrypted
Public Property Let Text(strText As String)
    mstrText = strText
End Property

`Read text that was encrypted or decrypted
Public Property Get Text() As String
    Text = mstrText
End Property

`~~~.DoXor
`Exclusive-or method to encrypt or decrypt
Public Sub DoXor()
    Dim lngC As Long
    Dim intB As Long
    Dim lngN As Long
    For lngN = 1 To Len(mstrText)
        lngC = Asc(Mid(mstrText, lngN, 1))
        intB = Int(Rnd * 256)
        Mid(mstrText, lngN, 1) = Chr(lngC Xor intB)
    Next lngN
End Sub

`~~~.Stretch
`Convert any string to a printable, displayable string
Public Sub Stretch()
    Dim lngC As Long
    Dim lngN As Long
    Dim lngJ As Long
    Dim lngK As Long
    Dim lngA As Long
    Dim strB As String
    lngA = Len(mstrText)
    strB = Space(lngA + (lngA + 2) \ 3)
    For lngN = 1 To lngA
        lngC = Asc(Mid(mstrText, lngN, 1))
        lngJ = lngJ + 1
        Mid(strB, lngJ, 1) = Chr((lngC And 63) + 59)
        Select Case lngN Mod 3
            Case 1
                lngK = lngK Or ((lngC \ 64) * 16)
            Case 2
                lngK = lngK Or ((lngC \ 64) * 4)
            Case 0
                lngK = lngK Or (lngC \ 64)
                lngJ = lngJ + 1
                Mid(strB, lngJ, 1) = Chr(lngK + 59)
                lngK = 0
        End Select
    Next lngN
End Sub

```

```

Next lngN
If lngA Mod 3 Then
    lngJ = lngJ + 1
    Mid(strB, lngJ, 1) = Chr(lngK + 59)
End If
mstrText = strB
End Sub
`~~~.Shrink
`Inverse of the Stretch method;
`result can contain any of the 256-byte values
Public Sub Shrink()
    Dim lncG As Long
    Dim lndG As Long
    Dim lneG As Long
    Dim lnaA As Long
    Dim lnbB As Long
    Dim lncN As Long
    Dim lncJ As Long
    Dim lncK As Long
    Dim strB As String
    lnaA = Len(mstrText)
    lnbB = lnaA - 1 - (lnaA - 1) \ 4
    strB = Space(lnbB)
    For lncN = 1 To lnbB
        lncJ = lncJ + 1
        lncC = Asc(Mid(mstrText, lncJ, 1)) - 59
        Select Case lncN Mod 3
            Case 1
                lncK = lncK + 4
                If lncK > lnaA Then lncK = lnaA
                lncE = Asc(Mid(mstrText, lncK, 1)) - 59
                lndD = ((lncE \ 16) And 3) * 64
            Case 2
                lndD = ((lncE \ 4) And 3) * 64
            Case 0
                lndD = (lncE And 3) * 64
                lncJ = lncJ + 1
        End Select
        Mid(strB, lncN, 1) = Chr(lncC Or lndD)
    Next lncN
    mstrText = strB
End Sub

`Initializes random numbers using the key string
Private Sub Initialize()
    Dim lncN As Long
    Randomize Rnd(-1)
    For lncN = 1 To Len(mstrKey)
        Randomize Rnd(-Rnd * Asc(Mid(mstrKey, lncN, 1)))
    Next lncN
End Sub

```

When using the Cipher object in your own applications, be sure to set the **KeyString** property immediately before each call to the **DoXor** method. Setting **KeyString** causes the pseudorandom numbers to be initialized to a repeatable starting point.

NOTE

These ciphered strings also work well for sending data over a modem because no hidden escape code sequences or control codes will mess up the communication. You won't have to use any complicated binary transfer method. Be aware that you don't have to use the

Shrink and Stretch methods if you don't mind working with binary data. Note also that the Shrink and Stretch methods might be useful without the accompanying exclusive-or encryption. Sending binary files (graphics images, for example) through standard e-mail over the Internet is one instance in which you might want to implement Shrink and Stretch without encryption.

Securing Registry Data

Finally, here's an idea for making demonstration software secure. You can keep track of the installation date, number of runs, user name, or other information for an application by encrypting a string and storing it in the Registry entry for the application. Your user's name and organization can be used for the key string to encrypt the data stored in the Registry string. This makes it easy for the program to take appropriate action if the demonstration is out of date, if the maximum number of runs has been reached, or if the user name or organization has been tampered with.

Here's a simple example based on the Cipher class. The following code fragment allows the user to enter a key string and a user name. When the *cmdEncrypt* button is clicked, an encrypted version of the user name is stored in the Registry. A click of the *cmdDecrypt* button then displays the decrypted user name. After running this code and entering a sample user name such as *Santa Claus*, you can run Windows 95's Regedit program and search on *User Name* to find the encrypted entry. Don't search on *Santa Claus*, because that string is stored in encrypted format.

```
Option Explicit
```

```
Private mstrAppName As String
Private mstrSection As String
Private mstrKey As String
Private mstrSetting As String

Private Sub cmdEncrypt_Click()
    Dim cipherTest As New Cipher
    cipherTest.KeyString = txtKey.Text
    cipherTest.Text = txtUser.Text
    cipherTest.DoXor
    cipherTest.Stretch
    mstrSetting = cipherTest.Text
    mstrAppName = App.Title
    mstrSection = "Testing"
    mstrKey = "User Name"
    SaveSetting mstrAppName, mstrSection, mstrKey, mstrSetting
    txtEncrypted.Text = cipherTest.Text
End Sub

Private Sub cmdDecrypt_Click()
    Dim cipherTest As New Cipher
    mstrAppName = App.Title
    mstrSection = "Testing"
    mstrKey = "User Name"
    cipherTest.Text = GetSetting(mstrAppName, mstrSection, mstrKey)
    cipherTest.KeyString = txtKey.Text
    cipherTest.Shrink
    cipherTest.DoXor
    txtDecrypted.Text = cipherTest.Text
End Sub
```

Figure 18-4 shows an example of the output.

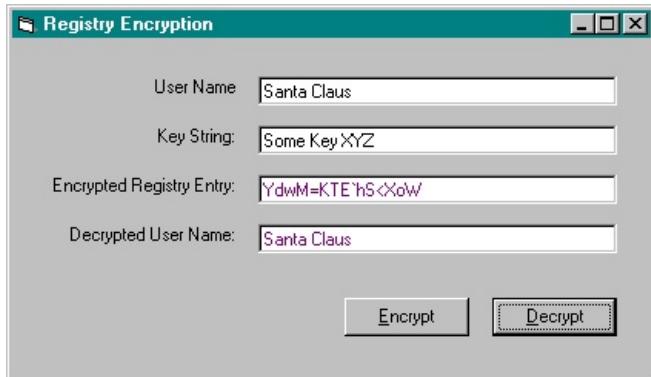


Figure 18-4. Example of an encrypted user name stored in the Registry.

There's room for creative variations on this theme, but the important concept to note is that strings ciphered with the object presented here are compatible with standard printable and displayable text strings. You can read and write them with no problem, even though they appear to be a random sequence of characters.

SEE ALSO

- The Secret application in Chapter 34, "[Advanced Applications](#)," for a thorough demonstration of the cipher algorithm
- Chapter 16, "[The Registry](#)," for more information about accessing the Registry

Dear John, How Do I... Work with Internet Security Features?

For very simple security concerns you can use some of the ideas presented in this chapter. These methods can provide simple, low-security text and binary file encryption and password verification for Internet applications. However, if you have serious security issues that need to be addressed, go to the experts! The phrase "Don't try this at home!" comes to mind when I think of the ramifications of incorrect or incomplete attempts at Internet security. You might be surprised at the amount of work that has gone into providing many different, high-quality security technologies for Internet deployment.

Microsoft has taken the security issues to heart. The company has experts, both in-house and in partnerships with companies that specialize in security software solutions, who know the answers and make it relatively easy to implement them. The full subject is beyond the scope of this book, but I can provide a good starting point for you to gain more information and knowledge on the subject. Go to <http://www.microsoft.com/security> on the World Wide Web to get started. This Web page provides starting points for information about a wide variety of security technologies that you can trust.

Chapter Nineteen

The Mouse

Visual Basic provides 16 standard mouse pointers and also lets you define your own mouse pointer. This chapter explains the manipulation of these pointers and shows you how to easily create custom mouse pointers to use in your applications.

Dear John, How Do I... Change the Mouse Pointer?

Forms and many visible controls have a property named `MousePointer` that allows you to control the appearance of the mouse pointer when it is displayed in front of the form or control. This property is usually set programmatically at runtime because the appropriate mouse pointer might vary according to what the program is executing. (For example, while a time-consuming task is being performed it is usually best to set the mouse pointer to an hourglass shape.) The `MousePointer` property remains set as `vbDefault` in the development environment.

Visual Basic provides a handy collection of constant declarations for setting the `MousePointer` property. You don't have to load a special file into your project—the following constants are readily available in Visual Basic whenever you need to use them in your program. To view a full list of constants and paste the appropriate constant directly into your code, open the Object Browser, select `MousePointerConstants` from the Classes list for the VBRUN library, select the constant, press Ctrl-C to copy the constant, switch to the code window, and then press Ctrl-V to paste the constant. The table below lists these constants and describes the appearance of the mouse pointer when the constant is applied to the `MousePointer` property.

Constant	Value	Mouse Pointer Description
<code>vbDefault</code>	0	Default: shape determined by the object
<code>vbArrow</code>	1	Arrow
<code>vbCrosshair</code>	2	Cross (crosshair pointer)
<code>vblbeam</code>	3	I-beam
<code>vblconPointer</code>	4	Icon
<code>vbSizePointer</code>	5	Size: four-headed arrow
<code>vbSizeNESW</code>	6	Size: NE-SW double-headed arrow
<code>vbSizeNS</code>	7	Size: N-S double-headed arrow
<code>vbSizeNWSE</code>	8	Size: NW-SE double-headed arrow
<code>vbSizeWE</code>	9	Size: W-E double-headed arrow
<code>vbUpArrow</code>	10	Up arrow
<code>vbHourglass</code>	11	Hourglass
<code>vbNoDrop</code>	12	No drop
<code>vbArrowHourglass</code>	13	Arrow and hourglass
<code>vbArrowQuestion</code>	14	Arrow and question mark
<code>vbSizeAll</code>	15	Size all
<code>vbCustom</code>	99	Custom icon specified by the <code>Mouselcon</code> property

Notice that there are four double-headed arrows for sizing. Each one of these arrows points in two directions. The pointing directions are referenced as if your screen were a map, with up as north, down as south, left as west, and right as east. For example, the mouse pointer set with the `vbSizeNS` constant is a double-headed arrow pointing up and down, and the `vbSizeWE` constant specifies a double-headed arrow that points left and right. The other two double-headed arrows point diagonally toward the corners of the screen.

Notice also that the fairly common "hand" pointer is missing from the list of standard mouse pointers. Visual Basic lets you create your own mouse pointer shapes. The `MousePtr` application presented in Chapter 33, "[Utilities](#)," demonstrates how to load and view any icon file as a mouse pointer.

One common use of the `MousePointer` property is for changing the mouse pointer shape to an hourglass

while your program is busy. This pointer lets the user know that something is going on and encourages the user to wait patiently. Figure 19-1 shows the hourglass mouse pointer in action. Be sure to change the pointer back to its previous state when the program has finished its time-consuming task.

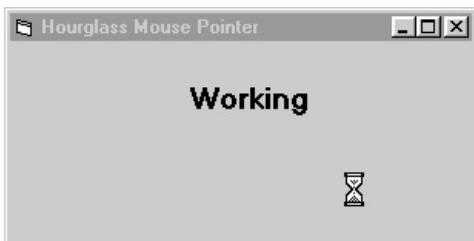


Figure 19-1. The hourglass mouse pointer that appears while the application is busy.

The following lines of code illustrate this technique; the MousePtr application in Chapter 33 provides a more detailed example.

```
frmMain.MousePointer = vbHourglass  
ReturnValue = TimeConsumingFunction()  
frmMain.MousePointer = vbDefault
```

SEE ALSO

- "[Dear John, How Do I... Create a Custom Mouse Pointer?](#)" for more information about creating and using custom mouse pointers
- The MousePtr application in Chapter 33, "[Utilities](#)," for an opportunity to experiment with the standard mouse pointers and see what they look like

Dear John, How Do I... Create a Custom Mouse Pointer?

You can use any icon or cursor file as a custom mouse pointer. To do this, set the `MousePointer` property to `vbCustom` and set the `Mouselcon` property to the name of the icon (ICO) or cursor (CUR) file. That's all there is to it! You can quickly switch between any of the standard mouse pointers and the custom pointer by resetting the `MousePointer` property. If you want to change to a second custom pointer, you'll have to assign a different icon to the `Mouselcon` property. Figure 19-2 shows a left-pointing hand icon loaded from the icons included with Visual Basic.



Figure 19-2. Displaying a custom mouse pointer using an icon file.

NOTE

Animated cursors are not yet supported by Visual Basic's `Mouselcon` property.

You can load multiple icon files into an `ImageList` control at design time and access them from the `ListImages` `Picture` property to quickly flip through a set of custom mouse pointers. The `ImageList` control is one of the Microsoft Windows Common Controls (MSCOMCTL.OCX). The following code uses an `ImageList` control to display a different custom pointer each time the form is clicked on:

```
Option Explicit

Private Sub Form_Click()
    Static intList As Integer
    intList = intList + 1
    If intList > 3 Then intList = 1
    MousePointer = vbCustom
    MouseIcon = ImageList1.ListImages(intList).Picture
End Sub
```

Dear John, How Do I... Display an Animated Mouse Pointer?

It puzzles me that Microsoft doesn't include support for 32-bit animated mouse pointers in the Mouselcon property. In order to display an animated pointer for a window, you have to go to the Windows API. This is not terribly difficult, however.

You can load any type of mouse pointer supported by Microsoft Windows using the LoadCursorFromFile Windows API function. To change the mouse pointer for a window, use the SetClassLong Windows API function. The following code shows how this is done:

```
Option Explicit

`Loads cursors and creates mouse pointer handle
Private Declare Function LoadCursorFromFile _
Lib "user32" Alias "LoadCursorFromFileA" ( _
    ByVal lpFileName As String _
) As Long

`Changes class information for a window
Private Declare Function SetClassLong _
Lib "user32" Alias "SetClassLongA" ( _
    ByVal hwnd As Long, _
    ByVal nIndex As Long, _
    ByVal dwNewLong As Long _
) As Long

`Index of mouse pointer in window class structure
Private Const GCL_HCURSOR = (-12)
Private hOldCursor As Long

Private Sub Form_Load()
    Dim hNewCursor As Long
    `Get handle to new animated mouse pointer
    hNewCursor = LoadCursorFromFile _
        ("C:\WINDOWS\CURSORS\DRUM.ANI")
    `Replace window's mouse pointer with new
    `animated mouse pointer
    hOldCursor = SetClassLong(Form1(hwnd, GCL_HCURSOR, _
        hNewCursor)
End Sub

Private Sub Form_Unload(Cancel As Integer)
    `Restore original mouse pointer to prevent animated
    `mouse pointer from being retained
    hOldCursor = SetClassLong(Form1(hwnd, GCL_HCURSOR, _
        hOldCursor)
End Sub
```

Notice that the Form_Unload event procedure resets the mouse pointer to its original state. If you don't do this, the drum pointer is displayed for every form at runtime—even forms from other projects—until you close and restart Visual Basic. You wouldn't want to hard-code the path to DRUM.ANI in code that you plan to distribute. Instead, use the GetWindowsDirectory Windows API function to get the real Windows directory.

Dear John, How Do I... Determine Where the Mouse Pointer Is?

The `MouseMove`, `MouseUp`, and `MouseDown` events can provide you with several useful parameters when they are activated. Search the Visual Basic online help for these events to get a full explanation of their parameters. Note that these events all provide x- and y-coordinates that tell your application exactly where the mouse pointer is located at the time the event occurs. In most cases, you'll need to copy these x- and y-coordinates to more permanent variables to keep track of them, depending on what you're trying to accomplish.

A simple example demonstrates this technique clearly. The following code allows you to draw lines on your form: one endpoint appears where the mouse button is pressed, and the other endpoint appears where the mouse button is released. Notice that the location of the mouse pointer at the `MouseDown` event is stored in the module-level variables `X1` and `Y1` so that these values are available to the `Form_MouseUp` event procedure. When the `MouseUp` event occurs, the `X1` and `Y1` module-level variables are used with the `X` and `Y` local variables as the starting and ending coordinates for the line.

```
Option Explicit

Private X1, Y1

Private Sub Form_MouseDown( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single _
)
    X1 = X
    Y1 = Y
End Sub

Private Sub Form_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single _
)
    Line (X1, Y1)-(X, Y)
End Sub
```

Figure 19-3 shows this simple line-drawing code in action.

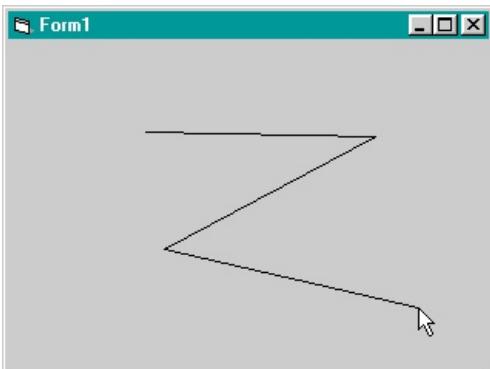


Figure 19-3. Using the `MouseUp` and `MouseDown` events to draw straight line segments.

SEE ALSO

- The `MousePtr` application in Chapter 33, "[Utilities](#)," for a demonstration of mouse pointer location

Chapter Twenty

The Keyboard

Here are a few useful ideas for handling user keypresses from within a running Visual Basic application. This chapter covers several techniques that you might otherwise overlook and that you might want to add to your bag of tricks.

Dear John, How Do I... Change the Behavior of the Enter Key?

Although it's not standard Windows programming practice, you might occasionally want to have the Enter key act like a Tab key when the focus is on a particular control—that is, you might want a press of the Enter key to move the focus to the next control instead of having it cause any other action. The following code does the trick for a text box and can be modified to work on any other control that provides a KeyPress event:

```
Private Sub txtText1_KeyPress(KeyAscii As Integer)
    If KeyAscii = vbKeyReturn Then
        SendKeys "{tab}"
        KeyAscii = 0
    End If
End Sub
```

The ASCII code for the pressed key is handed to the KeyPress event in the *KeyAscii* parameter. The ASCII value of the Enter key is 13, which is the value of the built-in constant *vbKeyReturn*. SendKeys lets your Visual Basic application send any keypress to the window that currently has the focus. (This is a very handy statement because it lets your application send keypresses to other Windows-based applications just as easily as to itself.) The string *{tab}* shows how SendKeys sends a Tab keypress, which will be processed exactly as if the Tab key had really been pressed. To override the default action of the Enter key, set the *KeyAscii* value to 0. If you don't do this, you'll get a beep from the control.

If you want to ignore all Enter keypresses when the focus is on a particular control, you can easily set this up: simply assign the value 0 to *KeyAscii* to ignore the Enter key, as above, and don't use SendKeys to substitute any other keypress.

Dear John, How Do I... Determine the State of the Shift Keys?

The KeyPress event does not directly detect the state of the Shift, Ctrl, and Alt keys (collectively known as the shift keys) at the time of a keypress, but the Shift key state does modify the character that is detected (by making it an uppercase or a lowercase letter, for example). To directly detect the state of these shift keys, you can use the closely related KeyDown and KeyUp events. You can act on the state of these keys directly in the KeyDown and KeyUp event procedures, or you can keep track of their states in module-level variables. I prefer the second technique in many cases because it lets me act on the shift keys' states from within the KeyPress event procedure or from any other code in the module.

The following code immediately updates the state of one of three Boolean variables whenever any of the shift keys is pressed or released. I've used the Visual Basic constants `vbShiftMask`, `vbCtrlMask`, and `vbAltMask` to test for each of the shift states and return a Boolean value. To enable this code for the entire form, be sure to set the form's KeyPreview property to *True*. Then, regardless of which control is active, your code can instantaneously check the state of the three shift keys simply by referring to the current value of these variables.

```
Option Explicit
```

```
Private mblnShiftState As Boolean
Private mblnCtrlState As Boolean
Private mblnAltState As Boolean

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    mblnShiftState = (Shift And vbShiftMask)
    mblnCtrlState = (Shift And vbCtrlMask)
    mblnAltState = (Shift And vbAltMask)
End Sub

Private Sub Form_KeyUp(KeyCode As Integer, Shift As Integer)
    mblnShiftState = (Shift And vbShiftMask)
    mblnCtrlState = (Shift And vbCtrlMask)
    mblnAltState = (Shift And vbAltMask)
End Sub

Private Sub tmrTest_Timer()
    Cls
    Print "Shift = "; mblnShiftState
    Print "Ctrl  = "; mblnCtrlState
    Print "Alt   = "; mblnAltState
End Sub
```

To see how this code works, create a new form, add the code to the form, and add a timer named `tmrTest`. Set the timer's Interval property to 100 to sample the state of the keys every 0.1 second. The preceding code will then display the state of the shift keys as you press them. Try holding down combinations of the Shift, Ctrl, and Alt keys to see how all three state variables are updated independently. Figure 20-1 shows the form as it appears when the Ctrl and Alt keys are simultaneously held down.



Figure 20-1. Real-time display of the status of the shift keys.

Dear John, How Do I... Create Hot Keys?

The KeyPreview property for Visual Basic forms provides an excellent way to set up hot keys. This property lets your application act on any combination of keypresses, such as function keys, shifted function keys, or numeric keypad keys. Here's the general technique: first set your form's KeyPreview property to *True*, and then add code to the form's KeyDown event procedure to check for and act on any desired keypresses. In the code below, I check for F1, F2, and any of the shift keys in combination with the F3 key. I use the Visual Basic constants *vbKeyF1*, *vbKeyF2*, and *vbKeyF3* to identify the key that is pressed. Note that this type of procedure works well to test one or a few keys; the technique in the previous section would help simplify the key tests if we needed to check several function keys for the current shift keys' states.

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    Select Case KeyCode
        Case vbKeyF1
            Print "F1"
        Case vbKeyF2
            Print "F2"
        Case vbKeyF3
            If (Shift And vbShiftMask) Then
                Print "Shift-F3"
            ElseIf (Shift And vbCtrlMask) Then
                Print "Ctrl-F3"
            ElseIf (Shift And vbAltMask) Then
                Print "Alt-F3"
            Else
                Print "F3"
            End If
    End Select
End Sub
```

Figure 20-2 shows the result of running this code and pressing a few of the function keys. Note that the form intercepts the keypresses before the Command1 button receives them, even though Command1 has the focus.

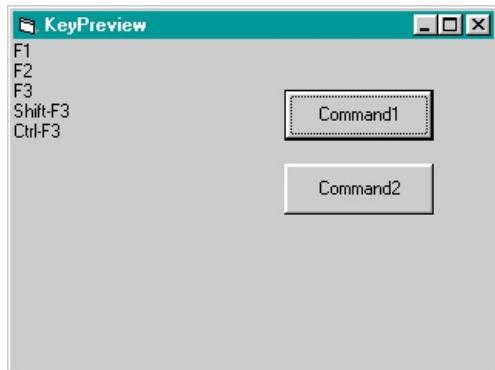


Figure 20-2. A demonstration of the hot key code.

SEE ALSO

- "[Dear John, How Do I... Determine the State of the Shift Keys?](#)" earlier in this chapter
- The Jot application in Chapter 32, "[Databases](#)," for a demonstration of this hot key setup

Chapter Twenty-One

Text Box and Rich Text Box Tricks

One of the most powerful and useful controls in Visual Basic is the RichTextBox control. With just a few simple changes to the control's property settings, a rich text box can become a decent editor, with capabilities similar to those of Windows' WordPad utility. This chapter shows you how to accomplish this and also explains a few other techniques that take your TextBox and RichTextBox control capabilities to new heights.

Dear John, How Do I... Display a File?

You can set up either a text box or a rich text box as a convenient way to display the contents of a file. The following code loads the AUTOEXEC.BAT file and displays its contents in a scrollable window. To try this example, draw one of these controls on a new form and name it either *txtTest* (if you're using a text box) or *rtfTest* (if you're using a rich text box). If you're using a TextBox control, you'll need to set a couple of the control's properties: set MultiLine to *True* and ScrollBars to *3 - Both*. If you want to view a file other than AUTOEXEC.BAT, change the filename in the Open statement. In this example, I've used a RichTextBox control to display my AUTOEXEC.BAT file, but you can easily substitute a TextBox control if you want.

```
Option Explicit
```

```
Private Sub Form_Load()
    Dim strF As String
    'Load a file into a string
    Open "C:\AUTOEXEC.BAT" For Binary As #1
    strF = Space$(LOF(1))
    Get #1, , strF
    Close #1
    'Display file in rich text box
    rtfTest.Text = strF
End Sub
```

Figure 21-1 shows the displayed file.

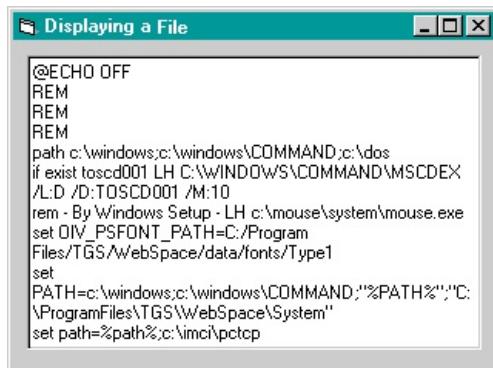


Figure 21-1. The contents of AUTOEXEC.BAT displayed in a rich text box.

In the previous example, the entire AUTOEXEC.BAT file is read into a single string using binary file input. Binary file input is a useful technique because it reads every byte from the file regardless of the content of the byte. This means that when a file with multiple lines is read into a single string, the linefeeds and carriage returns are included in the string. (Chapter 15, "[File I/O](#)," goes into greater detail about binary file input/output techniques.)

If you are reading a rich text format (RTF) file, use the rich text box's TextRTF property instead of the plain Text property. When the RTF file is read, the formatting codes in the file will be interpreted and the text correctly displayed.

Here's another technique that you should know about for building a multiline string: you can concatenate strings (tack them together end to end), but you must insert a carriage return and a linefeed at the end of each string, as demonstrated in the following code. This example performs the same action as the previous example, except that in this case AUTOEXEC.BAT is loaded from the file one line at a time to build up the string for the rich text box. Compare this code with the previous listing.

```
Option Explicit
```

```
Private Sub Form_Load()
    Dim strA As String, strF As String
    'Load a file into a string
    Open "C:\AUTOEXEC.BAT" For Input As #1
    Do Until EOF(1)
```

```
Line Input #1, strA
strF = strF & strA & vbCrLf
Loop
Close #1
`Display file in rich text box
rtfTest.Text = strF
End Sub
```

NOTE

Prior to Visual Basic 4, you had to create a string of carriage return and linefeed characters yourself. Visual Basic now provides the built-in constant *vbCrLf*, as shown in the example.

In the past, I have used a TextBox control in both of these examples, but the TextBox control has a maximum limit of roughly 64,000 characters. For smaller files, such as the AUTOEXEC.BAT file shown above, a TextBox control will work just fine, but I suggest using the RichTextBox control because of its greater flexibility.

SEE ALSO

- "[Dear John, How Do I... Fit More than 64 KB of Text into a Text Box?](#)" later in this chapter for a workaround for the text box size limitation

Dear John, How Do I... Create a Simple Text Editor?

If you need a full-featured word processor window in your application, consider using OLE capabilities to embed a Microsoft Word document object. But if you want just a simple text editor along the lines of the Windows Notepad or WordPad utility, a TextBox or RichTextBox control is probably all you need. Let's see how you can do this by building a simple editor on a form. The code below shows how to build the text editor.

```
Option Explicit
```

```
Private Sub cmdCut_Click()
    'Cut selected text to clipboard
    Dim strWork As String
    Dim Wstart, Wlength
    'Keep focus on rich text box
    rtfEdit.SetFocus
    'Get working parameters
    strWork = rtfEdit.Text
    Wstart = rtfEdit.SelStart
    Wlength = rtfEdit.SelLength
    'Copy cut text to clipboard
    Clipboard.SetText Mid$(strWork, Wstart + 1, Wlength)
    'Cut out text
    strWork = Left$(strWork, Wstart) +
        Mid$(strWork, Wstart + Wlength + 1)
    rtfEdit.Text = strWork
    'Position edit cursor
    rtfEdit.SelStart = Wstart
End Sub

Private Sub cmdCopy_Click()
    'Keep focus on edit box
    rtfEdit.SetFocus
    'Copy selected text to clipboard
    Clipboard.SetText rtfEdit.SelText
End Sub

Private Sub cmdPaste_Click()
    'Paste text from clipboard
    Dim strWork As String, strClip As String
    Dim Wstart, Wlength
    'Keep focus on rich text box
    rtfEdit.SetFocus
    'Get working parameters
    strWork = rtfEdit.Text
    Wstart = rtfEdit.SelStart
    Wlength = rtfEdit.SelLength
    'Cut out text, if any, and insert clipboard text
    strClip = Clipboard.GetText()
    strWork = Left$(strWork, Wstart) + strClip +
        Mid$(strWork, Wstart + Wlength + 1)
    rtfEdit.Text = strWork
    'Position edit cursor
    rtfEdit.SelStart = Wstart + Len(strClip)
End Sub
```

Figure 21-2 shows the completed text editor in action.

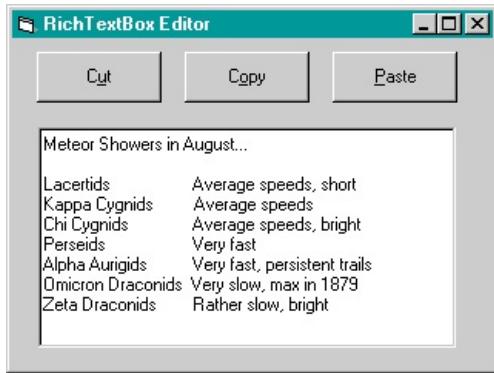


Figure 21-2. A rich text box editor in action.

In this case, the RichTextBox control is named *rtfEdit*. Set the RightMargin property to make the text wrap automatically. For instance, the following code makes the text wrap when the user types to the edge of the rich text box:

```
rtfEdit.RightMargin = rtfEdit.Width
```

Set the ScrollBar property to 2 - *rtfVertical* to enable text scrolling in long documents.

The form contains three command buttons: *cmdCut*, *cmdCopy*, and *cmdPaste*. You don't really need these buttons if you want to rely on the Ctrl-X, Ctrl-C, and Ctrl-V shortcut keys because these keys are automatically processed by the control to cut, copy, and paste without any additional coding on your part. The code I've added for these command buttons allows them to mimic the action of the control keys and provides more flexibility in your programming. You might, for example, want to add a standard Edit menu to the form, and this code is the only convenient way to process menu events. Simply copy the code to the corresponding menu item Click event procedure to set up your own menu-driven cut, copy, and paste procedures.

So when would you want to use a TextBox control instead of a RichTextBox control? In the 32-bit world, the TextBox control is used primarily for simple data entry fields, but in the 16-bit world the TextBox control is your only option, so you must use it for displaying files and collecting text input. The 32-bit RichTextBox control allows much larger pieces of text, supports OLE drag-and-drop, and displays formatting; almost anything a text box can do, a rich text box can do better. I haven't even touched on the multiple fonts and the rich set of embedded RTF commands or the many methods and properties of the RichTextBox control that make it much more powerful and adaptable than the TextBox control. See the Visual Basic online help and printed documentation for more information about all these details.

SEE ALSO

- The Jot application in Chapter 32, "[Databases](#)," for a demonstration of the use of a rich text box as a text editor

Dear John, How Do I... Detect Changed Text?

The Change event is a text box/rich text box event, and you would think that this would make it easy to detect whether a control's contents are edited by the user while a form is displayed. But detecting changed text in a text box or a rich text box can get a little tricky. Usually the initial or default text is loaded into the Text property when the control's form loads. This causes the control's Change event to occur, even though the text has not yet been changed by the user, which is what we're really trying to detect.

The following code solves this problem for a RichTextBox control by manipulating two Boolean flags, but the same method can be used with a TextBox control. The *blnNotFirstFlag* variable is set to *True* after the first Change event takes place. The *blnTextChangeFlag* variable is set to *True* only if *blnNotFirstFlag* is *True*. This happens the second time the control's Change event occurs, which is when the user makes the first change to the text in the *rtfEdit* rich text box. *blnTextChangeFlag*'s scope is the entire form, so it's easy to check the value of the flag in the form's Unload event procedure. In the following sample code, the unloading of the form is disabled if the contents have been changed by the user:

```
Option Explicit

Private blnTextChangeFlag As Boolean

Private Sub Form_Load()
    rtfEdit.Text = "Edit this stringDear John, How Do I... "
    blnTextChangeFlag = False
End Sub

Private Sub Form_Unload(Cancel As Integer)
    If blnTextChangeFlag = True Then
        Cancel = True
    End If
End Sub

Private Sub rtfEdit_Change()
    Static blnNotFirstFlag As Boolean
    blnTextChangeFlag = blnNotFirstFlag
    blnNotFirstFlag = True
End Sub
```

When you use this technique in your applications, you might want to perform some other action beyond this simple example. To try this example, place a rich text box named *rtfEdit* on a form. When you run the program, change the text in the rich text box and then try to unload the form by pressing Alt-F4 or by clicking the Close button to close the form.

Dear John, How Do I... Fit More than 64 KB of Text into a Text Box?

In 16-bit Visual Basic 4 applications, the Text property for a text box is limited to less than 64 kilobytes (KB) of text. (Actually, depending on how you are using string space, the limit might be around 32 KB.) In Windows 95 and Windows NT, you can use a RichTextBox control instead of a TextBox control to bypass this limitation. This book focuses on programming for 32-bit Windows, so I considered not addressing this topic. However, the techniques shown here, such as using the Preserve keyword with the ReDim statement, are quite useful for other programming tasks, so I decided to include this topic after all. If you are doing 16-bit programming, you'll find it immediately useful. If you are doing 32-bit programming, read on anyway, because the techniques presented here are useful additions to your bag of tricks.

You can work around the string size limitation of the text box by placing the text in an array and then assigning an element of the array to the Text property of a TextBox control. With this method, the individual array elements are limited to less than 64 KB, but because arrays can be huge even in 16-bit Visual Basic, a very large amount of text can be contained in the array and subsequently displayed in the text box. In the example, the text box will contain only a handful of lines, but these lines will be quickly updated from a huge dynamic string array that contains more than 64 KB of text. Figure 21-3 below shows the result of using this technique to display a text file containing approximately 668,000 bytes in a text box.

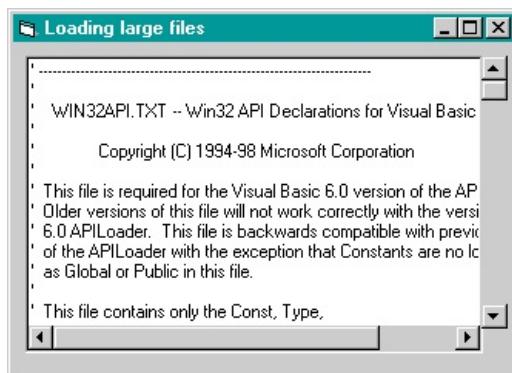


Figure 21-3. Displaying a large text file in a text box.

To re-create this example, add a text box named *txtTest* and a vertical scrollbar named *vsbTest* to a new form. Set the text box's MultiLine property to *True* and its ScrollBars property to *1 - Horizontal*. The vertical scrollbar takes the place of the text box's built-in scrollbar; move the scrollbar to the right edge of the text box so that it appears to be attached. Add the following code to the form, and change the name of the file in the Open statement to the name of a file on your disk.

```

Option Explicit

Private Const LINES = 15
Private strA() As String

Private Sub Form_Load()
    Dim intN
    `Load dynamic string array from large text file
    Open "C:\WIN32API.TXT" For Input As #1 Len = 1024
    Do Until EOF(1)
        intN = intN + 1
        ReDim Preserve strA(intN + LINES)
        Line Input #1, strA(intN)
    Loop
    Close #1
    `Set scrollbar properties
    With vsbTest
        .Min = 1
        .Max = intN
        .SmallChange = 1
        .LargeChange = intN \ 10
    End With
End Sub

```

```
Private Sub vsbTest_Change()
Dim intI As Integer
Dim strTmp As String
    'Create display string from array elements
    For intI = vsbTest.Value To vsbTest.Value + LINES
        strTmp = strTmp + strA(intI) + vbCrLf
    Next intI
    txtTest.Text = strTmp
End Sub
```

At form load time, the entire large text file is read into a dynamically allocated string array. This can take a few seconds, so in a real application you might want to load the file while other things are going on so that the user doesn't become anxious. For this demonstration, you can use any text file greater than 64 KB in size, but be prepared to wait if the file is very large.

Once the file is loaded into memory, the string array dimensions are used to set the vertical scrollbar's properties. The text box contents are updated whenever the vertical scrollbar's value changes. A block of strings from the string array are concatenated, with carriage return and linefeed characters inserted at the end of each line, and the resulting temporary string is copied into the text box's Text property.

This demonstration displays only the file's contents and doesn't attempt to keep track of changes made by the user. If you want to turn this text box into a text editor, you'll have to add code to update the string array when there are changes to the text box's contents.

Dear John, How Do I... Allow the User to Select a Font for a Text Box or a Rich Text Box?

The CommonDialog control provides the most convenient and foolproof way to let the user select a font at runtime. (It's also the best tool for allowing the user to select files, choose colors, set up a printer, and so on.) To experiment with this technique, add a text box named *txtTest*, a command button named *cmdFont*, and a CommonDialog control named *dlgFonts* to a new form. Add the following code to complete the demonstration:

```
Option Explicit

Private Sub cmdFont_Click()
    dlgFonts.Flags = cdlCFScreenFonts
    dlgFonts.ShowFont
    With txtTest.Font
        .Name = dlgFonts.FontName
        .Bold = dlgFonts.FontBold
        .Italic = dlgFonts.FontItalic
        .Size = dlgFonts.FontSize
    End With
End Sub

Private Sub Form_Load()
    txtTest.Text = "ABCDEF abcdef 0123456789"
End Sub
```

The Form_Load event procedure simply loads the text box with some sample text so that the font changes can be seen. The cmdFont_Click event procedure does the interesting stuff. It first sets the CommonDialog control's Flags property to display the system's screen fonts (see the online help for other Flags options), then it activates the ShowFont method to activate the Font dialog box, and then it sets the text box font properties according to the user's selections. Figure 21-4 and Figure 21-5 show the text box before and after font selection, and Figure 21-6 shows the Font dialog box itself.



Figure 21-4. Text box showing font properties at their default settings.

NOTE

The TextBox control allows just one font at a time to be in use for the entire control, but the RichTextBox control allows fonts to vary throughout its text contents. Refer to the rich text box properties that start with *Sel* (there are a lot of them) in the Visual Basic online help for a detailed explanation of how to manipulate character fonts within the RichTextBox control.



Figure 21-5. Text box font set to user's font choice.

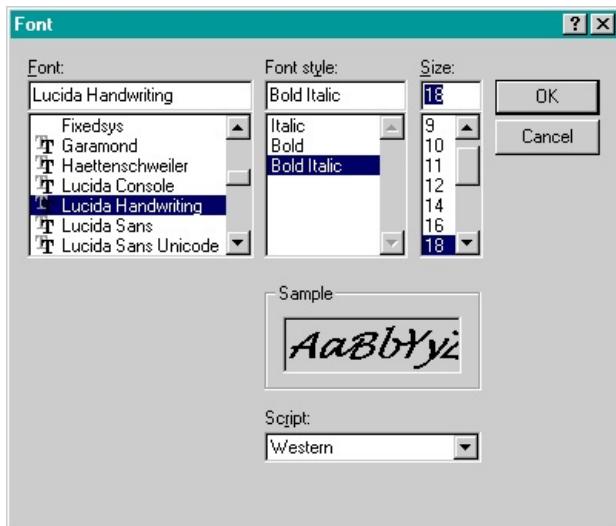


Figure 21-6. The Font dialog box in action.

SEE ALSO

- "[Dear John, How Do I... Scale a Font Infinitely?](#)" in Chapter 14, "[Graphics Techniques](#)," for information about scaling fonts

Chapter Twenty-Two

Multiple Document Interface

For applications that require more than one document open at a time, the most appropriate user interface is the Multiple Document Interface (MDI). However, the subtleties of creating MDI applications have prevented many of us from taking full advantage of this user interface. This chapter highlights some of the main concepts behind creating an MDI application and provides a solid background on which to expand your expertise.

Dear John, How Do I... Create an MDI Application?

This section provides concise coverage of the basic concepts involved in creating an MDI application. For more details, refer to the Visual Basic online help or Visual Basic Books Online.

When I first started working with MDI forms, I had trouble integrating all the relevant information in such a way that it made sense to me. The following discussion should give you a better foothold than I had.

The MDI Form

A Visual Basic project can have only one MDI form. The MDI form is the container for all the individual child forms. To add an MDI form to your project, choose Add MDI Form from the Project menu and double-click the MDI Form icon in the Add MDI Form dialog box. You'll notice that this form has a shorter list of properties than a standard form. The MDI form doesn't have to be the startup form for your application, but you can make it the startup form by choosing Project Properties from the Project menu and, on the General tab of the Project Properties dialog box, selecting the MDI form from the Startup Object drop-down list.

MDI Child Forms

Your project can have any number of child forms contained in an MDI form. (It can have any number of independent, nonchild forms, too.) To create a child form, set a standard form's MDIChild property to *True*. The form becomes a child of the MDI form in your project. At runtime, the program code is responsible for loading these child forms and for showing, hiding, or unloading them as desired. Notice that, at design time, child forms look and act just like standard forms. The big difference in their behavior shows up at runtime.

Figure 22-1 shows a running MDI application with an assortment of child forms. I've minimized two of the child forms and displayed two others.



Figure 22-1. A sample MDI form with an assortment of child forms.

The ActiveForm and ActiveControl Properties

When used together, the ActiveForm and ActiveControl properties let your MDI form refer to controls on the child form that currently has the focus. For example, your MDI application might have identical text controls on several child forms. When text is highlighted in a text box on one of these child forms and when a Copy menu item is chosen, the ActiveForm.ActiveControl.SelText property refers to the text selected in the currently active text box on the currently active child form.

The Me Keyword

The Me keyword acts as a variable, referring to the identity of the active child window. When multiple instances of a child form are created, Me provides a way for the code that was designed for the original form to operate on the specific instance of the form that currently has the focus.

The Tag Property

The Tag property provides a handy way to uniquely identify exact copies of child forms. As each new copy is created, have a number or string placed into its Tag property. This creates an index (somewhat like an array index) that keeps track of forms that otherwise are identical copies of one another.

Fundamental MDI Features

The MDI form is unique among forms in several respects. The following features of the MDI form should be kept in mind when you are designing and coding MDI applications:

- Child forms always appear in the MDI form and never appear elsewhere. Even when minimized, the child form icons appear only in the MDI form. They ride along with the MDI form—for instance, when the MDI form is minimized you'll see nothing of the child forms it contains.
- If a child form has a menu, that menu shows up on the MDI form when that child form has the focus. At runtime, you'll never see menus on any child forms; they instantly migrate to the parent MDI form.
- You can add a menu to the MDI form at design time, but the only controls that can be added are those with an Align property. (There is a way to work around this, though—simply draw a picture box on the MDI form and draw your controls inside this picture box.) When a control is placed on the MDI form, child windows cannot overlap any part of the control.
- You can create multiple instances of a single child form at runtime using the Dim As New statement. In many MDI applications, this is an extremely important feature. As mentioned above, the ActiveForm, ActiveControl, and Tag properties and the Me keyword are very useful for working with multiple copies of child forms. Take a look at the Visual Basic online help for more information.
- A useful MDI form property, *Picture*, lets you install a bitmap or other graphics image onto the backdrop of the MDI form. This is handy for displaying graphics such as company logos.

SEE ALSO

- The Jot application in Chapter 32, "[Databases](#)," to see how a single child edit form is mass-produced to let the user edit in multiple windows

Dear John, How Do I... Add a Logo (Splash Screen) to an MDI Form?

It's tricky to place an image in the center of an MDI form. The Picture property always plops an image in the upper-left corner, and only aligned controls can be placed on an MDI form. But an MDI form makes a good backdrop for an entire application, and it's nice to be able to position your company logo right in the middle of things. Here's a workaround that displays a logo (or any image) in the middle of your MDI form. With this technique, the logo stays centered even if the form is resized. Figure 22-2 shows an imaginary application at startup. In this application, the MDI form is set as the startup form and the company logo appears centered on the MDI form.



Figure 22-2. An MDI form displaying a centered logo.

To try this example, create a new project containing an MDI form and one child form named *frmLogo*. Add an Image control named *imgLogo* to the child form. You can't add the Image control directly to the MDI form, and if you use a picture box to contain the Image control (or to directly display the logo), you won't be able to center the logo if the MDI form is resized. By placing the Image control on the child form, you can always center the child form and thus center the logo. So the trick here is to size *frmLogo* to the size of the image, remove *frmLogo*'s borders and title bar, and then keep *frmLogo* centered on the MDI form. Set *frmLogo*'s MDIChild property to *True*, and set the BorderStyle property to *0 - None*. Load a bitmap logo file into *imgLogo*. Don't worry about the placement of this image on *frmLogo*—the following runtime code will take care of these details. Add this code to the MDI form, and give it a try:

```
Option Explicit
```

```
Private Sub MDIForm_Resize()
    'Move logo to upper-left corner of Image control
    frmLogo.imgLogo.Move 0, 0
    'Size form to size of image it contains
    frmLogo.Width = frmLogo.imgLogo.Width
    frmLogo.Height = frmLogo.imgLogo.Height
    'Center logo form on MDI form
    frmLogo.Left = (ScaleWidth - frmLogo.Width) \ 2
    frmLogo.Top = (ScaleHeight - frmLogo.Height) \ 2
    'Show logo
    frmLogo.Show
End Sub
```

The image is moved to the upper-left corner of its containing form. The form is then resized to match the image size and is relocated to the center of the MDI form. I've put this code in the MDIForm_Resize event procedure so that the logo will always shift to the center of the MDI form if the MDI form is resized.

Chapter Twenty-Three

Database Access

Microsoft's evolving approach to data access has been increasingly evident in recent versions of Visual Basic. Visual Basic has many important and powerful features that make it an ideal language for all of your database programming requirements. Although a full discussion of database programming is beyond the scope of this book, this chapter provides a short introduction to give you a feel for this subject. For more in-depth information, see Visual Basic Books Online and the Visual Basic online help. Your local bookstore probably carries several books devoted entirely to the subject of database programming.

This chapter covers the wizards available in Visual Basic 6 that help you get up and running with databases in record time, presents two different ways to connect a Visual Basic application to a database, and describes Visual Basic's new built-in report-generating capabilities.

Of the two ways to connect your applications to a database, the first technique uses the Data control and is a simple and straightforward approach that requires very little programming. The second method employs data access objects (DAOs), which provide flexible, programmatic control of a database.

Dear John, How Do I... Use Wizards in My Database Development Cycle?

You can save a lot of time and energy by leveraging the tools that Microsoft packages with Visual Basic and Microsoft Access. For example, Visual Basic includes two wizards that can greatly simplify the task of adding a database form to your applications. These wizards create a simplified version of a typical database form, and if you think about it, most of the time this is all you really need. For those occasions when further customization is required, the forms created by these wizards still provide an excellent starting point.

If you know you will be adding a database form to a new Visual Basic project, you can save development time and effort by stepping through the Application Wizard. To start this wizard, choose New Project from the File menu and double-click the VB Application Wizard icon in the New Project dialog box. The Application Wizard can add several useful forms, menus, and similar parts that make up a standard application, including a database form, to the new project. You definitely should become familiar with the Application Wizard—it can save a lot of time in the application development cycle.

For even more options and control over the creation of a database form, you can run the Data Form Wizard whenever you add a new form to your project. To start this wizard, choose Add Form from the Project menu and double-click the VB Data Form Wizard icon. The Data Form Wizard steps you through several design decisions and then uses your input to create a complete, working form connected to a database of your choice.

Visual Basic 6 now provides several other predefined types of database forms that you can use when adding a new form to your project. When you select Add Form from the Project menu, check out the Datagrid and Querys form types. Again, these forms can save a lot of up-front routine and repetitive development efforts.

One of the nice features of the Data Form Wizard is the choice of fundamental database form types. A form that shows one record at a time from a table or query is often all that's required; at other times, you might want a full grid of records. Both of these form types are automatically generated by the Data Form Wizard. A third form type, Master/Detail, is also very useful. This form type displays a single record from a table or a query and a grid of data from the same database. The single record and the grid are connected through one of the common fields. For example, using the BIBLIO.MDB sample database found in the Visual Basic directory, the Data Form Wizard can easily create a form that displays the data for one publisher at a time along with an associated grid of all books published by that publisher, as shown in Figure 23-1. As you move from record to record through the publisher information, the grid automatically fills with the relevant book titles.

Rather than describe the Data Form Wizard's steps in detail here, I recommend that you experiment with the wizard directly. It's easy to use and self-explanatory; with a few minutes of exploration up front, you'll probably save many hours of labor down the road when you need to get database forms up and running as quickly as possible. You can experiment with the BIBLIO.MDB and NWIND.MDB databases that ship with Visual Basic.

If you want to get database forms up and running quickly, an excellent approach is to combine tools, using Access for the creation and maintenance of each database, and using the Data Form Wizard whenever possible. As you'll see in the next few sections, you can use Visual Basic itself to create and maintain your databases and to make custom database forms — but for maximum leverage you can't beat using Access and the Data Form Wizard.

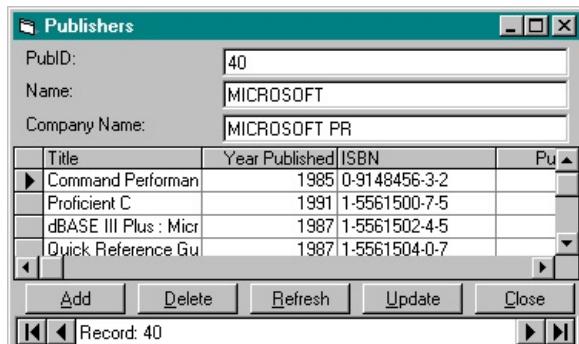


Figure 23-1. Master/Detail-type database form created with the Data Form Wizard.

Dear John, How Do I... Use the Data Control to Connect An Application to a Database?

Let's put together a very simple database using a Data control. You must take two basic steps to accomplish this: first create a database, and then create the user interface.

Creating a Database Using the Visual Data Manager

You can programmatically create a database, which we'll do in the next section, "[Dear John, How Do I... Use Data Access Objects to Connect An Application to a Database?](#)" or you can create a database using the Visual Data Manager add-in. When I am working with a Data control, I prefer to use the Visual Data Manager for this one-time task. This utility lets you create a database by defining its tables and fields.

To create a database using the Visual Data Manager utility, follow these steps:

1. From the Add-Ins menu, choose Visual Data Manager.
2. Click No if a dialog box appears that asks, "Add SYSTEM.MD? (Microsoft Access Security File) to INI File?" For this example, you do not need file security. The Visual Data Manager utility starts, and the Visual Data Manager window is displayed.
3. From the Visual Data Manager's File menu, choose New, choose Microsoft Access as the database type from the first submenu, and choose Version 7.0 MDB from the second submenu. The Select Microsoft Access Database To Create dialog box appears.
4. Select a location in which to save the database, type *BDAY* in the File Name text box to name the database, and click Save. The filename extension MDB is automatically added to the filename. The Database and SQL Statement windows appear. At this point, the database exists but it contains no table or field definitions.
5. Right-click in the Database window, and choose New Table from the pop-up menu to add a new table. The Table Structure dialog box appears.
6. Type *Birthdays* in the Table Name text box. This is the name of the database's one and only table.
7. Click the Add Field button to display the Add Field dialog box. Type *Name* in the Name text box to define the first field of the table.
8. The Type drop-down list already displays *Text*, so you don't need to change the setting in this case.
9. Type 30 in the Size text box to define the maximum number of characters that can be contained in the Name field.
10. Click the OK button to add the field definition to the list of fields in the current table. You can now enter the definitions for the second field.
11. Type *Birthday* in the Name text box, select Date/Time from the Type drop-down list, and click the OK button to define the second field and to add the field definition to the list of fields.
12. Click the Close button to close the Add Field dialog box. Figure 232 shows the resulting Table Structure dialog box.
13. Click the Build The Table button to complete the process, and close the Birthdays table. Figure 23-3 shows the BDAY database.
14. Exit from the Visual Data Manager to return to Visual Basic.

We have now properly defined the database, although it contains no valid records yet.

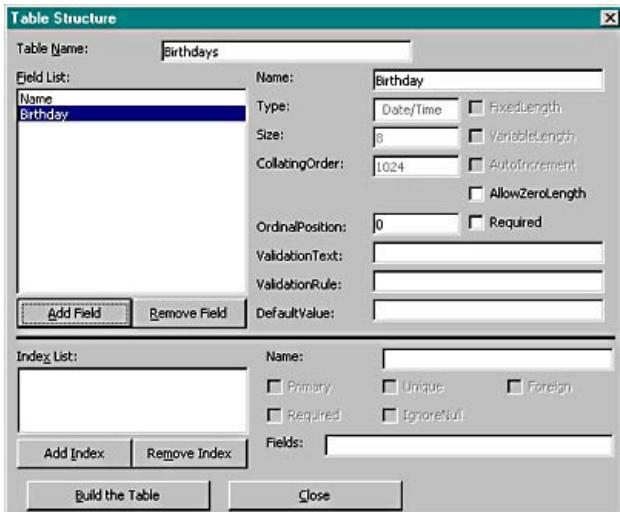


Figure 23-2. Adding fields to the Birthdays table.

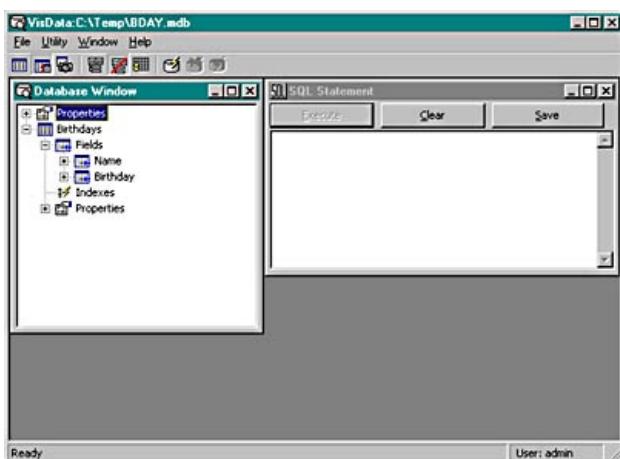


Figure 23-3. The BDAY database created with the Visual Data Manager.

Creating the User Interface

Our database now needs a user interface. Start with a new Standard EXE project, and draw three controls on Form1: a text box named *txtName*, a second text box named *txtBirthday*, and a Data control named *datBDay*. You can add optional labels if you want, as I've done in this example, but they aren't required. The completed form, containing sample data, is shown in Figure 23-4.



Figure 23-4. A simple birthday database application that uses a Data control.

The key to using the Data control and associated data-bound controls is to set the properties correctly. Always start with the Data control by setting its *DatabaseName* property to the name of the database file to be connected. In this case, set *DatabaseName* to *BDAY.MDB*, the file we just created with the Visual Data Manager. Set the Data control's *RecordSource* property to *Birthdays*, the name of the table in the database that this Data control is to reference. Generally speaking, one Data control is associated with one table in a database.

In this example, we have one other property to set on the Data control. We need to be able to add new records when the application runs. Set the *EOFAction* property to *2 - Add New*. Now when the user

scrolls to the blank record at the end of the database and fills in a new name and birth date, the database is automatically updated to contain the new record.

The Data control doesn't do much by itself. What we need to do is to associate other controls with the Data control. When they are associated, or data-bound, the controls become tools for working with the database. In our example, we use the DataSource and DataField properties of the text boxes to make these controls data-bound to the *datBDay* Data control. Set the DataSource property of each text box to *datBDay*, and set the DataField property of each text box to the name of the field with which it is to be associated. Set *txtName*'s DataField to *Name* and *txtBirthday*'s DataField to *Birthday*. These database-related properties provide a drop-down list of available tables and fields after the Data control connection has been made, making it easier to set these properties.

Running the Application

That's it! You don't need to write any code to work with the database. Run the application, enter a new name and birth date, and click the Data control's next record scrollbar arrow. You'll probably want to add some code and modify some of the properties to customize the application.

This simple example doesn't let you delete records or search for specific names or dates in the database. To turn this into a real application, you'd need to add code to provide capabilities along these lines.

Dear John, How Do I... Use Data Access Objects to Connect An Application to a Database?

Objects are everywhere in Visual Basic, and one of the areas where they've enhanced the language most is data access. Let's use these objects, without touching a Data control at all, to build another simple database application. But first let's chart out the data access objects in Visual Basic.

Using Data Access Objects

One of the best ways to understand the organization of objects in Visual Basic is to study a chart that shows the hierarchy of these objects, called an object model. The hierarchy of data access objects is shown in Figure 23-5. As I learned about data access objects, this chart helped me see where collections and objects fit into the scheme of things, and with its help I was able to get the syntax down pat for accessing these nested objects.

I've added s to object names that can also be names of collections of objects. For example, there's a Workspace object, and then there's the Workspaces collection, of which Workspaces(0) is the first Workspace object. All object names follow this scheme: the pluralizing s is added to the name of a type of object in order to name the collection. Notice in Figure 23-5 that the only data access object that is not part of a collection is DBEngine. There's only one DBEngine object.

The sample code presented in this section provides examples that declare and use these objects.

Compare the notation I've used to access these objects with the structure shown in Figure 23-5. If any of this notation seems confusing, be sure to search the Visual Basic online help for these object names and study the examples and explanations you'll find there.

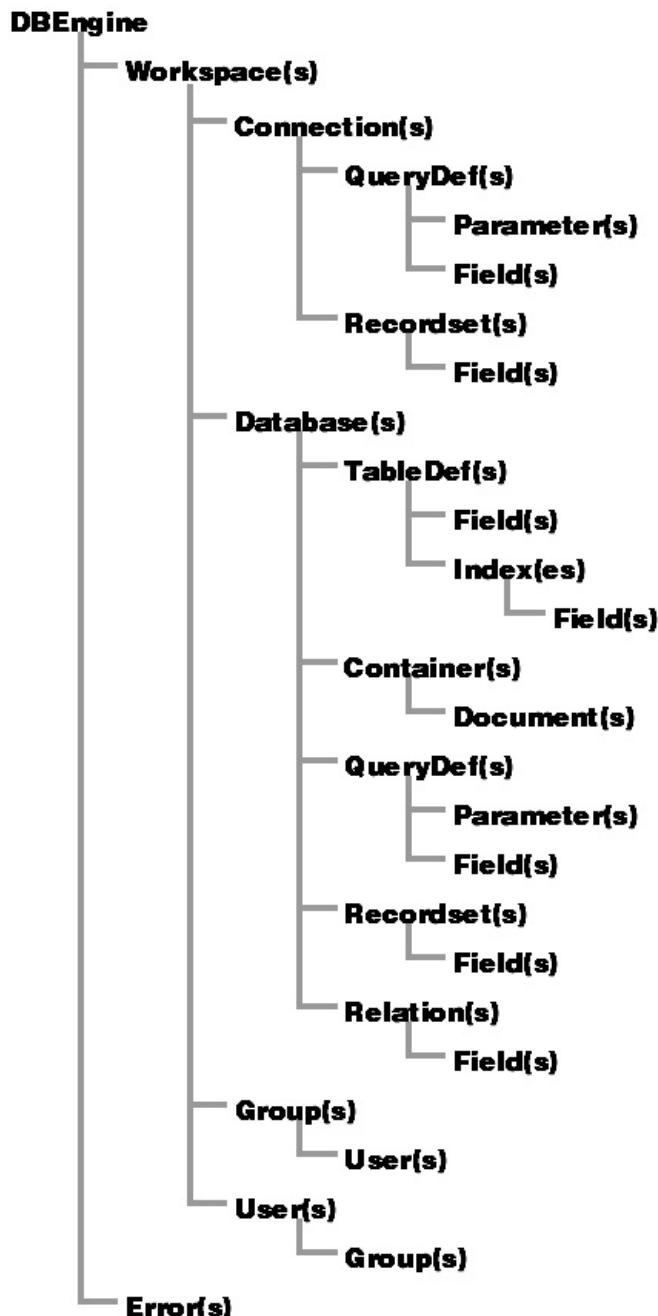


Figure 23-5. The structure of data access objects in Visual Basic.

Creating a Database Using Data Access Objects

The following short program creates a telephone number database from scratch, using only data access objects (no Data control). I've made this database similar to the birthday database we created earlier so that you can easily compare the two. In this case, two text fields are created: one for a name and one for its associated telephone number (or any other text you want to enter, such as an e-mail address).

To create this example, start a new Standard EXE project and add one command button named *cmdCreate* to Form1. Set the button's Caption property to *Create*. Be sure that you have included the DAO library as a reference or you will get an error. If the DAO library is not referenced, choose References from the Project menu and check the Microsoft DAO 3.51 Object Library check box. Add the following code to the form. Be sure to change the database path and filename as desired if you don't want the new database to be saved in the root directory of your C drive.

```
Option Explicit
```

```
Private Sub cmdCreate_Click()
  'Create data access object variables
```

```

Dim dbWorkspace As Workspace
Dim dbDatabase As Database
Dim dbTableDef As TableDef
Dim dbName As Field
Dim dbNumber As Field
`Create data access objects
Set dbWorkspace = DBEngine.Workspaces(0)
Set dbDatabase = dbWorkspace.CreateDatabase( _
    "C:\PHONES.MDB", dbLangGeneral)
Set dbTableDef = dbDatabase.CreateTableDef("Phones")
Set dbName = dbTableDef.CreateField("Name", dbText)
Set dbNumber = dbTableDef.CreateField("Number", dbText)
`Set field properties
dbName.Size = 30
dbNumber.Size = 30
`Append each field object to its table object
dbTableDef.Fields.Append dbName
dbTableDef.Fields.Append dbNumber
`Append each table to its database
dbDatabase.TableDefs.Append dbTableDef
`Close completed database
dbDatabase.Close
MsgBox "Database Created", , "Create Phones"
End Sub

```

Figure 23-6 shows this short program in action. All you have to do is click the Create button once to create the new database. I've put a lot of comments in the code to describe the action, but the overall design of the routine is as follows: In the first half of the preceding code, all object variables are declared and then objects are created and set to be referenced by these variable names. The rest of the code deals with these new objects and their properties and methods. The Size property of the two text fields is set, and then the structure of the new database is completed by linking the various objects to one another using each object's Append method.

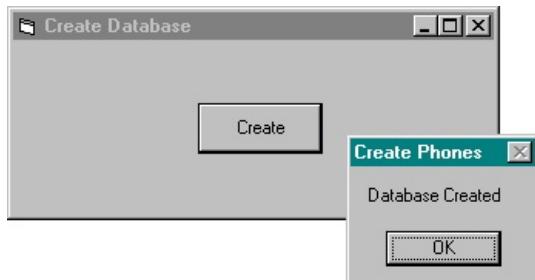


Figure 23-6. Creating a database with the click of a button.

Note that although the new telephone number database contains no valid data, the internal structure is complete and ready to go. We've created one table, Phones, and two text fields in this table, Name and Number. These two text fields are fixed to a maximum length of 30 characters.

NOTE

If, in the course of your experimentation, you want to start all over with the telephone number database, simply delete the PHONES.MDB file and rerun this program.

Accessing the Database

Now let's create a simple program to access the telephone number database, again without using a Data control. I've kept this demonstration simple: the following application lets you move forward and backward through the table's records, append new records, and type any text (up to 30 characters) to change the records as desired. To enhance this application, you'd probably want to add code to delete records,

search and sort, and so on.

Start with a new Standard EXE project, and add two command buttons to Form1. Name the buttons *cmdPrevious* and *cmdNext*, and set their Caption properties to *Previous* and *Next*. Then add two text boxes, and name them *txtName* and *txtNumber*. Figure 23-7 shows how I placed and sized these controls.



Figure 23-7. The simple telephone number database program in action.

The following code completes the demonstration. Add it to your form, and when you run it you'll have a working, albeit simple, database editor. Check that the pathname and filename of the database match the location of the telephone number database we created in the previous short program and that your project references the Microsoft DAO 3.51 Object Library. If the DAO library is not referenced, choose References from the Project menu and check the Microsoft DAO 3.51 Object Library check box.

```
Option Explicit
```

```
'Create new data access object variables
Private dbWorkspace As Workspace
Private dbDatabase As Database
Private dbTable As Recordset
Private dbName As Field
Private dbNumber As Field

Private Sub cmdNext_Click()
    'Move only if current record isn't a blank one
    If txtName.Text <> " " And txtNumber.Text <> " " Then
        'Save any changes to current record
        UpdateRecord
        'Move to next record
        dbTable.MoveNext
        'Prepare new record if at end of table
        If dbTable.EOF Then NewRecord
        'Display data from record
        DisplayFields
    End If
    'Keep focus on text boxes
    txtName.SetFocus
End Sub

Private Sub cmdPrevious_Click()
    'Save any changes to current record
    UpdateRecord
    'Step back one record
    dbTable.MovePrevious
    'Don't go past first record
    If dbTable.BOF Then dbTable.MoveNext
    'Display data from record
    DisplayFields
    'Keep focus on text boxes
    txtName.SetFocus
End Sub

Private Sub Form_Load()
    'Create data access objects
    Set dbWorkspace = DBEngine.Workspaces(0)
```

```
'Change database path if necessary
Set dbDatabase = dbWorkspace.OpenDatabase("C:\PHONES.MDB")
Set dbTable = dbDatabase.OpenRecordset("Phones", dbOpenTable)
`Use special handling if new database
If dbTable.BOF And dbTable.EOF Then NewRecord
`Start on first record
dbTable.MoveFirst
`Display data from record
DisplayFields
End Sub

Private Sub NewRecord()
`Add new record
dbTable.AddNew
`Install a space in each field
dbTable!Name = " "
dbTable!Number = " "
`Update database
dbTable.Update
`Move to new record
dbTable.MoveLast
End Sub

Private Sub UpdateRecord()
`Prepare table for editing
dbTable.Edit
`Copy text box contents into record fields
dbTable!Name = txtName.Text
dbTable!Number = txtNumber.Text
`Update database
dbTable.Update
End Sub

Private Sub DisplayFields()
`Display fields in text boxes
txtName.Text = dbTable!Name
txtNumber.Text = dbTable!Number
End Sub
```

To make things simple, I have set up this code so that it automatically updates the current record whenever you click the Previous or Next button. Notice that because we're not using data-bound text controls, it is entirely up to the program's code to coordinate what appears in the text boxes with what is in the fields of the current record. This takes a little more coding than a program that uses a Data control would require, but it's a lot more flexible.

For example, I've chosen to update the current record as the user moves to the next record or to the previous record. But it would be easy to update the record whenever any change occurs in either text box or to add an Update button and change the record if and only if this button is clicked. Data access object programming is much more flexible than using Data controls, although it requires a little more effort up front.

Dear John, How Do I... Create a Report?

Before version 6 of Visual Basic, the answer to this question had always been to use some third-party report generation package. Now, however, Visual Basic provides an excellent new technique for creating reports. When you start a new project, one of the project types you can choose is Data Project. This option creates three objects that constitute your new project: a DataEnvironment object that provides connections, queries, and other database interface elements; a DataReport object that provides an interactive way to create and edit the appearance of a printed report; and a standard form to interface with the user.

The DataReport object displays header, footer, and detail sections to which any of six special purpose controls can be added to define the report. These controls are similar to the standard Label, TextBox, Image, Shape, and Line controls used on standard forms, but they are specially designed just for the DataReport container. The Function control is the only one that differs substantially from those used on standard forms: it provides a way to calculate sums and perform other math functions on data displayed on the report.

Many different properties provide extensive control over the appearance and behavior of the DataReport and the controls it contains.

Chapter Twenty-Four

ActiveX Objects in Other Applications

ActiveX is an important part of Microsoft's vision for Windows and the Internet. Central to the overall ActiveX picture is Automation, the technology that enables objects to be easily shared between applications. More and more applications will begin to provide Automation objects to the external world, and Visual Basic will be able to immediately take advantage of those objects. As shown in Chapter 5, "[Object-Oriented Programming](#)," Visual Basic lets you create class modules that define ActiveX objects for use by external applications. The inclusion of Automation capabilities makes Visual Basic a logical choice for software development in Windows.

This chapter presents a sampling of common and useful tasks that are simplified by the Automation technology provided in Word and Excel.

Dear John, How Do I... Use ActiveX to Perform Spell Checking?

All of the Microsoft Office 97 applications provide ActiveX objects you can use in Visual Basic. Word and Excel, the two premier applications in Office, are of particular interest. Each contains an extensive library of objects that programmers can easily tap into. Let's see how we can use Automation from Visual Basic to access the spell checker available in each of these applications and return the corrected spelling to our Visual Basic application.

Microsoft Word Spell Checking

The following code creates a Word Document object and manipulates a few of the object's properties and methods to perform spell checking on a string from within a Visual Basic application. The Document object lets you create and control a document in Word. I've added many comments to the code to help you follow the action, but here's the basic scenario: The Document object is created, the text is inserted into it, and the spell checker engine is activated for this document. The results are then copied back to the text box, where they replace the original text. (This code is designed to work with Microsoft Word 97.)

```
Option Explicit
```

```
Dim mdocSpell As New Document
Dim mblnVisible As Boolean

Private Sub Form_Load()
    'Check whether application is visible;
    'used in Unload to determine whether this
    'application started Word
    mblnVisible = mdocSpell.Application.Visible
End Sub

'Check spelling of text box contents
Private Sub cmdSpell_Click()
    'Add text to a Word Range object
    mdocSpell.Range.Text = txtSpell
    'IMPORTANT: You must perform the following two steps
    'before using the CheckSpelling method!!
    'Be sure that Word is visible
    mdocSpell.Application.Visible = True
    'Activate Word
    AppActivate mdocSpell.Application.Caption
    'Check spelling
    mdocSpell.Range.CheckSpelling
    'Update text box with changes from Word
    txtSpell = mdocSpell.Range.Text
    'Trim off null character that Word adds
    txtSpell = Left(txtSpell, Len(txtSpell) - 1)
    'Activate this application
    AppActivate Caption
End Sub

'Clean up
Private Sub Form_Unload(Cancel As Integer)
    'Check whether this application started Word
    If mblnVisible Then
        'Close document
        mdocSpell.Close savechanges:=False
    Else
        'Shut down Word
        mdocSpell.Application.Quit savechanges:=False
    End If
End Sub
```

This code switches activation back and forth between Word and the sample application. The CheckSpelling method is modal, so the Visual Basic code does not continue executing until the user has

closed the spell checker. Be sure to make Word visible before calling any modal method; otherwise, the user won't be able to switch to Word, and your application will appear to hang.

To try this example, create a new Visual Basic application. Establish a reference to the Microsoft Word 8.0 Object Library. Add a command button named *cmdSpell* and a text box named *txtSpell*, and name the form *frmSpell*. Add the preceding code to the form module, and run the application. Figure 24-1 shows some misspelled text just before the Check Spelling button is clicked, and Figure 24-2, below, shows Word's spell checker dialog box in action. Figure 24-3 shows the results of running Word's spell checker on the text.

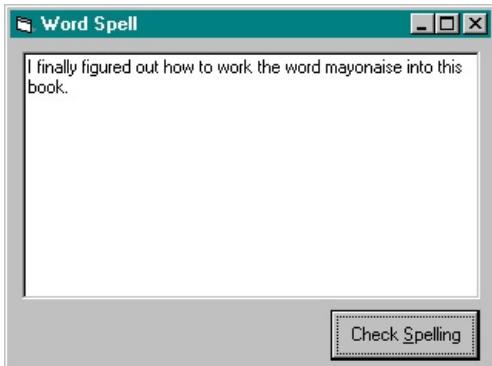


Figure 24-1. Text as it appears before Word's spell checker is invoked.

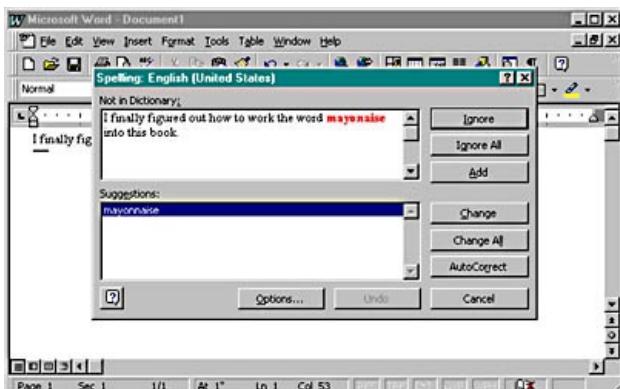


Figure 24-2. Changing the spelling of the text using Word's spell checker.

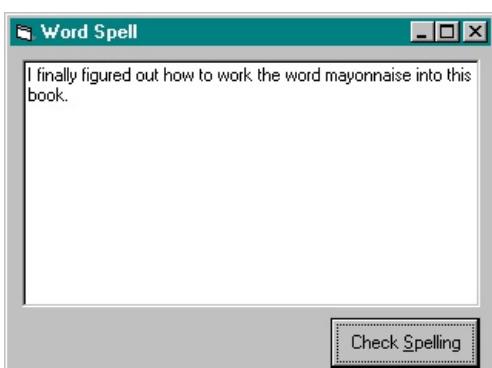


Figure 24-3. Results of invoking Word's spell checker.

The Word spell checker is also available through the global method *GetSpellingSuggestions*. Use the *GetSpellingSuggestions* method to check the spelling of a single word. The following code displays a list of suggestions when the user selects a word and presses the F7 key:

```
Const KEY_F7 = 118
`Check spelling of a single word

Private Sub txtSpell_KeyDown(KeyCode As Integer, Shift As Integer)
    Dim Corrections
    If KeyCode = KEY_F7 Then
```

```

If txtSpell.SelLength = 0 Then
    `Select the word
    SendKeys "+^{Right}"
End If
`Check spelling of selection
Set Corrections = GetSpellingSuggestions(txtSpell.SelText)
`If misspelled, display suggestions
If Corrections.Count Then
    frmCorrections.Display Corrections
End If
End If
End Sub

`Called by frmCorrections to replace text
Friend Sub Replace(Word As String)
    txtSpell.SelText = Word
End Sub

```

The preceding code displays the collection of spelling suggestions using a second form named *frmCorrections*. This form contains a list box, named *lstCorrections*, and two command buttons, named *cmdReplace* and *cmdCancel*, that allow the user to select the correction or cancel the operation. Here is the code for *frmCorrection*:

```

`Called by frmSpell to display suggestions from Word
Friend Sub Display(Corrections)
    Dim Word
    For Each Word In Corrections
        lstCorrections.AddItem Word
    Next Word
    `Select first suggestion
    lstCorrections.Selected(0) = True
    `Display the form
    Show vbModal
End Sub

`Replace word with selection
Private Sub cmdReplace_Click()
    frmSpell.Replace lstCorrections.List(lstCorrections.ListIndex)
    Unload Me
End Sub

`Cancel correction
Private Sub cmdCancel_Click()
    Unload Me
End Sub

```

Figure 24-4 shows the *GetSpellingSuggestions* method in action, after the misspelled word *vehements* is selected and the F7 key is pressed. You may notice that the *GetSpellingSuggestions* method executes faster than the *CheckSpelling* method and that both methods execute much faster if Word is already running. Speed of execution can be an important factor when you are designing applications that make use of ActiveX objects—most objects require the entire application to be running; others may run independently, loading only a few modules of an application.

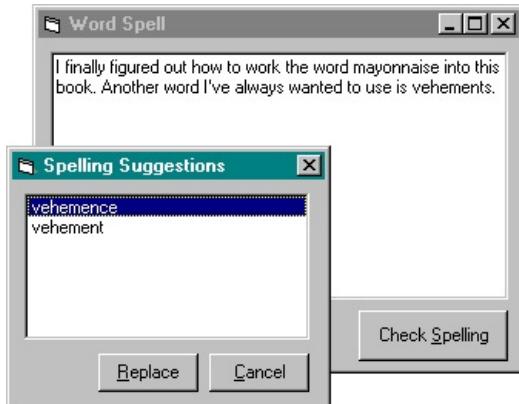


Figure 24-4. Microsoft Word's *GetSpellingSuggestions* method in action.

Microsoft Excel Spell Checking

Programming through Automation to access the spell checker engine in Excel is similar to the Word example. The code for this example accomplishes the same results, as shown in Figure 24-5 and Figure 24-6. (This code is designed to work with Excel 97.)

In this code, an Excel Sheet object is created, the Visual Basic text string is loaded into a single cell of the Sheet object, Excel's spell checker is invoked, and the cell's contents are copied back to the Visual Basic text box:

```
Option Explicit

Dim objXL As Object

Private Sub cmdSpell_Click()
    'Use Automation to create an Excel object
    Set objXL = CreateObject("Excel.Sheet")
    'Get programmable reference (apparently a bug in Excel)
    Set objXL = objXL.Application.ActiveWorkbook.ActiveSheet
    'Copy from text box to sheet
    objXL.Range("A1").Value = txtSpell.Text
    'Check spelling
    objXL.CheckSpelling
    'Copy back to text box
    txtSpell.Text = objXL.Range("A1").Value
    'Remove object from memory
    Set objXL = Nothing
    'Be sure that this application is active
    AppActivate Caption
End Sub
```

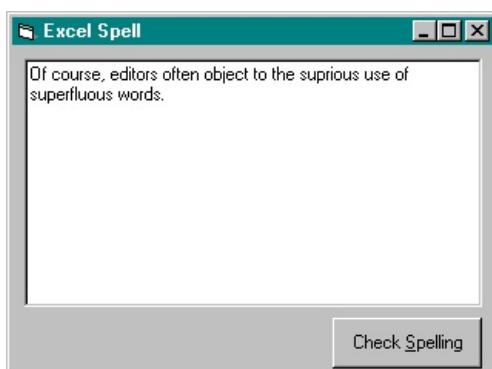


Figure 24-5. Text as it appears before Excel's spell checker is invoked.

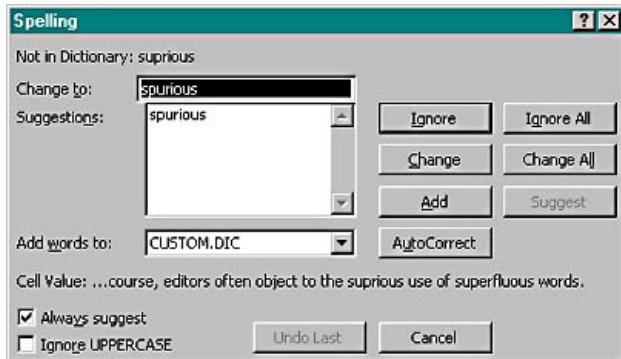


Figure 24-6. Results of invoking Excel's spell checker.

To run this code, create a new form containing a command button named *cmdSpell* and a text box named *txtSpell*. Add the code to the form, and give it a try.

Notice that the AppActivate function is being used. If AppActivate was not used and this program was run with Excel already loaded, Excel would stay active after the spell checking was completed. To prevent this problem, the Visual Basic AppActivate function was used to reactivate your application when you finished using Excel's spell checker.

Also notice the following line of code:

```
Set objXL = objXL.Application.ActiveWorkbook.ActiveSheet
```

Normally, you would not need to include this line, but due to an apparent bug in Excel 97, this line of code is necessary.

Early vs. Late Binding

You can use either early or late binding when working with ActiveX objects.

Early binding lets you use specific types for your ActiveX objects. This enables Visual Basic's command-completion feature and enables the Object Browser for the referenced ActiveX object library. Early binding also writes the ActiveX object's class ID into your application's executable file. This results in faster object access. Early binding is demonstrated in the Word spelling example, which uses the Visual Basic References dialog box to establish a reference to the Word object library.

Late binding uses a generic Object data type for ActiveX object variables and then creates a reference to the specific ActiveX object at runtime. The ActiveX application's class ID is not written into your application's executable file, and object access takes a little longer. Late binding is demonstrated in the Excel spell checking example, which uses the CreateObject method to establish a reference to the Excel object library.

Whether or not the class ID is written into your application's executable file is very important because class IDs are likely to change with each new version of a product. If a new version of Word or Excel is released, applications that use their objects through early binding must be revised and redistributed. Applications that use late binding may not need to be redistributed, although you should test them to be sure that changes to Word or Excel don't adversely affect your application.

Dear John, How Do I... Use ActiveX to Count Words?

Let's take a look at one more example of Automation with Word, this time to count the words in a string. In the following code, access to Word is achieved through the Document object, as was shown in the spell checking example. A second object variable, *dlg*, is created to access and hold the settings of Word's Document Properties dialog box.

```
Private Sub cmdWords_Click()
    Dim dlg As Word.Dialog
    'Copy text into Word document
    mdocSpell.Range = txtSpell.Text
    'Create an object to hold dialog box settings
    Set dlg = mdocSpell.Application.Dialogs _
        (wdDialogDocumentStatistics)
    'Count words and characters
    dlg.Execute
    'Display results in a Label control
    Caption = Str(dlg.Words) & " words, " _
        & Str(dlg.Characters) & " characters"
End Sub
```

In this example, I've grabbed the number of words and the number of characters from the object holding the dialog box settings, but there are other properties to access if you want. Search Word's Visual Basic online help for the Dialogs collection to learn more about the contents of this and other dialog boxes.

Figure 24-7 shows the word and character count for a small sample of text.

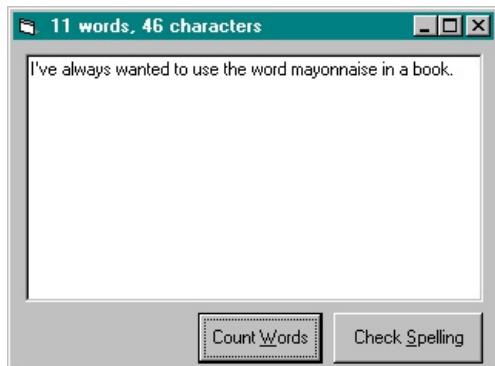


Figure 24-7. Counting words and characters using Word's Word Count feature.

To run this example, start with the form created in the section "[Microsoft Word Spell Checking](#)," earlier in this chapter, and add a command button named *cmdWords*. At runtime, type several words of text into the text box, and then click the command button.

Dear John, How Do I... Use Microsoft Excel's Advanced Math Functions?

Excel contains a number of math functions that are not available in Visual Basic. The following procedure uses Automation to create an Excel Sheet object and then access one of Excel's functions to perform a calculation on a value passed to the object from the Visual Basic program. More important, this example shows how you can use Excel to create bigger and better Visual Basic programs.

In this example, I use two cells in the Excel Sheet object for input and two other cells for reading back the results of the calculation performed on the values placed in the first two cells. Excel's ATAN2 function is used to calculate the angle between the positive x-axis and a line drawn from the origin to the point represented by these two data values. The angle is returned in radians, which can easily be converted to degrees. Unlike Visual Basic's Atn function, which uses the ratio of two sides of a triangle to return a value of -pi/2 to pi/2 radians, Excel's ATAN2 function uses the coordinates of the point to return a value between -pi and pi radians. In this way, the return value of ATAN2 is correctly calculated for the quadrant that the given point is in.

The following code creates an Excel Sheet object, places the coordinates for the point into two cells of the Excel Sheet object, has Excel calculate the arctangent using the ATAN2 function, and returns the value to the Visual Basic application:

```
Option Explicit
```

```
Private Sub cmdAngle_Click()
    Dim objXL As Object
    'Use Automation to create an Excel object
    Set objXL = CreateObject("Excel.Sheet")
    'This solves the Excel bug mentioned earlier
    Set objXL = objXL.Application.ActiveWorkbook.ActiveSheet
    'Set known values in cells
    objXL.Range("A1").Value = txtX.Text
    objXL.Range("A2").Value = txtY.Text
    'Calculate third cell
    objXL.Range("A3").Formula = "=ATAN2(A1,A2)"
    objXL.Range("A4").Formula = "=A3*180/PI()"
    'Display results in Label controls
    lblRadians.Caption = objXL.Range("A3").Value
    lblDegrees.Caption = objXL.Range("A4").Value
    'Remove object from memory
    Set objXL = Nothing
End Sub
```

To try this example, create a new form and add to it two text boxes, named *txtX* and *txtY*; two labels, named *lblRadians* and *lblDegrees*; and a command button, named *cmdAngle*. As shown in Figure 24-8, an xy-coordinate pair entered into the text boxes is processed when the Angle button is clicked to calculate the angle from the x-axis to the given point. I've set up two calculations, the first to find the angle in radians and the second to convert this measurement to degrees. The results of these two Excel calculations are copied back to the labels for display. I've also added labels to the form to clarify the identity of the input text boxes and output labels.

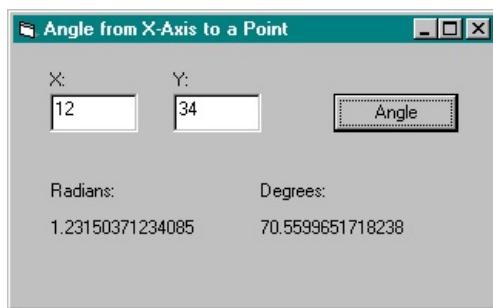


Figure 24-8. An advanced Excel math function called from Visual Basic.

This technique of using an Excel Sheet object can be adapted to work with varying amounts of data, types of calculations, and levels of complexity—and the final result of all the number crunching can be

copied back to your Visual Basic application. You'll find that using an Excel object for some types of data manipulations and calculations is easier and more efficient than trying to code the entire process in Visual Basic.

Chapter Twenty-Five

Screen Savers

It's actually fairly easy to create a screen saver using Visual Basic. In this chapter, I cover how to create a basic screen saver. But I also go further into the subject, showing you how to enhance the screen saver and make it work with the Windows 95 Display Properties dialog box. The enhancements include ensuring that only one instance of the program is running, turning off the mouse cursor, speeding up the graphics, detecting different user actions to terminate the screen saver, and using the current screen contents as part of a screen saver. I also look at how to use the Windows Display Properties dialog box to preview the screen saver and let the user set options.

Dear John, How Do I... Create a Screen Saver?

Screen savers were originally created to protect screens from the "burn-in" that can be caused by pixels that don't change very often. Perhaps screen savers are now used more for entertainment than for screen protection, but the underlying principle of toggling all pixel locations through a variety of colors over time is still the foundation of their design. For this reason, a screen saver should display some sort of graphics that continually move and change.

Any Visual Basic application can run as a screen saver. Of course, some programs will work better than others for this purpose. A basic screen saver is quite simple: a form that takes up the entire screen, code to create moving and changing graphics, and code to terminate the program when the user executes some action. This is all that is required to create a simple screen saver, but you will probably want to use the enhancements discussed in the following sections to improve the operation of your application.

To make your application function as a screen saver in the Windows environment, you need to compile the program as a screen saver. To do this, choose Make EXE from the File menu, and click the Options button in the Make Project dialog box. In the Application Title text box on the Make tab of the Project Properties dialog box, type SCRNSAVE: followed by the name of your screen saver. For example, if your screen saver is named MySaver, type SCRNSAVE:MySaver. Click OK to close the Project Properties dialog box, and instead of creating an executable file with the extension EXE, change the filename's extension to SCR by typing over EXE in the File Name text box. Click the OK button to finish building the executable screen saver. Copy the resulting SCR file into your Windows directory so that it can be located and its options set from the Display Properties dialog box.

NOTE

Be aware that if you give your screen saver a name starting with two uppercase S's, Windows will remove them and SSaver1.scr will be listed as "aver1" in the Display Properties dialog box. This occurs because of the way earlier versions of Windows handled screen savers.

The following listing presents a simple screen saver application. The application does not handle some of the more complex features that are normally integrated into a screen saver. To act as a proper screen saver, your application should end when the mouse is moved or clicked, or when a key is pressed. Also, it should not allow multiple instances of itself to run, and in Windows 95 it should gracefully handle the Settings, Preview, and Password Protected options that are accessed from the Screen Saver tab of the Display Properties dialog box.

In this application, the screen saver does not respond to all of the control options in the Display Properties dialog box (except the Wait setting, which is handled by the system). The screen saver ends when you press a key or click the mouse, but it does not respond to any mouse movement. Adding all these features to the application is discussed in the sections following this one.

To create this screen saver, start a new Standard EXE project, and add a standard module to accompany the single form. Name them frmMySaver1 and modMySaver1 and save them in files named MySaver1.frm and MySaver1.bas. To the form, add a Timer control named *tmrExitNotify*. Set the timer's Interval property to 1 and its Enabled property to *False*. Set the form's BorderStyle property to 0 - *None* and its WindowState property to 2 - *Maximized*, and add the following code:

```
'MySaver1.frm
Option Explicit

`Module-level variables
Dim mblnQuit As Boolean

Private Sub Form_Paint()
    Dim lngX As Long

    `Display different graphics every time
    Randomize
```

```
'Initialize graphics parameters
Scale (0, 0)-(1, 1)
BackColor = vbBlack
`Do main processing as a loop
Do
    `Update display
    DoGraphics
    `Yield execution
    DoEvents
Loop Until mblnQuit = True
`Can't quit in this context; let timer do it
tmrExitNotify.Enabled = True
End Sub

Private Sub Form_Click()
    mblnQuit = True
End Sub

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    mblnQuit = True
End Sub

Private Sub tmrExitNotify_Timer()
    End
End Sub

'~~~~~
`This is where the real graphics drawing takes place
'~~~~~

Sub DoGraphics()
    `Occasionally change line color and width
    If Rnd < 0.03 Then
        ForeColor = QBColor(Int(Rnd * 16))
        DrawWidth = Int(Rnd * 9 + 1)
    End If
    `Draw circle
    Circle (Rnd, Rnd), Rnd
End Sub
```

Add the following code to the MySaver1.bas module, and be sure to set the project's startup object to Sub Main:

```
`MySaver1.bas
Option Explicit

`Starting pointDear John, How Do I... .
Public Sub Main()
    Dim strCmd As String
    Dim strTwo As String
    `Process the command line
    strCmd = UCase(Trim(Command))
    strTwo = Left(strCmd, 2)
    Select Case strTwo
        `Show screen saver in normal full screen mode
        Case "/S"
            Load frmMySaver1
            frmMySaver1.Show
            Exit Sub
        Case Else
            Exit Sub
    End Select
End Sub
```

Compile this project to create an SCR file name MySaver1.scr, as described earlier, and place it in your Windows directory. Right-click on the desktop, select Properties, and click the Screen Saver tab of the Display Properties dialog box. Here you should be able to select MySaver1 from the drop-down list and test the screen saver by clicking the Preview button.

When activated, this screen saver draws random circles of varying thickness and color over a black background, as shown in Figure 25-1. Press any key or click the mouse to terminate the screen saver.

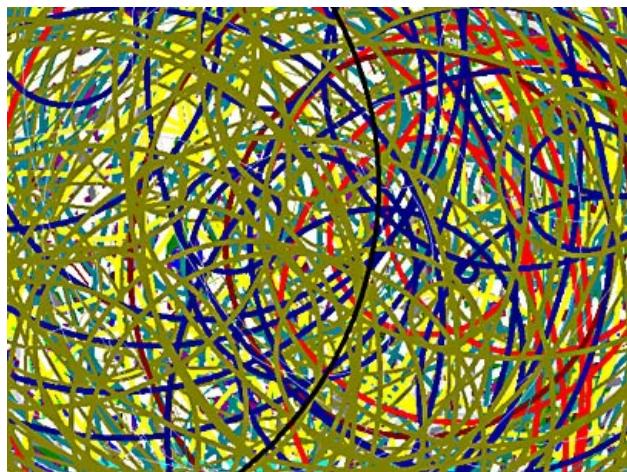


Figure 25-1. The MySaver1 screen saver program in action.

SEE ALSO

- The other sections in this chapter for details about enhancements to this screen saver
- The MySaver application in Chapter 29, "[Graphics](#)," for an example of a more complete screen saver application

Dear John, How Do I... Prevent Two Instances of a Screen Saver from Running at the Same Time?

There are two approaches to preventing multiple instances of a screen saver from running at the same time. Visual Basic provides an App object that has a `PrevInstance` property set to `True` if a previous instance of the current Visual Basic application is already running. This handy property makes it easy to bail out of an application quickly, during the `Form_Load` event procedure, to avoid complications that occur when running multiple instances of a screen saver simultaneously. The following code shows how `App.PrevInstance` is typically used in a screen saver application:

```
'Don't allow multiple instances of program
If App.PrevInstance = True Then
    Unload Me
    Exit Sub
End If
```

The second way to prevent multiple instances of a screen saver is by calling an application programming interface (API) function, which informs the system that a screen saver is active. This approach lets the operating system handle the details of starting the screen saver if, and only if, it's not already running. This is the method I've chosen for the rest of the screen saver examples in this chapter and in Chapter 26, ["Project Development."](#)

To see how this API function works, start with a new project identical to MySaver1. Rename the files `frmMySaver2` and `modMySaver2`. Edit the `modMySaver2` module's code as follows, and compile the project to create `MySaver2.scr`:

```
'MySaver2.bas
Option Explicit

`Constants for some API functions
Private Const SPI_SETSCREENSAVEACTIVE = 17

Private Declare Function SystemParametersInfo _
Lib "user32" Alias "SystemParametersInfoA" ( _
    ByVal uAction As Long, _
    ByVal uParam As Long, _
    ByRef lpvParam As Any, _
    ByVal fuWinIni As Long
) As Long

`Starting pointDear John, How Do I...
Public Sub Main()
    Dim lngRet As Long
    Dim lngParam As Long
    Dim strCmd As String
    Dim strTwo As String
    `Process the command line
    strCmd = UCASE(Trim(Command))
    strTwo = Left(strCmd, 2)
    `Tell system screen saver is active now
    lngRet = SystemParametersInfo(
        SPI_SETSCREENSAVEACTIVE, 1, lngParam, 0)
    Select Case strTwo
        `Show screen saver in normal full screen mode
        Case "/S"
            Load frmMySaver2
            frmMySaver2.Show
            Exit Sub
        Case Else
            Exit Sub
    End Select
End Sub
```

Select and preview MySaver2 to compare its behavior with that of MySaver1. If you set the Wait delay time to 1 minute you'll see MySaver1 suddenly seem to restart itself after the first minute of its runtime. MySaver2, on the other hand, will continue to run smoothly as the minutes go by.

Dear John, How Do I... Hide the Mouse Pointer in a Screen Saver?

The Windows API ShowCursor function allows you to hide or show the mouse pointer in a Visual Basic application. To hide the mouse pointer, pass *False* to ShowCursor, and to show it again, pass *True*.

In a screen saver application, I declare the ShowCursor function and call it where appropriate to temporarily hide or reshow the mouse pointer.

NOTE

In Visual Basic, we refer to the mouse *pointer*, but in Visual C++ it's called the mouse *cursor*. This is why the function name is ShowCursor instead of ShowPointer, which would be more logical to us Visual Basic types.

Here is the declaration for the API function ShowCursor:

```
Private Declare Function ShowCursor _  
Lib "user32" ( _  
    ByVal bShow As Long _  
) As Long
```

Here are two examples of the use of the ShowCursor function, one that hides and one that shows the mouse pointer:

```
`Hide mouse pointer  
x = ShowCursor(False)  
  
`Show mouse pointer  
x = ShowCursor(True)
```

Be sure to pair these functions so that the mouse pointer is properly restored at the termination of the program. To add the mouse-pointer hiding feature to your screen saver, make a new MySaver3 project by copying and renaming the files from MySaver2. Change all instances of "frmMySaver2" to "frmMySaver3" in the MySaver3.bas code, and edit the MySaver3.frm form code as follows:

```
`MySaver3.frm  
Option Explicit  
  
`API function to hide or show the mouse pointer  
Private Declare Function ShowCursor _  
Lib "user32" ( _  
    ByVal bShow As Long _  
) As Long  
  
`Module-level variables  
Dim mblnQuit As Boolean  
  
Private Sub Form_Paint()  
    Dim lngRet As Long  
    `Display different graphics every time  
    Randomize  
    `Initialize graphics parameters  
    Scale (0, 0)-(1, 1)  
    BackColor = vbBlack  
    `Hide mouse pointer  
    lngRet = ShowCursor(False)  
    `Do main processing as a loop  
    Do  
        `Update display  
        DoGraphics  
        `Yield execution
```

```
DoEvents
Loop Until mblnQuit = True
`Show mouse pointer
lngRet = ShowCursor(True)
`Can't quit in this context; let timer do it
tmrExitNotify.Enabled = True
End Sub

Private Sub Form_Click()
    mblnQuit = True
End Sub

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    mblnQuit = True
End Sub

Private Sub tmrExitNotify_Timer()
    End
End Sub

'~~~~~`This is where the real graphics drawing takes place`~~~~~
Sub DoGraphics()
    `Occasionally change line color and width
    If Rnd < 0.03 Then
        ForeColor = QBColor(Int(Rnd * 16))
        DrawWidth = Int(Rnd * 9 + 1)
    End If
    `Draw circle
    Circle (Rnd, Rnd), Rnd
End Sub
```

This time, when you preview MySaver3 you won't see the mouse pointer, although you can still click the mouse to terminate the screen saver.

Dear John, How Do I... Detect Mouse Movement or a Mouse Click to Terminate a Screen Saver?

The most obvious place to detect mouse movement is in the MouseMove event, but this presents a problem. The MouseMove event is triggered once when the application starts, even if the mouse doesn't physically move. The workaround for this problem is to detect the first MouseMove event and to stop the screen saver only if the mouse actually moves from its starting position. Here's the code that accomplishes this technique:

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    'Keep track of last known mouse position
    Static sngX As Single
    Static sngY As Single
    'On first move, simply record position
    If sngX = 0 And sngY = 0 Then
        sngX = X
        sngY = Y
    End If
    'Quit only if mouse actually changes position
    If X <> sngX Or Y <> sngY Then
        mblnQuit = True
    End If
End Sub
```

The static variables *sngX* and *sngY* keep track of the starting position of the mouse pointer. The program terminates if and only if the MouseMove event occurs and the mouse is no longer at the starting location. It's assumed that the mouse is always at a nonzero location to begin with, but even if the mouse is at location (0, 0) when the screen saver starts, the program will still terminate correctly when the mouse starts to move.

Copy the MySaver3 files to make a new MySaver4 project. Change all occurrences of "frmMySaver3" to "frmMySaver4" in the MySaver4.bas code, add the MouseMove code shown above to MySaver4.frm, and compile to create MySaver4.scr. When you preview this version of the screen saver you'll notice it terminates immediately when you nudge the mouse.

Dear John, How Do I... Detect a Keypress to Terminate a Screen Saver?

The form's KeyPress event can be used to detect keyboard activity, but this doesn't actually catch all keyboard activity. For example, pressing and releasing a shift key does not cause the form's KeyPress event to fire. Instead, I chose to watch for keyboard activity by using the form's KeyDown event. This stops the screen saver the moment any key, including one of the shift keys, is pressed:

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    mblnQuit = True
End Sub
```

Dear John, How Do I... Use an Image of the Screen as a Screen Saver?

Many nifty screen savers act on the current display without permanently affecting any running applications. Perhaps you've seen those screen savers that cause the display to melt and drip away or to swirl down a drain. By using a few Windows API functions, you can easily copy the current display into a full screen form on which you can do whatever you want to the pixels without actually affecting the "real" display.

Below is the listing for a screen saver form that demonstrates this effect. This program is similar to the MySaverN series we've been developing. The main difference is the addition of the API functions used to grab the screen. Notice the declarations at the start of the listing for the API functions BitBlt, GetDesktopWindow, GetDC, and ReleaseDC, which are all required to copy the screen onto the form. These functions are called in the Form_Load event procedure to display a copy of the screen on the main form. I've named this screen saver example MySaver5.

Follow the same procedures to compile this application to an executable file in the Windows directory with the extension SCR. Also, for proper operation, be sure to set the following properties of the form: set AutoRedraw to *True*, BorderStyle to *0 - None*, KeyPreview to *True*, and WindowState to *2 - Maximized*. Finally, add a Timer control named *tmrExitNotify* to the form and set its Enabled property to *False* and its Interval property to 1.

Here's the code for MySaver5.bas:

```
'MySaver5.bas
Option Explicit

`Constants for some API functions
Private Const SPI_SETSCREENSAVEACTIVE = 17

Private Declare Function SystemParametersInfo _
Lib "user32" Alias "SystemParametersInfoA" ( _
    ByVal uAction As Long, _
    ByVal uParam As Long, _
    ByRef lpvParam As Any, _
    ByVal fuWinIni As Long _
) As Long

`Starting pointDear John, How Do I... .
Public Sub Main()
    Dim lngRet As Long
    Dim lngParam As Long
    Dim strCmd As String
    Dim strTwo As String
    `Process the command line
    strCmd = UCASE(Trim(Command))
    strTwo = Left(strCmd, 2)
    `Tell system screen saver is active now
    lngRet = SystemParametersInfo(
        SPI_SETSCREENSAVEACTIVE, 1, lngParam, 0)
    Select Case strTwo
        `Show screen saver in normal full screen mode
        Case "/S"
            Load frmMySaver5
            frmMySaver5.Show
            Exit Sub
        Case Else
            Exit Sub
    End Select
End Sub
```

Here's the code for MySaver5.frm:

```
'MySaver5.frm
Option Explicit
```

```
'API function to hide or show the mouse pointer
Private Declare Function ShowCursor _
Lib "user32" ( _
    ByVal bShow As Long _
) As Long

`Declare API to get a copy of entire screen
Private Declare Function BitBlt _
Lib "gdi32" ( _
    ByVal hDestDC As Long,
    ByVal lnx As Long, ByVal lny As Long, _
    ByVal nWidth As Long, -
    ByVal nHeight As Long, -
    ByVal hSrcDC As Long, -
    ByVal lnxSrc As Long, ByVal lnySrc As Long, -
    ByVal dwRop As Long _
) As Long

`Declare API to get handle to screen
Private Declare Function GetDesktopWindow _
Lib "user32" () As Long

`Declare API to convert handle to device context
Private Declare Function GetDC _
Lib "user32" ( _
    ByVal hwnd As Long _
) As Long

`Declare API to release device context
Private Declare Function ReleaseDC _
Lib "user32" ( _
    ByVal hwnd As Long,
    ByVal hdc As Long _
) As Long

`Module-level variables
Dim mblnQuit As Boolean

Private Sub Form_Load()
    Dim lnx As Long
    Dim lny As Long
    Dim lnxSrc As Long
    Dim lnySrc As Long
    Dim dwRop As Long
    Dim hwndSrc As Long
    Dim hSrcDC As Long
    Dim lnxRes As Long
    Dim lnxM1 As Long
    Dim lnxM2 As Long
    Dim lnxN1 As Long
    Dim lnxN2 As Long
    Dim lnxPixelColor As Long
    Dim lnxPixelCount As Long
    Dim lnxRet As Long
    Dim intPowerOfTwo As Integer
    `Display different graphics every time
    Randomize
    `Copy entire desktop screen into picture box
    ScaleMode = vbPixels
    Move 0, 0, Screen.Width + 1, Screen.Height + 1
    dwRop = &HCC0020
    hwndSrc = GetDesktopWindow()
    hSrcDC = GetDC(hwndSrc)
```

```

    lngRes = BitBlt(hdc, 0, 0, ScaleWidth, _
                    ScaleHeight, hSrcDC, 0, 0, dwRop)
    lngRes = ReleaseDC(hwndSrc, hSrcDC)
    `Display full size
    Show
    `First time use high power of 2
    intPowerOfTwo = 128
    `Hide mouse pointer
    lngRet = ShowCursor(False)
    `Do main processing as a loop
    Do
        `Map screen into rectangular blocks
        Scale (0, 0)-(intPowerOfTwo, intPowerOfTwo)
        `Set a random solid color
        lngPixelColor = (lngPixelColor * 9 + 7) Mod 16
        lngPixelCount = 0
        `Algorithm to hit each location on screen
        lngM1 = Int(Rnd * (intPowerOfTwo \ 4)) * 4 + 1
        lngM2 = Int(Rnd * (intPowerOfTwo \ 4)) * 4 + 1
        lngN1 = Int(Rnd * (intPowerOfTwo \ 2)) * 2 + 1
        lngN2 = Int(Rnd * (intPowerOfTwo \ 2)) * 2 + 1
        Do
            `Jump to next coordinate
            lnx = (lnx * lngM1 + lngN1) Mod intPowerOfTwo

            If lnx <> 0 Then
                lny = (lny * lngM2 + lngN2) Mod intPowerOfTwo
            Else
                `Let system do its thing
                DoEvents
            End If
            `Fill rectangular block with solid color
            Line (lnx, lny)-
            (lnx + 1, lny + 1), QBColor(lngPixelColor), BF
            lngPixelCount = lngPixelCount + 1
            `Exit this loop only to quit screen saver
            If mbInQuit = True Then Exit Do
        Loop Until lngPixelCount = intPowerOfTwo * intPowerOfTwo
        intPowerOfTwo = 2 ^ (Int(Rnd * 5) + 2)
        `Yield execution
        DoEvents
    Loop Until mbInQuit = True
    `Show mouse pointer
    lngRet = ShowCursor(True)
    `Can't quit in this context; let timer do it
    tmrExitNotify.Enabled = True
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
                           X As Single, Y As Single)
    Dim sngXnow As Single
    Dim sngYnow As Single
    sngXnow = Round(X)
    sngYnow = Round(Y)
    `Keep track of last known mouse position
    Static sngX As Single
    Static sngY As Single
    `On first move, simply record position
    If sngX = 0 And sngY = 0 Then
        sngX = sngXnow
        sngY = sngYnow
    End If
    `Quit only if mouse actually changes position
    If sngXnow <> sngX Or sngYnow <> sngY Then

```

```
    mblnQuit = True
End If
End Sub

Private Sub Form_Click()
    mblnQuit = True
End Sub

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    mblnQuit = True
End Sub

Private Sub tmrExitNotify_Timer()
    End
End Sub
```

The graphics update loop in the Form_Load event procedure draws solid rectangular blocks of color, causing the display to gradually dissolve to the given color. Figure 25-2 shows the effect, with the original display dissolving to a solid gray.

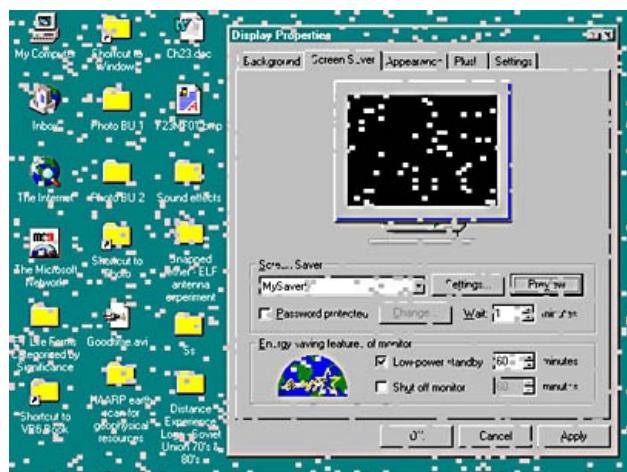


Figure 25-2. The dissolving-display screen saver in action.

SEE ALSO

- The MySaver application in Chapter 29, "[Graphics](#)," for additional creative screen saver effects

Dear John, How Do I... Add Password and Setup Capabilities to a Screen Saver?

Windows automatically passes a command line parameter to the screen saver when it starts it up, based on how, and in what mode, the program is started. These command parameters and their descriptions are as follows:

/a	The Change password button has been clicked in the Display Properties dialog box.
/p nnnn	A preview is shown in the Display Properties dialog box whenever the Screen Saver tab is activated. The number on the command line is the hWnd for the small picture of the desktop where the preview is to be displayed.
/c	The Settings button has been clicked in the Display Properties dialog box.
/s	The Preview button has been clicked in the Display Properties dialog box, or the application was started normally by the system.

I check for these parameters in the Form_Load event procedure and take appropriate action based on the way the screen saver was started. In the sample screen savers already presented in this chapter, I check only for the /s command parameter and terminate the program whenever any other parameter is used.

This could be approached in a more user-friendly way by looking for some of the other command parameters and showing a message about the options, as shown here:

```
'Configuration command
Case "/C"
    `Temporarily show mouse pointer
    x = ShowCursor(True)
    `Perform any user interaction
    MsgBox "No setup options for this screen saver"
    `Hide mouse pointer
    x = ShowCursor(False)
    `Configuration is completed
    Unload Me
    Exit Sub
`Password setting command
Case "/A"
    `Temporarily show mouse pointer
    x = ShowCursor(True)
    `Get and record new password here
    MsgBox "No password for this screen saver"
    `Hide mouse pointer
    x = ShowCursor(False)
    `Setting of new password is completed
    Unload Me
    Exit Sub
```

This provides a general example of how to process the command line parameters. For a complete working example of the use of all parameters (with the exception of the password command), skip ahead to the MySaver application in Chapter 29, "[Graphics](#)." The MySaver screen saver adds to the code presented in this chapter to complete the major parts of a standard screen saver, including the preview display on the miniature desktop within the image of the monitor.

SEE ALSO

- The MySaver application in Chapter 29, "[Graphics](#)," for an example of a more complete screen saver application

Chapter Twenty-Six

Project Development

This chapter presents a few tricks I've picked up along the way that are helpful in overall project development. The first section shows you a simple but often overlooked technique for grabbing images of your running application so that they can be added to your help file. (Help files, which are an important part of any complete application, are covered more fully in Chapter 17, "[User Assistance](#).")

The second and third sections in this chapter explain how to use resource files and string databases to internationalize applications. These techniques will give you a great start on building applications for users around the world.

Dear John, How Do I... Grab a Running Form and Save It as a Bitmap?

Here's a simple technique for grabbing all or part of the display to convert it to a bitmap (BMP) file, a technique you might already know: simply press the Print Screen key to copy the entire screen to the clipboard. You can do this at any time while Windows is running. (If you want to copy only the form that currently has the focus, press Alt-Print Screen.) You won't hear a beep or notice anything other than perhaps a slight delay in the current program's operation, but the screen's contents (or the form's contents) will quickly be copied to the system's clipboard. Most of the forms shown in this book were grabbed using the Alt-Print Screen technique.

Pasting the Graphic into Paint

To process the clipboard's graphics contents, run the Windows Paint application, and paste the clipboard's contents into it. You should see the upper-left corner of the captured display in Paint's workspace, as shown in Figure 26-1. With Paint, you can scroll to any part of the image, and you can use its tools to make editing changes.

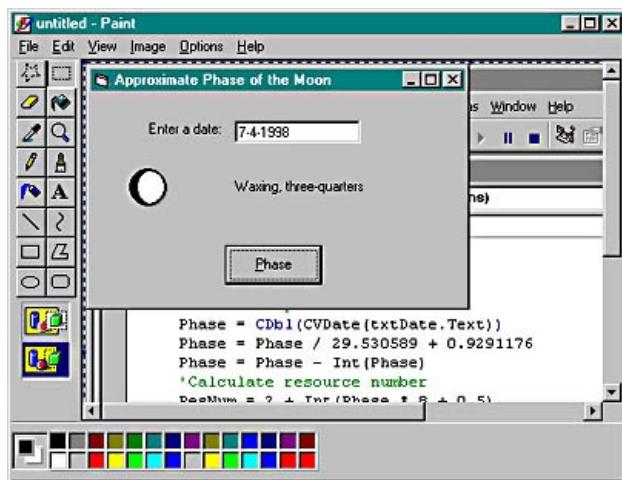


Figure 26-1. A portion of a captured display, as shown in Paint.

Using Save As and Copy To

Choose Save As from the File menu to save the image as a BMP file. To save a smaller region of the image, select part of the image using the rectangle selection tool, and then choose Copy To from Paint's Edit menu. This menu option prompts for a filename. Unlike the Save As option, the Copy To option saves only the area of the image within the selection rectangle. Figure 26-2 shows the selection rectangle delineating a small part of the entire original screen image, and Figure 26-3 shows this small image after it has been reloaded into Paint from the BMP file to which it was written.

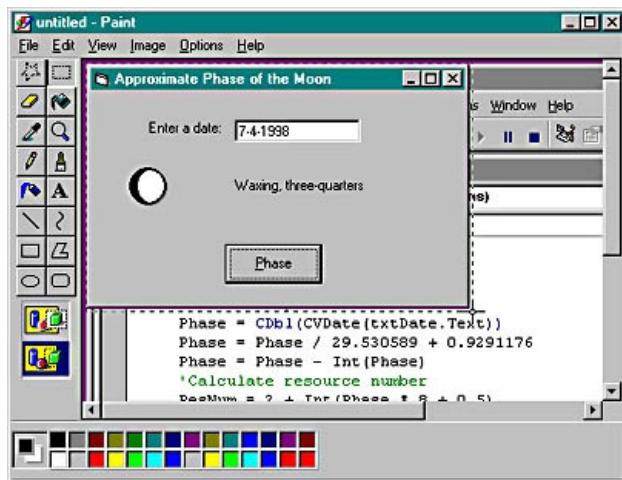


Figure 26-2. A selection rectangle delineating a portion of a larger image.

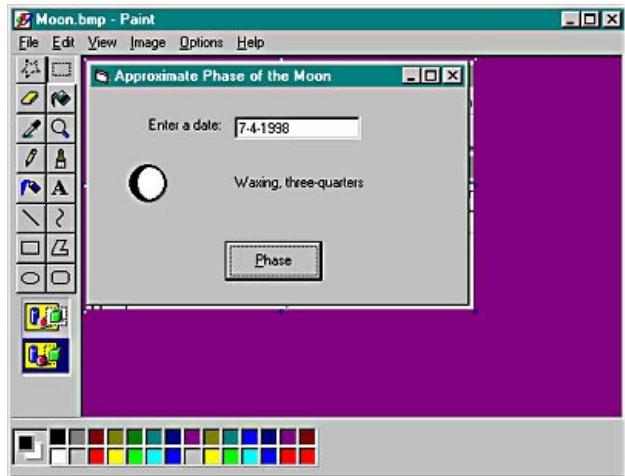


Figure 26-3. The selected image loaded into Paint.

Dear John, How Do I... Use Resource Files?

Resource file capability was added to Visual Basic primarily to make it easier to internationalize applications. Before resource files were available, you had to edit all the captions, labels, and other strings for an application within the Visual Basic project itself if you wanted your application to appear in another language. (I'm referring to spoken languages here, not computer programming languages.) But now you can isolate all of your application's strings in a resource file, making it much easier to edit the strings for a second language. I've also found another useful feature of resource files: they allow you to easily include numerous graphics, bitmaps, and icon files directly in your project. These images can be loaded and manipulated from resource files much faster than from individual external files.

You can load one, and only one, resource file into your Visual Basic project. But you can stuff a lot of strings, icons, bitmaps, sound files, and even video clips into a single resource file. Below, I demonstrate the process of building a small resource file and then provide an example application that uses its contents.

Creating a Resource File

The Resource Editor Add-In provides a graphic user-interface for creating resource files. Use the Add-In Manager to load the Resource Editor in Visual Basic, and then click the Resource Editor toolbar button to start the editor, shown in Figure 26-4.

When you save a resource file, the Resource Editor automatically adds it to your project as an RES file. There are two points you should remember when working with resource files in Visual Basic:

1. Don't use the resource number 1 because Visual Basic reserves this resource number for your application's internally stored icon.
2. RES files are in binary format. To create a resource file using a text editor, create an RC file and compile it using RC.EXE as described in RESOURCE.TXT in the COMMON\TOOLS\VB\RESOURCE directory of the Visual Studio CD-ROM.

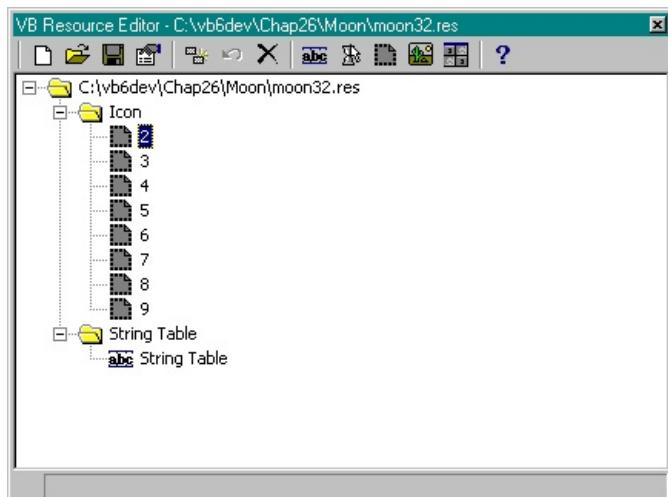


Figure 26-4. The Resource Editor Add-In lets you create new resource files within Visual Basic.

Using a Resource File in Your Application

Figure 26-4 shows the resource file for the Moon.VBP sample application. I designed this simple resource file to demonstrate the loading of strings and one type of binary data file (the Moon icons). The sample program I describe here loads these resources as required, to show a very approximate phase-of-the-moon report for any given date. Start a new Standard EXE project, add a TextBox control named *txtDate*, a Label control named *lblString*, an Image control named *imgMoon*, and a CommandButton control named *cmdPhase*. Refer to Figure 26-5 for the placement and appearance of these controls. If you want, you can add a prompting label next to the text box and change the form's caption to spruce up the form a little. Add the compiled resource file to the project: from the Project menu, choose Add File and then select MOON32.RES.

Add the following code to the form:

```
Option Explicit

Private Sub cmdPhase_Click()
    Dim Phase, ResNum
    'Calculate phase
    Phase = CDbl(CVDate(txtDate.Text))
    Phase = Phase / 29.530589 + 0.9291176
    Phase = Phase - Int(Phase)
    'Calculate resource number
    ResNum = 2 + Int(Phase * 8 + 0.5)
    If ResNum = 10 Then ResNum = 2
    'Load bitmap and string
    imgMoon.Picture = LoadResPicture(ResNum, vbResIcon)
    lblString.Caption = LoadResString(ResNum)
End Sub

Private Sub Form_Load()
    txtDate.Text = Date$
    cmdPhase_Click
End Sub
```

Most of this code is used to calculate the approximate phase of the moon for any given date. Once the phase is determined, a corresponding image and a string description are loaded from the project's resource file. The LoadResPicture and LoadResString functions do the loading. Figure 26-5 shows the Approximate Phase Of The Moon form during development.

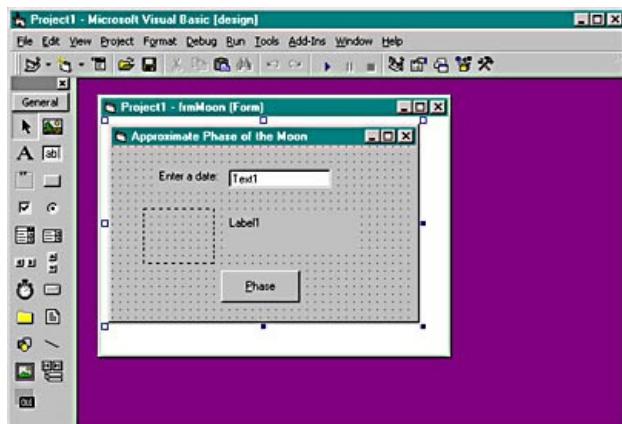


Figure 26-5. The Approximate Phase Of The Moon form under construction.

Figure 26-6 below shows the program in action, displaying and describing the approximate phase of the moon for July 4, 1998.

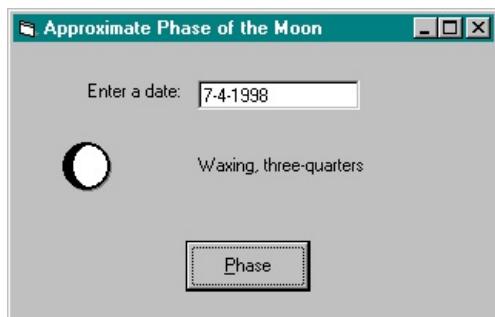


Figure 26-6. The Approximate Phase Of The Moon program in action.

Deciding When to Use a Resource File

As mentioned, the original motivation behind Microsoft's decision to add resource file capabilities to Visual Basic was to make it easier for applications to be internationalized. In a form's Load event procedure, for

example, you can easily load string resources into the Caption properties of command buttons, forms, and labels. Menus, sounds, pictures, text boxes—you name it—can all be modified at runtime to reflect the language of the user. A big advantage of this approach is that the task of reconstructing an application for one or more foreign languages is reduced to the single task of duplicating and editing an external ASCII resource file. You can then turn the resource file over to a skilled translator, who can create a new ASCII resource file without any working knowledge of Visual Basic programming.

In addition to making it easier to internationalize, resource files provide a few other advantages worth mentioning: For one thing, application speed is improved—graphics images load faster from resource files than from individual external files. A second advantage is that multiple images are easier to manipulate from resource files. With earlier versions of Visual Basic, a common technique for manipulating multiple images was to load them into multiple controls. The Moon icons, for example, can be loaded into separate Image or PictureBox controls, and these controls can then be manipulated to display one moon image at a time. Since Visual Basic 4, however, users have been able to store the bitmaps in a resource file and then load each bitmap into a single control as needed, thus avoiding the complexity and processing time of manipulating multiple controls.

Dear John, How Do I... Use a String Database for Internationalization?

You can use a database to store strings that need to be translated and load those strings as needed. This lets you use Access or Visual Basic to build tools that simplify the translation process. In general, resource files are simpler and easier to program than databases, but when localization is a primary concern it's hard to beat a string database.

The structure of a string database can be very simple. In the example shown in Figure 26-7, a single recordset contains all the translated versions of each control on a form. Three option buttons on the start form determine which language to use.

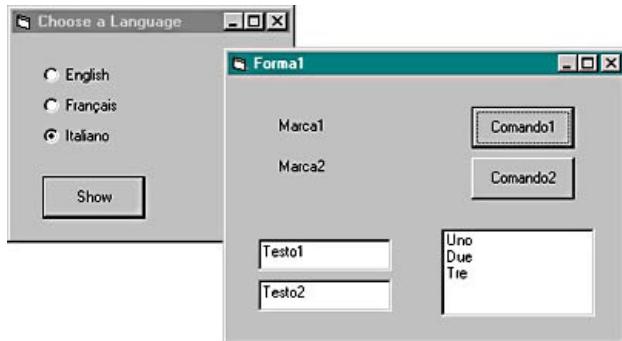


Figure 26-7. The Language Database program uses a string database to display the same form in three different languages.

Entries in the database are indexed by the control's name, so a For Each loop can retrieve the translated version of each control's caption or text based on the name of the control. A Select Case statement decides which property to change, based on the control type, which can be determined using the TypeName function. The code is shown below.

```
Option Explicit
```

```
Private Sub cmdShow_Click()
    'Translate the test form into the
    'selected language
    If optLanguage(0) Then
        ShowLocal frm1, "English"
    ElseIf optLanguage(1) Then
        ShowLocal frm1, "French"
    Else
        ShowLocal frm1, "Italian"
    End If
End Sub

`Translate the strings on the form
Sub ShowLocal(frmLocal As Object, Language As String)
    Dim wrkDatabase As Workspace
    Dim dbStrings As Database
    Dim recStrings As Recordset
    Dim vntString As Variant
    Dim intCount As Integer
    Dim cntIndex As Control
    `Create workspace, open database, and get recordset
    Set wrkDatabase = CreateWorkspace("", "admin", "", dbUseJet)
    Set dbStrings = wrkDatabase.OpenDatabase("strings.mdb")
    Set recStrings = dbStrings.OpenRecordset("Strings")
    `Use names of controls as index in recordset
    recStrings.Index = "ControlName"
    `Internationalize each control name
    For Each cntIndex In frmLocal.Controls
        recStrings.Seek "=", cntIndex.Name
        Select Case TypeName(cntIndex)
            `Change Text property for text boxes
            Case "TextBox"
```

```

        cntIndex.Text = recStrings.Fields(Language)
        'Change captions on others
        Case "Label", "OptionButton", "CheckBox", "Frame", _
            "CommandButton"
            cntIndex.Caption = recStrings.Fields(Language)
        'Change List property for list boxes (record contains
        'an array)
        Case "ListBox", "ComboBox"
            vntString = MakeArray(recStrings.Fields(Language))
            For intCount = 0 To UBound(vntString)
                cntIndex.AddItem vntString(intCount)
            Next intCount
        'Ignore pictures, timers, scrollbars, and so on
        Case Else
            End Select
        Next cntIndex
    'Internationalize form name
    recStrings.Seek "=", frmLocal.Name
    frmLocal.Caption = recStrings.Fields(Language)
    'Close recordset and database
    recStrings.Close
    dbStrings.Close
    'Show the form
    frmLocal.Show
End Sub

'Utility function to convert a semicolon-delineated list
'to an array
Function MakeArray(strSource As String) As Variant
    Dim vntTemp()
    Dim intCount As Integer, intPos As Integer
    Do
        ReDim Preserve vntTemp(intCount)
        intPos = InStr(strSource, ";")
        If intPos Then
            vntTemp(intCount) = Left(strSource, intPos - 1)
            strSource = Right(strSource, Len(strSource) - _
                (intPos + 1))
            intCount = intCount + 1
        Else
            vntTemp(intCount) = strSource
            Exit Do
        End If
    Loop
    MakeArray = vntTemp
End Function

```

I've included an Edit button on the startup form so that you can easily modify the database that contains all the string information. I created the form shown in Figure 26-8 using the Data Form Wizard on the STRINGS.MDB database. STRINGS.MDB was created using Access.

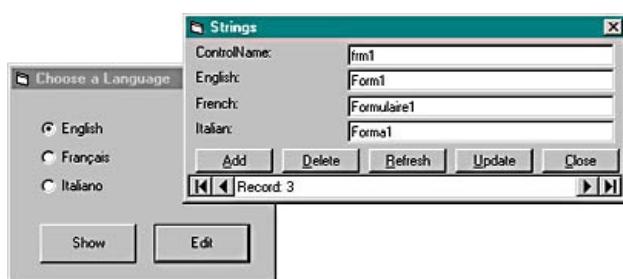


Figure 26-8. The Strings database form allows you to modify the STRINGS.MDB database.

You can use the DebugBuild flag to control the display of the Edit button, preventing users from changing

the string database. Just set the Edit button's Visible property to *False*, and use the following code to change it for debug builds:

```
#Const DebugBuild = 1

#If DebugBuild Then
    Private Sub Form_Load()
        'Display Edit button only on debug builds
        cmdEdit.Visible = True
    End Sub

    Private Sub cmdEdit_Click()
        'Display data entry form for the database
        frmStrings.Show
    End Sub
#End If
```

In this way, you can create applications that contain built-in tools for maintaining translated versions.

SEE ALSO

- The Designing International Software and General Considerations When Writing International Code topics in Visual Basic Books Online for more information about internationalization issues

Chapter Twenty-Seven

Advanced Programming Techniques

With Visual Basic, it's easy to streamline and extend the capabilities of your applications. This chapter presents a few advanced programming techniques that will help you make your Visual Basic applications more efficient and robust.

First I show you how to create a dynamic link library (DLL), which is a great way to speed up the execution of critical sections of your Visual Basic applications. Prior to Visual Basic 4, the only way you could create a DLL was by using another programming language, such as C. Although C is still a good choice, now you can also use Visual Basic itself to create high-speed, compiled DLL modules using in-process ActiveX technology. I'll show you examples of both DLL creation techniques.

Although not new to Visual Basic, *remote automation* makes its debut in this chapter. Remote automation lets you access objects running on servers from your local machine. The Visual Basic Enterprise Edition can create remote applications that run on Windows NT and Windows 95 servers. I'll walk you through creating, debugging, installing, and using a simple remote application. Plus I'll give you some troubleshooting tips for working with remote automation.

Next I'll introduce another feature of Visual Basic: the ability to create add-ins for Visual Basic's integrated development environment (IDE). In this section, you will learn how to create a simple add-in using Visual Basic itself.

I'll also talk about how you can make your application interact with macros written in VBScript. These macros can be run outside your application using the Windows Scripting Host (WScript.EXE) or within your application using the Microsoft Script control (MSScript.OCX). Macros let users automate repetitive tasks and create testing suites for applications that provide ActiveX objects.

Finally I'll show you how you can define and use object properties that have user-defined types (UDTs).

Dear John, How Do I... Use Visual Basic to Create an ActiveX DLL?

You can use Visual Basic to create DLLs for 32-bit Windows-based applications. Visual Basic uses in-process ActiveX technology to accomplish this. A DLL created in this manner is a real DLL, complete with the filename extension DLL. The main difference between an in-process ActiveX DLL and a conventional DLL for earlier versions of Windows is the way that calling programs interface with the provided routines. For example, you create and access objects in an ActiveX DLL by declaring a variable of the object's type, and then you use that object's properties and methods to achieve the desired results. With the old DLL calling convention, you use the Declare statement to define functions within the DLL, which is not possible with ActiveX DLLs, which provide objects, not just functions and procedures.

The Fraction Object

In this example, you will create a simple in-process ActiveX DLL, step by step, to see how this works. For complete information about all the technical details and guidelines for creating DLLs of this type, see the Visual Basic online documentation. However, if you follow all the steps listed here, you should be able to create this example DLL with no complications. You'll create an in-process ActiveX DLL component named Math that provides a Fraction class, which will let a calling program perform simple math operations with fractions. The Fraction object will provide two public properties, Num and Den, and four methods for performing addition, subtraction, multiplication, and division of fractions. One private method within the Fraction class will assist in reducing fractions to lowest terms.

Start a new project, and double-click the ActiveX DLL icon in the New Project dialog box. Visual Basic will add a class module to your project and set the project type to ActiveX DLL. In the Properties window, change the class module's Name property to *Fraction*. This is the name other applications will use to create instances of this object. From the File menu, choose Save Fraction As, and save the class module as MATHFRAC.CLS. This is the name of the file itself—I like to combine the name of the project and the name of the class to identify the file.

This project will contain only the Fraction class module, but let's suppose that we want to add more object definitions of a related, mathematical nature at a later time. For this reason, set the project's Name property to *Math* (instead of *Fraction*). To do this, choose Project Properties from the Project menu to display the Project Properties dialog box. In the Project Name text box, type *Math*, and then click OK. Then choose the Save Project As option from the File menu, and save the project as MATHPROJ.VBP.

It's useful to keep in mind the distinction between the names of files that make up the development of a component and the Name properties of the various parts within that component. For example, the project's filename—in this case, MATHPROJ.VBP—is important only during the development cycle of the project. However, the project's Name property—in this case, *Math*—can be important for programmatically identifying the component containing the Fraction object. For example, if several ActiveX DLLs are used by an application, there might be two Fraction objects defined in different components. If so, you can explicitly identify a particular Fraction object by prefixing references to it with the name of its component. The following line, for example, would create an instance of the Fraction object that's defined within the Math project, even if some other Fraction object is defined in a different component:

```
Public Frac As New Math.Fraction
```

If only one Fraction class is defined within all components that make up an application, the following line suffices to create an instance of the object:

```
Public Frac As New Fraction
```

The next step is to fill out the Fraction class module with code to define its two public properties, four public methods, and one private method. Add the following code to MATHFRAC.CLS:

```
Option Explicit
```

```
Public Num As Integer
Public Den As Integer
```

```
Public Sub Add(Num2, Den2)
    Num = Num * Den2 + Den * Num2
```

```

    Den = Den * Den2
    Reduce
End Sub

Public Sub Sbt(Num2, Den2)
    Num = Num * Den2 - Den * Num2
    Den = Den * Den2
    Reduce
End Sub

Public Sub Mul(Num2, Den2)
    Num = Num * Num2
    Den = Den * Den2
    Reduce
End Sub

Public Sub Div(Num2, Den2)
    Mul Den2, Num2
End Sub

Private Sub Reduce()
    Dim s As Integer
    Dim t As Integer
    Dim u As Integer

    s = Abs(Num)
    t = Abs(Den)
    If t = 0 Then Exit Sub
    Do
        u = (s \ t) * t
        u = s - u
        s = t
        t = u
    Loop While u > 0
    Num = Num \ s
    Den = Den \ s
    If Den < 0 Then
        Num = -Num
        Den = -Den
    End If
End Sub

```

The Public properties Num and Den let a calling application set and read values that define a fraction. The four public methods Add, Sbt, Mul, and Div perform fraction math on the Num and Den properties using a second fraction that is passed as two parameters. (Notice that I couldn't use the more logical Sub abbreviation to name the subtract procedure, because this is reserved by Visual Basic for naming subroutines.) The Reduce method is private and is used internally by the DLL to reduce all fraction results to lowest terms.

We're almost ready to compile and test the Math ActiveX DLL, but first open the Project menu and choose Math Properties to display the Project Properties dialog box. Notice that the Startup Object drop-down list is already set to *(None)* because we didn't add a Sub Main procedure to our project as a startup point. If your object needs to perform any initialization steps, add a Sub Main procedure and change the Startup Object setting in this dialog box to *Sub Main*. In this case, our Fraction object gets along just fine without any initialization code, so leave the setting as *(None)*.

The Project Description text box is blank, but you should always enter a short description of your project here. This is the text users will see later when they are adding a reference to your component to their projects, and it helps identify the purpose of your component. In this case, type in *Demonstration of Fraction Object* or something similar and click OK.

The final step is to compile the Math project to create a completed ActiveX DLL. Choose Make MATHPROJ.DLL from the File menu to take this final step. In the Make Project dialog box, select a

location in which to save the DLL and click OK. This compiles the DLL and registers it in your computer, and you're good to go! Because Visual Basic lets us load and work with multiple projects simultaneously, let's proceed by testing the Math component right in the Visual Basic development environment.

Testing in the Development Environment

To test an ActiveX DLL in the development environment, choose Add Project from the File menu and double-click the Standard EXE icon in the Add Project dialog box. Notice that both ActiveX DLL and Standard EXE projects are listed in the Project Explorer window. Our new project won't be anything fancy—it's designed simply to test our new Fraction object in the Math component—so leave the form name as Form1 and the project name as Project1. We do need to make Project1 the startup project, however, because nothing will happen if you try to run with the Math DLL as the startup project. Right-click Project1 within the Project Explorer window to display a pop-up menu. Choose Set As Start Up from this menu, and notice that the highlight moves from the Math project to Project1.

Now that a specific project in our project group has been selected as the startup project, we need to make the project aware of the Math component and the objects it defines. From the Project menu, choose References and, in the References dialog box, check the Math check box. Recall that this is the setting we specified for the Math project's Name property. Click OK to close the References dialog box.

Before proceeding with the construction of the test project, save the entire current development environment configuration as a project group. The project group is simply an extension of the concept of saving modules and forms in a project. Only the project group saves information about the currently loaded group of projects. A project filename extension is VBP, and a project group filename extension is VBG. From the File menu, choose Save Project Group As. Save Form1 as FORM1.FRM, the Standard EXE project as PROJECT1.VBP, and the project group as MATHDEMO.VBG.

To Form1, add six text boxes and four buttons in a layout similar to that shown in Figure 27-1. Notice that I added a few Line controls and some labels to improve the appearance a little—you can add these extras if you want to.

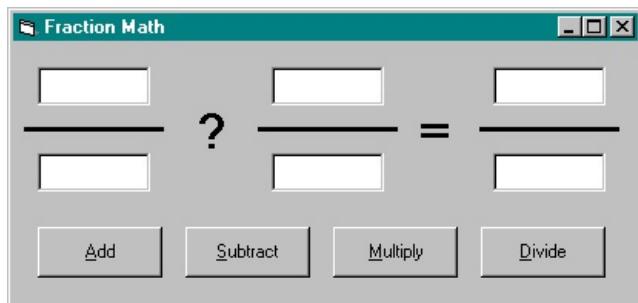


Figure 27-1. A form for testing the new Fraction object.

Name the text controls *txtN1*, *txtD1*, *txtN2*, *txtD2*, *txtN3*, and *txtD3* to represent the numerators and denominators for three fractions as viewed from left to right. Name the command buttons *cmdAdd*, *cmdSubtract*, *cmdMultiply*, and *cmdDivide*, and change their captions to match their functions. Finally, add the following code to the form to complete the application:

```

Option Explicit

Public Frac As New Math.Fraction

Private Sub cmdAdd_Click()
    Frac.Num = txtN1.Text
    Frac.Den = txtD1.Text
    Frac.Add txtN2.Text, txtD2.Text
    txtN3.Text = Frac.Num
    txtD3.Text = Frac.Den
End Sub

Private Sub cmdDivide_Click()
    Frac.Num = txtN1.Text
    Frac.Den = txtD1.Text

```

```

Frac.Div txtN2.Text, txtD2.Text
txtN3.Text = Frac.Num
txtD3.Text = Frac.Den
End Sub

Private Sub cmdMultiply_Click()
    Frac.Num = txtN1.Text
    Frac.Den = txtD1.Text
    Frac.Mul txtN2.Text, txtD2.Text
    txtN3.Text = Frac.Num
    txtD3.Text = Frac.Den
End Sub

Private Sub cmdSubtract_Click()
    Frac.Num = txtN1.Text
    Frac.Den = txtD1.Text
    Frac.Sbt txtN2.Text, txtD2.Text
    txtN3.Text = Frac.Num
    txtD3.Text = Frac.Den
End Sub

```

At the form level, I've declared the variable *Frac* as an object reference of type Fraction, as defined in the Math DLL. The *Frac* object is defined at the module level, but it's not actually created until the first time you click any of the four command buttons. The *Frac* object is destroyed automatically when its reference goes out of scope—that is, when the form unloads.

At this point, you're ready to run the test application and let it create and use a Fraction object as defined in the Math DLL. Enter numbers for the numerators and denominators of the two fractions on the left side of the dialog box, and click the four buttons, one at a time, to perform the fraction math. The result, displayed on the right, should always be a fraction reduced to its lowest terms. Figure 27-2 shows the result of multiplying $\frac{3}{4}$ by $\frac{5}{6}$.

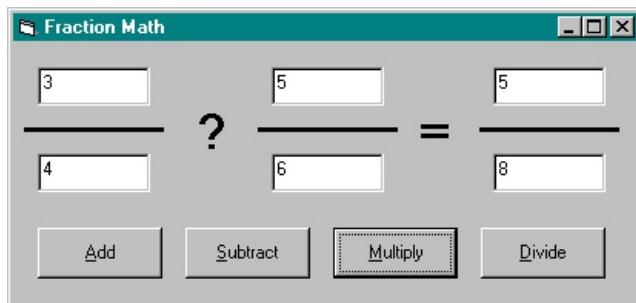


Figure 27-2. The Fraction Math test program in action.

Creating and Using the Final DLL Module

Once your DLL has been debugged and is ready to roll, you'll want to compile your in-process ActiveX project to create a shippable DLL module. All you have to do to achieve this is choose Make MATHPROJ.DLL from the File menu. It's that simple. The DLL will automatically be registered on your computer and is immediately available to any application capable of using ActiveX components.

When your DLL is in the hands of an end user, it must be registered with the user's system before it will show up in the list of references or before any external application will be able to load the DLL into its running space so that it can use the objects the DLL defines. The simplest way to register a DLL with your user's system is to use Visual Basic's Package and Deployment Wizard to create an installation disk. The registration of a DLL is set up automatically by the Package and Deployment Wizard.

Dear John, How Do I... Use C to Create a DLL?

Visual Basic's great strength is in the speed with which it allows you to produce applications for Windows; it's hard to beat the Visual Basic programming environment on the productivity score. On the other hand, C is the language of choice for speed-critical sections of code, which you can often write as a standard block of functions in an old-fashioned DLL (as opposed to an ActiveX) file.

With the latest Visual C++ compilers, creating a DLL is easier than ever. Because this book focuses on using Microsoft tools to create applications for the 32-bit Windows 95 environment, I've streamlined the following sample DLL code. This should make it easier for you to focus on the essential points of the DLL creation task. (If you need to program for the 16-bit Windows environment, or if you're using a version of C other than Microsoft Visual C++ version 2.2 or later, you'll need to make adjustments to the listings.) For the best in-depth explanation of every aspect of DLL creation, refer to the documentation that comes with your compiler.

The Two C Files

The following two listings are the only two files you'll need in your Visual C++ project. Start a new project in the 32-bit version of Visual C++, and select Win32 Dynamic-Link Library as the type of project to be built. Create a DEF file as part of your project, enter the following few lines, and name this file MYDLL.DEF.

```
; Mydll.def
LIBRARY Mydll

CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE

EXPORTS
    TestByte        @1
    TestInteger     @2
    TestLong        @3
    TestSingle      @4
    TestDouble      @5
    ReverseString   @6
```

The DEF file tells the outside world the names of exported functions. In other words, this file provides the list of functions you can call from your Visual Basic applications.

This DLL project has just one C source code file. Enter the following lines of code in a file, save the file as MYDLL.C, and be sure the file is included in your Visual C++ project:

```
#include <windows.h>
#include <ole2.h>

BYTE _stdcall TestByte( BYTE a, LPBYTE b )
{
    *b = a + a;
    return( *b + a );
}

short _stdcall TestInteger( short a, short far * b )
{
    *b = a + a;
    return( *b + a );
}

LONG _stdcall TestLong( LONG a, LPLONG b )
{
    *b = a + a;
    return( *b + a );
}

float _stdcall TestSingle( float a, float far * b )
{
```

```

        *b = a + a;
        return( *b + a );
    }

double __stdcall TestDouble( double a, double far * b )
{
    *b = a + a;
    return( *b + a );
}

void __stdcall ReverseString( BSTR a )
{
    int i, iLen;
    BSTR b;
    LPSTR pA, pB;

    iLen = strlen( (LPCSTR)a );
    b = SysAllocStringLen( NULL, iLen );

    pA = (LPSTR)a;
    pB = (LPSTR)b + iLen -1;

    for ( i = 0; i < iLen; i++ )
        *pB-- = *pA++;

    pA = (LPSTR)a;
    pB = (LPSTR)b;

    for ( i = 0; i < iLen; i++ )
        *pA++ = *pB++;

    SysFreeString( b );
}

```

Click the Build All button in the Visual C++ environment to compile and link the two files in your project and create a small DLL module named MYDLL.DLL. Move or copy MYDLL.DLL to your Windows SYSTEM directory so that your Visual Basic Declare statements will be able to locate the DLL file automatically.

Testing the DLL

It's easy to try out the functions in your new DLL file from Visual Basic. To test the six functions in MYDLL.DLL, I started a new Visual Basic project and added the following code to a form containing a single command button named *cmdGo*:

```

Option Explicit

Private Declare Function TestByte _
Lib "mydll.dll" (
    ByVal a As Byte,
    ByRef b As Byte
) As Byte

Private Declare Function TestInteger _
Lib "mydll.dll" (
    ByVal a As Integer,
    ByRef b As Integer
) As Integer

Private Declare Function TestLong _
Lib "mydll.dll" (
    ByVal a As Long,
    ByRef b As Long
)

```

```
) As Long

Private Declare Function TestSingle _
Lib "mydll.dll" (
    ByVal a As Single,
    ByRef b As Single
) As Single

Private Declare Function TestDouble _
Lib "mydll.dll" (
    ByVal a As Double,
    ByRef b As Double
) As Double

Private Declare Sub ReverseString _
Lib "mydll.dll" (
    ByVal a As String
)

Private Sub cmdGo_Click()
    Dim bytA As Byte
    Dim bytB As Byte
    Dim bytC As Byte
    Dim intA As Integer
    Dim intB As Integer
    Dim intC As Integer
    Dim lngA As Long
    Dim lngB As Long
    Dim lngC As Long
    Dim sngA As Single
    Dim sngB As Single
    Dim sngC As Single
    Dim dblA As Double
    Dim dblB As Double
    Dim dblC As Double
    Dim strA As String

    bytA = 17
    bytC = TestByte(bytA, bytB)
    Print bytA, bytB, bytC

    intA = 17
    intC = TestInteger(intA, intB)
    Print intA, intB, intC

    lngA = 17
    lngC = TestLong(lngA, lngB)
    Print lngA, lngB, lngC

    sngA = 17
    sngC = TestSingle(sngA, sngB)
    Print sngA, sngB, sngC

    dblA = 17
    dblC = TestDouble(dblA, dblB)
    Print dblA, dblB, dblC

    strA = "This string will be reversed"
    Print strA
    ReverseString (strA)
    Print strA
End Sub
```

When you run this program and click the Go button, each of the new DLL functions will be called and the results will be printed on the form, as shown in Figure 27-3.

A few words of explanation about this simple DLL are in order. Each of the five numeric functions demonstrates the passing of one data type in two ways. The first parameter is passed using the `ByVal` keyword, and the second is passed using `ByRef`. Parameters passed with `ByVal` can't be changed by the DLL function, but parameters passed with `ByRef` can be. In this test program, each function is passed a numeric value of 17 in the first parameter, which the DLL function doubles and stores in the second. Each of the numeric functions also returns a value of the same data type, and the value returned by the DLL to the Visual Basic test program is simply the sum of the two parameters. The results displayed in Figure 27-3 show how these values are modified by the DLL functions. In the Visual Basic declarations of the test program, I explicitly declared the `ByRef` parameters using that keyword, but because this is the default for all parameter passing in Visual Basic, you can drop the `ByRef` if you want to. The `ByVal` keyword is not optional, however, and I find it less confusing if I go ahead and explicitly declare both types of parameters. This is a matter of style, and you might prefer to drop all the `ByRef` keywords in your declarations.

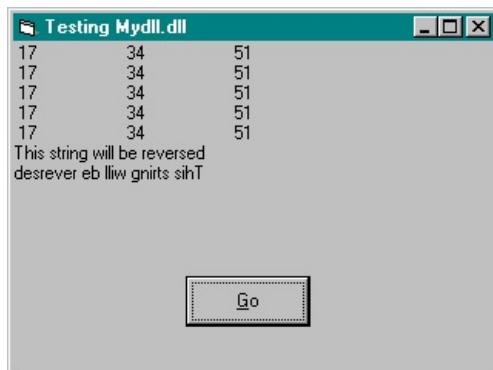


Figure 27-3. The results of testing the functions in MYDLL.DLL.

You might have noticed in the source code that I didn't use C's `int` data type in the function that expects Visual Basic's `Integer` parameters; instead I used the keyword `short`. I avoided C's `int` data type because in 32-bit Microsoft C an `int` is actually a 32-bit integer, rather than the 16-bit size that Visual Basic's `Integer` declaration refers to. Fortunately, if you use `short`, you're guaranteed a 16-bit integer in all versions of Microsoft C.

Strings are now handled internally as `BSTR` types, both in C code, if they are declared as such, and automatically in Visual Basic. In this example, I used a Visual Basic `ByVal` declaration to pass a string to the `ReverseString` function in MYDLL.DLL. You can pass a string using `ByRef`, but because `BSTR` strings are passed around by address anyway, a string passed with `ByVal` can be altered, as shown in this example, and little is gained by passing a string `ByRef`.

In the `ReverseString` function, you'll notice the use of the functions `SysAllocStringLen` and `SysFreeString`. These are just two of several API functions that can be used to manipulate `BSTR` strings, all of which make it easier than ever to manipulate Visual Basic strings within a C-language DLL. The many things you can do with strings in your DLLs is beyond the scope of this book; refer to the Visual C++ documentation for all the details. The simple example functions I've provided here will help you get your feet wet.

SEE ALSO

- The BitPack application in Chapter 34, "[Advanced Applications](#)," for a demonstration of the creation and use of another DLL

Dear John, How Do I... Create an Application That Runs Remotely?

The Visual Basic Enterprise Edition can create applications that run on remote computers but provide properties and methods to applications on your local machine. This is part of a concept known as *distributed computing*, in which processing tasks are shared across CPUs through part of the ActiveX technology called remote automation.

Just about any application that provides public properties and methods can be configured to run remotely. The concepts involved in creating remote applications are the same as those in local ActiveX applications, with just a few more compiler options and the added complexity of managing the system registration for both the remote and local machines.

In this section, I walk you through creating and running a simple remote application that finds prime numbers. It's best to start with a simple application because most of the gotchas that arise in remote automation involve system configuration.

Creating a Remote Application

To create a remote application, start with any ActiveX EXE application and check the Remote Server Files check box on the Component tab of the Project Properties dialog box, as shown in Figure 27-4. The Remote Server Files option tells Visual Basic to generate files that enable client machines to use the remote application. When you compile your application, Visual Basic will generate a registration file (VBR) and a type library (TLB) for use on the client machines.

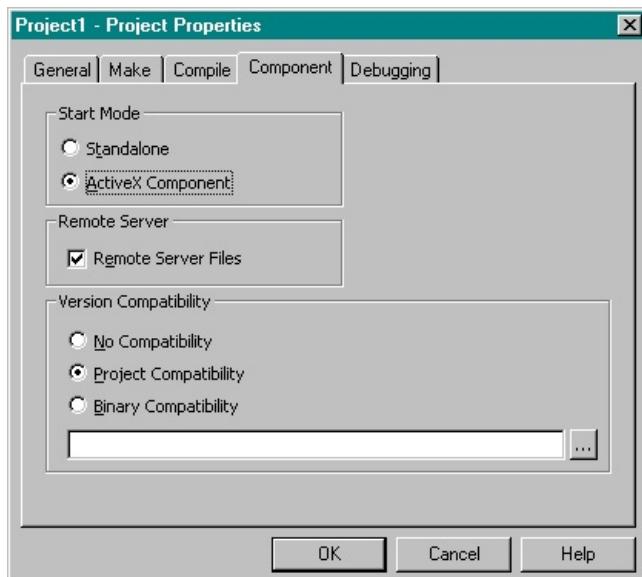


Figure 27-4. The Component tab of the Project Properties dialog box.

The Prime number sample application (PRIME.VBP) demonstrates how you can offload a processor-intensive task to a remote machine. The application provides one object, Number, with a Value property. You set Value to any positive integer, and Prime finds the nearest prime number that's less than or equal to the initial value.

One of the interesting features built in to the Prime number application is its asynchronous processing. Finding large prime numbers around, say, 1 billion takes a lot of time. The Prime number application lets you poll the Value property—if the returned value is nonzero, it is the requested prime number. This lets you find large prime numbers without tying up your local machine.

The following listing shows the Number class module in the Prime number application:

```
'NUMBER.CLS
Dim m1MaxNumber As Long
Dim m1Found As Long

Property Get Value() As Long
    'Return prime number
    'Note that Value is 0 until number is
```

```

`found
Value = mlFound
End Property

Property Let Value(Setting As Long)
    `Initialize module-level variables
    mlMaxNumber = Setting
    mlFound = 0
    `Launch asynchronous calculation
    frmLaunch.Launch Me
End Property

Friend Sub ProcFindPrime()
    Dim Count As Long
    For Count = 2 To mlMaxNumber \ 2
        DoEvents
        If mlMaxNumber Mod Count = 0 Then
            mlMaxNumber = mlMaxNumber - 1
            ProcFindPrime
            Exit Sub
        End If
    Next Count
    mlFound = mlMaxNumber
End Sub

```

The Prime number application uses a Timer control to launch the ProcFindPrime procedure after control has returned to the calling application. This may look like a hack, but it is a useful technique nonetheless. Here is the code for the Launch form named *frmLaunch*, which contains a Timer control named *tmrLaunch*:

```

`LAUNCH.FRM
Dim mnumObject As Number

Public Sub Launch(numObject As Number)
    Set mnumObject = numObject
    tmrLaunch.Enabled = True
    tmrLaunch.Interval = 1
End Sub

Private Sub tmrLaunch_Timer()
    `Turn off timer
    tmrLaunch.Enabled = False
    `Launch calculation within object
    mnumObject.ProcFindPrime
End Sub

```

Registering the Remote Application

The remote application is installed on the server machine and registered on both the server and the client machines. To register the application on the server, simply run the EXE application once on the server. Visual Basic applications are self-registering.

To register the application on the client machine, copy the application's VBR and TLB files to the client, and run the CLIREG32.EXE utility included on the Visual Basic CD-ROM in the COMMON\TOOLS\CLIREG directory. The following command line registers the Prime number application on a client machine and specifies the server named WOMBAT2:

```
CLIREG32 PRIME.VBR -t PRIME.TLB -s WOMBAT2
```

The CLIREG32 utility displays a dialog box that lets you modify the registration entries for the application, as shown in Figure 27-5.

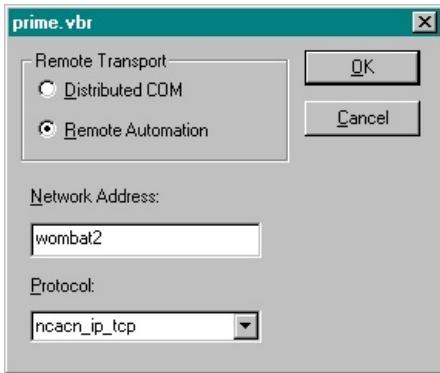


Figure 27-5. Using the CLIREG32 dialog box to set the network protocol for remote automation.

Remote automation supports a number of network protocols: TCP/IP, IPX, and NetBEUI. In general, if one protocol fails, remote automation will attempt to use the other supported protocols that are installed on the client machine. The order in which the protocols are registered for the application determines the search order.

CLIREG32 doesn't do anything magical; it simply provides a front end to the registration file created by Visual Basic. Most registration files in Windows have REG suffixes; you could simply change the suffix of the VBR file to REG, replace the first two lines with REGEDIT4 and a blank line, and use REGEDIT.EXE to register the application. However, you would also need to add server and protocol information to the Registry file. The following listing shows a sample of the Prime number application's VBR file:

```

VB5SERVERINFO
VERSION=1.0.0
HKEY_CLASSES_ROOT\Typelib\{9311AADB-B46F-11D1-8E5B-000000000000}\ \
  1.0\0\win32 = Prime.exe
HKEY_CLASSES_ROOT\Typelib\{9311AADB-B46F-11D1-8E5B-000000000000}\ \
  1.0\FLAGS = 0
HKEY_CLASSES_ROOT\Prime.Number\CLSID =
  {9311AADD-B46F-11D1-8E5B-000000000000}
HKEY_CLASSES_ROOT\CLSID\{9311AADD-B46F-11D1-8E5B-000000000000}\ \
  ProgID = Prime.Number
HKEY_CLASSES_ROOT\CLSID\{9311AADD-B46F-11D1-8E5B-000000000000}\ \
  Version = 1.0
HKEY_CLASSES_ROOT\CLSID\{9311AADD-B46F-11D1-8E5B-000000000000}\ \
  Typelib = {9311AADB-B46F-11D1-8E5B-000000000000}
HKEY_CLASSES_ROOT\CLSID\{9311AADD-B46F-11D1-8E5B-000000000000}\ \
  LocalServer32 = Prime.exe
HKEY_CLASSES_ROOT\INTERFACE\
  {9311AADC-B46F-11D1-8E5B-000000000000} = Number
HKEY_CLASSES_ROOT\INTERFACE\
  {9311AADC-B46F-11D1-8E5B-000000000000}\ProxyStubClid =
  {00020420-0000-0000-C000-00000000046}
HKEY_CLASSES_ROOT\INTERFACE\
  {9311AADC-B46F-11D1-8E5B-000000000000}\ProxyStubClid32 =
  {00020420-0000-0000-C000-00000000046}
HKEY_CLASSES_ROOT\INTERFACE\{9311AADC-B46F-11D1-8E5B-000000000000}\ \
  Typelib = {9311AADB-B46F-11D1-8E5B-000000000000}
HKEY_CLASSES_ROOT\INTERFACE\
  {9311AADC-B46F-11D1-8E5B-000000000000}\Typelib\"version" = 1.0

```

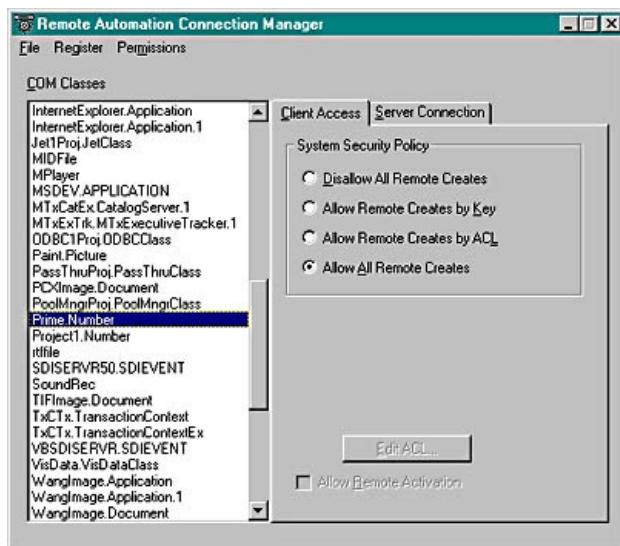
Running the Remote Application

To use an application through remote automation, the server must be running the Automation Manager application (AUTMGR32.EXE) and the server machine's security settings must allow the user to perform the required actions.

To start the Automation Manager, choose Automation Manager from the Windows Visual Basic Start menu. The Automation Manager displays a small window that indicates the server's status, as shown in Figure 27-6.

**Figure 27-6.** The Automation Manager window.

Windows NT provides full security profiles for each user. To use objects from a remote automation server, a user must have access privileges to the server. To start an application that has not already begun running, the application must allow remote creation. The Remote Automation Connection Manager (RACMGR32.EXE) provides a front end to the system Registry remote automation settings, as shown in Figure 27-7.

**Figure 27-7.** Using the Remote Automation Connection Manager on the server machine to set user access to the remote application.

For debugging purposes, set Client Access to Allow All Remote Creates. Later you can restrict access to the server application as needed.

NOTE

Because of differences in the security models, Access Control List (ACL) features available in the Remote Automation Connection Manager are available only on computers running Windows NT and not on those running Windows 95.

Accessing the Remote Application

Once you've set up your remote application and registered the application on both the remote and client machines, you can use objects from the remote server just as you would any local objects. For programming purposes, establish a reference to the application's TLB file. This file can reside locally because it simply provides information about the remote application and does not contain executable code.

The following code shows a simple test of the Prime number application:

```
Option Explicit
```

```
Private Sub Form_Load()
    Dim x As New Prime.Number
    x.Value = 42
    Debug.Print x.Value
End Sub
```

Figure 27-8 shows a sample of the output.

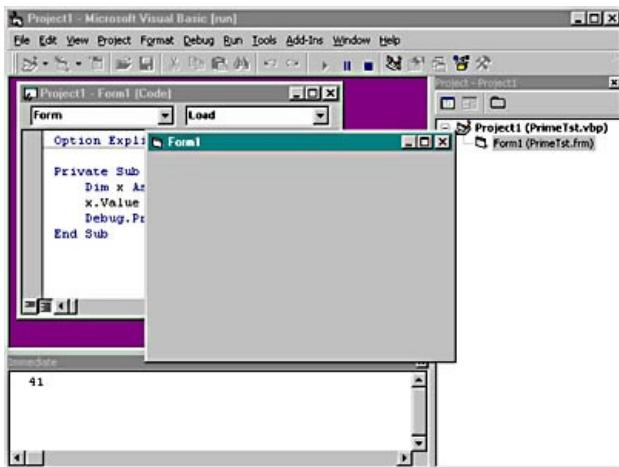


Figure 27-8. Results returned from the remote Prime number application.

Returning Errors from Remote Applications

Errors that occur in a remote application are passed back to the client application, just as they are with local ActiveX applications. You can display user-defined error messages to notify the client applications when something goes wrong. The modifications to the Value property below show how to return an error to the client application:

```
Property Let Value(Setting As Long)
    'Raise error for negative values
    If Setting <= 0 Then
        Err.Raise 8001, "Prime.Number",
            "Value must be a positive integer."
        Exit Property
    End If
    'Initialize module-level variables
    mlMaxNumber = Setting
    mlFound = 0
    'Launch asynchronous calculation
    frmLaunch.Launch Me
End Property
```

Figure 27-9 shows the displayed error message.

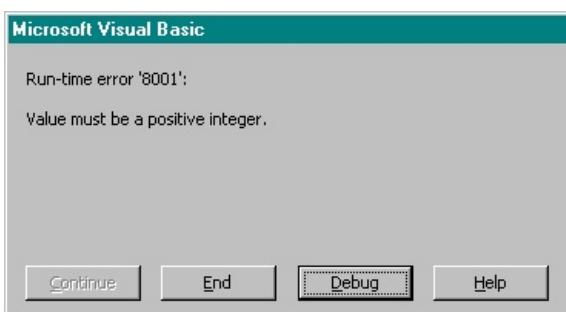


Figure 27-9. Error message displayed if a negative value is entered.

Debugging Remote Applications

You should thoroughly debug remote applications locally before installing them on the server and testing them for remote access. Once you've installed and registered a remote application on the server and client machines, any change to the application requires you to completely reregister the application on both the server and the client machines.

Remote applications may remain loaded in the server machine's memory after remote access, so be sure to check the Windows Task Manager for phantom instances of the application by pressing Ctrl-Alt-Del, as shown in Figure 27-10. End these instances of the application before you reregister new versions of the

remote application.

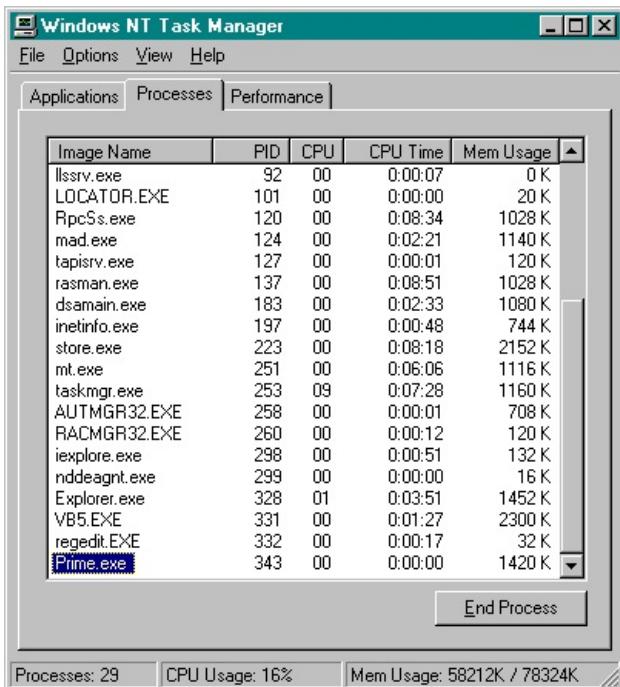


Figure 27-10. Checking the Windows Task Manager for phantom instances of your remote application before you install new versions.

Troubleshooting Remote Automation

Unfortunately, the error messages from remote automation are very general. Most can apply to many different problems, and tracking down all the possible causes can be frustrating. The following table shows a selection of automation errors and lists some of the most likely causes drawn from my experience.

As a rule, remote automation either works well or not at all, and most problems are simple ones—for instance, mistakenly including \\ with the server name or mistakenly including the share name with the server name in the system Registry.

Automation Errors and Causes		
Error Number (&H)	Message Text	Likely Causes
&H800706ba	The RPC server is unavailable.	The remote application's EXE file was not found on the server.
&H800706be	The remote procedure call failed.	Network connection to the server was dropped or timed-out waiting for a response.
&H800706d9	There are no more endpoints from the endpoint mapper.	The Automation Manager (AUTMGR32.EXE) is not running on the server machine.

Dear John, How Do I... Create an Add-In for the Visual Basic Development Environment?

A Visual Basic *add-in* is a special program that can be attached to the Visual Basic development environment to add extra or customized functionality to the environment. For example, you could use an add-in to add some capabilities to the environment in the form of new menu items on Visual Basic's menus or to perform housekeeping chores whenever you load or save forms or code modules. One obviously useful type of add-in is a source code librarian, such as the Microsoft Visual SourceSafe product that ships with the Enterprise Edition of Visual Basic.

Two sample add-in applications, TabOrder and VisData, are included in the MSDN disk's Visual Basic Samples directory, but I found these samples to be somewhat confusing at first because of their complex use of the Visual Basic Extensibility object model. Visual Basic also includes a project template named *Addin* that you can use when creating add-ins. Again, the template's code can be a bit overwhelming without a good explanation of the basic parts.

In this section, I present a streamlined example to help you better understand the mechanics of creating and running an add-in program. Once you have a clear understanding of this example, you'll find the examples provided with Visual Basic much easier to master.

Basic Concepts

As mentioned, a Visual Basic add-in is simply a specially constructed Visual Basic program. The example program presented here will start in the procedure Main, which is located in a standard module (BAS file). This portion of the routine will do nothing more than ensure that the appropriate entry is made in VBADDIN.INI. This entry enables Windows to find and load the add-in when you want it. Registering an add-in is a one-time-only process that is best accomplished by simply running the add-in program once during the installation of the add-in. Because you're creating your own add-in instead of installing a commercial product, you'll run the program once manually, after you have built it.

When you choose Add-In Manager from the Visual Basic Add-Ins menu, all registered add-ins will be available for either loading into or unloading out of the currently running instance of the Visual Basic development environment. For this example, you'll create a special class within the add-in program that implements Visual Basic's extensibility interface, IDTExtensibility. To implement this interface, use the Implements keyword, as shown here:

```
Implements IDTExtensibility
```

Interfaces are a contract with another program—in this case, Visual Basic—that says your program will implement specific procedures that the other program can call. In the case of IDTExtensibility, you must implement four procedures that are called at various times, as described in the following table.

The IDTExtensibility_OnConnection procedure provides a *VBInst* parameter that gives access to the Visual Basic extensibility root object (VBIDE.VBE). From *VBInst*, you can make changes to the Visual Basic menus and toolbars, connect procedures to events in the environment, and perform other actions.

NOTE

Visual Basic 4 used the ConnectEvents method to associate event procedures in a class module with events in the Visual Basic environment. The new Visual Basic extensibility object model provides access to many more aspects of the development environment.

The really cool work is accomplished in the MenuHandler_Click event procedure or in other event-driven procedures provided in the add-in. In this example, you'll use a small subset of the Visual Basic object hierarchy to locate and resize all command buttons on the currently active form within the Visual Basic environment. This is a small example of what can be accomplished with an add-in, yet I find it quite useful. I'm always tweaking my command buttons to make them all the same size in a project, and this little add-in lets me instantly resize all my command buttons to a fixed, standard size.

Building an Add-In

The following paragraphs provide a recipe-style series of steps to show you how to build your relatively simple add-in. When I was building my first add-in, I stumbled over several not-so-obvious details, mostly because I had to wade around in the Microsoft documentation to determine what I had done wrong. The steps presented here should help you breeze through these details.

Start with a new ActiveX EXE project, and insert a second class module and one standard module. Name the standard module *Myaddin*, and name the class modules *Connect* and *Sizer*. You'll need to make some important property settings in the project before you try to run the add-in, but you should enter all the source code first and take care of the property settings when you have finished.

Microsoft® Visual Basic® 6.0 Developer's Workshop

Add the following code to Myaddin. Note that the only task performed by the Main procedure is to register this add-in with Windows. The Main code runs quickly, and the program terminates almost immediately, with no activity visible to the outside world.

```
Option Explicit

`Declare API to write to INI file
Declare Function WritePrivateProfileString _
Lib "Kernel32" Alias "WritePrivateProfileStringA" ( _
    ByVal AppName$, _
    ByVal KeyName$, _
    ByVal keydefault$, _
    ByVal FileName$ _
) As Long

`Declare API to read from INI file
Declare Function GetPrivateProfileString _
Lib "Kernel32" Alias "GetPrivateProfileStringA" ( _
    ByVal AppName$, _
    ByVal KeyName$, _
    ByVal keydefault$, _
    ByVal ReturnString$, _
    ByVal NumBytes As Long, _
    ByVal FileName$ _
) As Long

Sub Main()
    Dim strReturn As String
    Dim strSection As String
    `Be sure you are in the VBADDIN.INI file
    strSection = "Add-Ins32"
    strReturn = String$(255, Chr$(0))
    GetPrivateProfileString strSection, _
        "cmdSizer.Connect", "NotFound", _
        strReturn, Len(strReturn) + 1, "Vbaddin.Ini"
    If InStr(strReturn, "NotFound") Then
        WritePrivateProfileString strSection, "cmdSizer.Connect", _
            "0", "vbaddin.ini"
    End If
End Sub
```

The Connect class module provides the IDTExtensibility_OnConnection and IDTExtensibility_OnDisconnection event procedures that Visual Basic will automatically call when the user loads or unloads this add-in into or out of the Visual Basic environment. It's enlightening to realize that the previously mentioned Main procedure will run *once per installation* of this add-in, whereas the procedures in the Connect class module will run *once per load or unload*. Continuing this pattern, the routines in the other class module will run *once per menu click*. Mentally partitioning this activity will help you understand better what's going on here. Add this code to the Connect class module:

```
Option Explicit

`Indicate that this class implements the extensibility
`interface for Visual Basic
Implements IDTExtensibility

Dim VBInstance As VBIDE.VBE
Dim mnuSize As Office.CommandBarControl
Dim SizerHandler As Sizer

`Set these constants as desired
Const CMDBTNWIDTH = 1200
Const CMDBTNHEIGHT = 400

Private Sub IDTExtensibility_OnConnection _
    (ByVal VBInst As Object, _
    ByVal ConnectMode As vbext_ConnectMode, _
    ByVal AddInInst As VBIDE.AddIn, _
    custom() As Variant)
    `Save this instance of Visual Basic so you can refer to it later
    Set VBInstance = VBInst
    `Add menu item to Visual Basic's Add-Ins menu
    Set mnuSize = VBInstance.CommandBars("Add-Ins").Controls.Add(1)
    mnuSize.Caption = "&Size Command Buttons"
```

```

`Create Sizer object
Set SizerHandler = New Sizer
`Establish a connection between menu events
`and Sizer object
Set SizerHandler.MenuHandler =
    VBInst.Events.CommandBarEvents(mnuSize)
`Pass VBInstance to Sizer object
Set SizerHandler.VBInstance = VBInstance
`Set command button sizing properties
SizerHandler.ButtonWidth = CMDBTNWIDTH
SizerHandler.ButtonHeight = CMDBTNHEIGHT
End Sub

`Removes menu item when user deselects this add-in in
`the Add-In Manager
Private Sub IDTExtensibility_OnDisconnection _
    (ByVal RemoveMode As VBIDE.vbext_DisconnectMode, _
    custom() As Variant)
`Remove menu item
VBInstance.CommandBars("Add-Ins").Controls _
    ("&Size Command Buttons").Delete
End Sub

`The following empty procedures are required because this
`class implements the IDTExtensibility interface
Private Sub IDTExtensibility_OnAddInsUpdate(custom() As Variant)

End Sub

Private Sub IDTExtensibility_OnStartupComplete(custom() As Variant)

End Sub

```

Notice that I've defined two constants in this module, *CMDBTNWIDTH* and *CMDBTNHEIGHT*, which define the size to which all command buttons processed by this add-in will be set. Feel free to change these constants if you want. Better yet, if you feel energetic, you might consider adding a dialog box to this add-in to let the user enter the desired sizing constants on the fly. I decided to keep this example simple by just using constants, which works well for almost all of my command buttons anyway.

The VBInstance object helps you stay in control if the user runs multiple copies of Visual Basic simultaneously. There will be only one instance at a time of your add-in in memory, even if multiple copies of Visual Basic are running. By storing the VBIDE.VBE object passed by Visual Basic to the IDTExtensibility_OnConnection event procedure, you can refer to the forms and controls in that particular instance of Visual Basic when a user selects the new add-in menu item.

The other class module, Sizer, contains the code to be activated when the user selects the new add-in menu item. This module contains one event procedure, MenuHandler_Click, which is automatically called by the system when its containing object is connected to that menu. Add the following source code to the Sizer class module:

```

Option Explicit

`Sizer object properties
Public ButtonWidth As Long
Public ButtonHeight As Long
Public VBInstance As VBIDE.VBE

`Declare menu event handler
Public WithEvents MenuHandler As CommandBarEvents

`This event fires when menu is clicked in IDE
Private Sub MenuHandler_Click _
    (ByVal CommandBarControl As Object, _
    Handled As Boolean, CancelDefault As Boolean)
Dim AllControls
Dim Control As Object
`Get collection containing all controls on form
Set AllControls =
    VBInstance.SelectedVBComponent.Designer.VBControls
`For each control on the active formDear John, How Do I...
For Each Control In AllControls
    `Dear John, How Do I... if the control is a command buttonDear John, How Do I...
    If Control.ClassName = "CommandButton" Then

```

```

`Dear John, How Do I... resize it
With Control.Properties
    .Item("Width") = ButtonWidth
    .Item("Height") = ButtonHeight
End With
End If
Next Control
End Sub

```

When the user selects the new add-in menu item, this subprogram wades through all the controls on the currently selected form within the user's Visual Basic project. If a control is a command button, its Width and Height properties are set to the predetermined constant values. Later, when you try out this add-in, you'll see all the command buttons on your current form snap to a fixed size.

Before you run this program as an add-in, you need to set some important project properties, as shown in Figure 27-11. Choose Project Properties from the Project menu, and click the General tab of the Project Properties dialog box that appears. Set the Startup Object to *Sub Main* so that the correct procedure will run when this program is executed for the first time. Type *cmdSizer* in the Project Name text box. This name must be the same as the first part of the string *cmdSizer.Connect* used in the system registration process executed in the *Main* procedure. If the two don't match, you won't be able to load this add-in using Visual Basic's Add-In Manager tool. This is one of those fine points that tripped me up the first time. Type *Command Button Sizer Add-In* in the Project Description text box. This text appears in the Object Browser and helps identify the exposed objects.

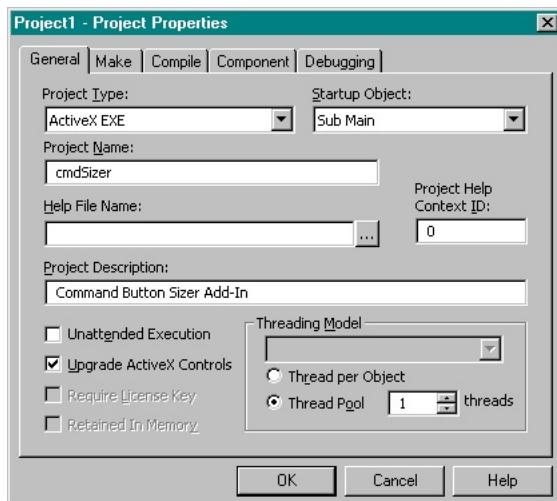


Figure 27-11. Project property settings for the *cmdSizer* add-in.

Next click the Component tab of the Project Properties dialog box, and select ActiveX Component from the StartMode options. An add-in executable contains ActiveX code to be activated from an external application; in this case, the Visual Basic environment uses ActiveX technology to connect to the various objects within our add-in program. Finally, click OK in the dialog box to set your options.

Another trip-up detail is remembering to enable the Microsoft Office objects and the VBIDE.VBE object on which this whole add-in application depends so heavily. Choose References from the Project menu, and check the Microsoft Office 8.0 Object Library and Microsoft Visual Basic 6.0 Extensibility items in the Available References list, and then click OK. If you forget to enable these references, an error message will be displayed when you try to run the program.

The Connect class module must have its Instancing property set to 5 - *MultiUse*. The Instancing setting allows this add-in to be loaded into multiple concurrent instances of Visual Basic. The Instancing property can be set to 1 - *Private* in the Sizer class module.

Running the Add-In for the First Time

Your new add-in must be run once to be registered with Windows. While you are debugging, you can run an add-in within the Visual Basic environment. Once the add-in has been compiled to an EXE file it can be run using the Run command on the Windows Start menu. The system will load the EXE file and make its ActiveX objects available normally, but these objects can still be accessed during debugging if the program is left in the run state.

To try this, click the Run button, and then click the Break button. In the Immediate window, type *Main* and press the Enter key to register the add-in in VBADDIN.INI. Next click the Continue button, and minimize the entire Visual Basic environment while your add-in is still running.

Start a second instance of Visual Basic, and try loading your new add-in using the Add-In Manager item

on the Add-Ins menu. If all has gone smoothly, you'll see the *cmdSizer.Connect* add-in listed in the Add-In Manager dialog box.

NOTE

Since the add-in's project Start Mode is set to ActiveX Component, the Main procedure won't run when you start the program in the Visual Basic environment. You need to run the program manually the first time to register the add-in.

Another approach is to create an EXE file for the add-in. When compilation is successful, exit Visual Basic, run the new CMDSIZER.EXE file once (noticing that nothing much appears to happen while it quickly registers itself with the system), and then start Visual Basic again to see whether you can then load the new add-in. Remember that you need to run the add-in's EXE file only once, to let it register and effectively install itself in the Windows system.

Using the Add-In

To test this add-in, start Visual Basic and choose Add-In Manager from the Add-Ins menu. Select *cmdSizer.Connect*, click the Loaded/Unloaded check box, and then click OK. The Size Command Button menu item is added to the Add-Ins menu, as shown in Figure 27-12.

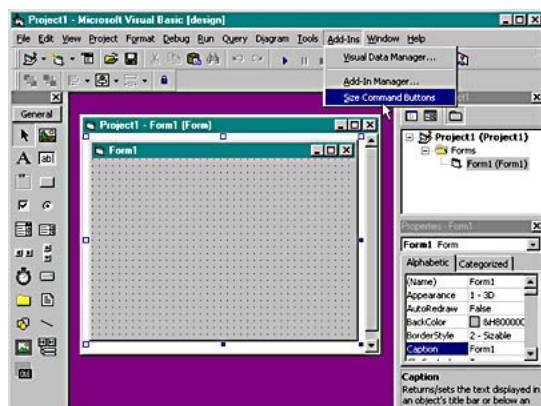


Figure 27-12. The new menu item installed by the *cmdSizer* add-in.

Throw a handful of odd-sized command buttons onto a form, and give this new menu option a try. You should see all the command buttons quickly snap to the same size. Figure 27-13 shows a form before command button resizing, and Figure 27-14, below, shows the same form after I have selected the new menu item.

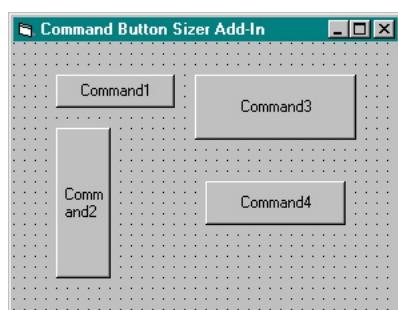


Figure 27-13. Command buttons ready to be resized.

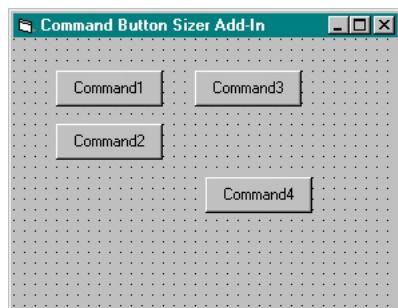


Figure 27-14. Command buttons resized by the add-in.

Dear John, How Do I... Make My Application Scriptable?

The Microsoft Windows Scripting Host (WScript.EXE) executes text files written in VBScript or JScript. Any application that provides ActiveX objects can be scripted using the Windows Scripting Host.

NOTE

The Windows Scripting Host (WScript.EXE) is available from Microsoft's Web site at <http://www.microsoft.com/scripting/windowshost>.

Scripts can be used to automate user tasks and application testing. For example, the following script creates an instance of the Prime sample shown earlier and displays a list of prime numbers:

```
'TstPrime.VBS
dim i, prime, wscript, last, results
`Create objects
set prime = createobject("prime.number")
set wscript = createobject("wscript.shell")

for i = 1 to 20
    prime.value = i
    `Pause until answer is found
    do until prime.value > 0 : loop
    if prime.value <> last then
        `Add unique numbers to results
        results = results & " " & prime.value
    end if
    last = prime.value
next

`Display results
wscript.popup results,, "Prime Test Script"
```

To run the script, double-click its filename in Windows Explorer or use the following command line:

```
wscript tstprime.vbs
```

The last line of TstPrime.VBS uses the PopUp method to display the results as shown in Figure 27-15. PopUp is the equivalent of the Visual Basic MsgBox statement.



Figure 27-15. WScript.EXE runs the test script and displays the result.

Using scripts to test an application is especially handy for applications like Prime that only provide objects and don't have a user interface. However, the true power of scripts comes when you use them within your application to allow users to write macros to automate routine tasks, as described in the next section.

Using the Script Control

The Microsoft Script control (MSScript.OCX) lets you run macros within your application. When using the Script control, you must follow the general steps listed below.

1. Add the Script control to a form.
2. Set the Language property of the control to VBScript or JScript, depending on which language you will use to author macros.
3. At runtime, use the Script control's AddObject method to add references to each object within your application that you want to control with macros. These objects can be private (such as forms or private classes) or they can be public (such as public ActiveX objects).
4. At runtime, use the Script control's AddCode method to load text containing the macro procedures you want to run. Once loaded, use the Script control's Run method to run a macro procedure.

NOTE

The Microsoft Script control (MSScript.OCX) is available from Microsoft's Web site at
<http://www.microsoft.com>.

There are several key points to remember when working with the Script control:

- Loading macros is a one-way operation. You can't export the text of the macros loaded in the Script control; therefore, you must save your macros in an external text file if you want to be able to change them.
- You can't selectively unload objects or macros. Instead, you must use the Reset method to unload all objects and macros, and then reload any modifications using the AddObject and AddCode methods.
- Compile-time errors can occur when the Script control loads a macro, and runtime errors can occur when the Script control runs a macro. You must add error-handling code any time you use the AddCode or Run methods.
- Methods in objects used by macros must use the Variant data type for parameters that accept strings.

The following sections demonstrate how to handle these situations using the Editor sample application (shown in Figure 27-16), which is included on the companion CD-ROM.

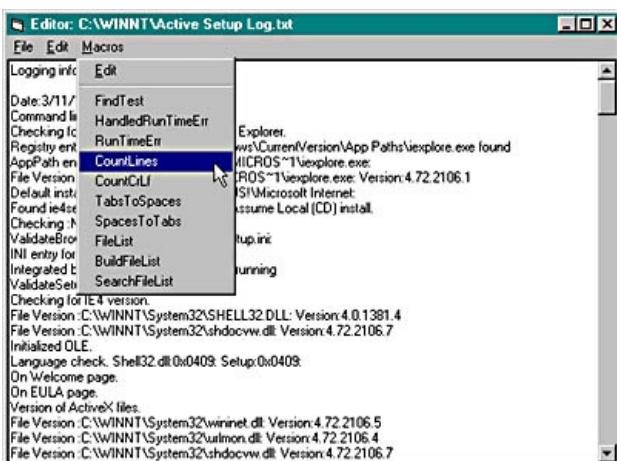


Figure 27-16. The Editor sample (*Editor.VBP*) lets you write macros that modify text files.

Adding Objects and Macro Procedures

Use the Script control's AddObject method to add objects from your application to the script control. The objects you choose to add don't have to be public _ you can add private forms and class modules as well. The objects you add are available in the VBScript code using the name you specify in the AddObject method. For example, the following code adds the Editor object to the Script control:

```
Sub LoadObject()
    scrVB.AddObject "Editor", medtObject, True
End Sub
```

Use the Script control's AddCode method to add the macros you want to run. The AddCode method accepts a string argument containing the code you want to add. You can add the code one procedure at a time or all at once. For instance, the following code loads the entire file Macros.VBS into the Script control:

```
Sub LoadMacros()
    scrVB.AddCode GetText("Macros.VBS")
End Sub

`Used by the preceding code to load file
Function GetText(FileName) As String
    Dim filScript As New Scripting.FileSystemObject
    Dim texScript As TextStream
    `Create a text stream from the FileSystemObject
    Set texScript = filScript.OpenTextFile(FileName, ForReading, True)
    `If file is empty or doesn't exist, don't read it
    If texScript.AtEndOfStream Then
        `Return an empty string
        GetText = ""
    Else
        `Return the text from the text stream into the text box
        GetText = texScript.ReadAll
    End If
    `Close the text stream
    texScript.Close
End Function
```

Since the Script control compiles the code as it is loaded, any compiler errors will occur when the AddCode method is called. The following section describes how to use the Script control's Error object to handle compile-time errors.

Handling Compile-Time Errors

Since users write macros interactively, it is important to give them useful feedback when they make a mistake in their code. The Script control's Error object returns specific information about compile-time errors including the line and column number of the error in the source file.

Within your application, macro compile-time errors occur when the AddCode method loads a macro containing a syntax error. The following code checks for errors after loading the macro and calls the MacroError procedure if any errors occur:

```
Private Sub UpdateMacros()
    `Unload all objects and macros
    scrVB.Reset
    `Add the Editor object to the Script control
    scrVB.AddObject "Editor", medtObject, True
    `Check for errors when code is parsed during
    `loading
    On Error Resume Next
    `Add new code to the Script control
    scrVB.AddCode medtObject.GetText(mstrMacroFile)
    `Call error handler if compile-time error occurs
    If Err Then MacroError
    `Add macros to the menu list
```

```

UpdateMenus
End Sub

Sub MacroError()
    'Using the Script control's Error object
    With scrVB.Error
        'Create a new instance of this form
        Dim frmMacro As New frmEdit
        'Set MacroMode property
        frmMacro.MacroMode = True
        'Show modeless
        frmMacro.Show vbModeless, Me
        'Move the cursor to the error line
        frmMacro.medtObject.MoveDown .Line
        'Move the cursor to the error column
        frmMacro.medtObject.MoveRight .Column
        'Display the error information
        MsgBox Join(Array(.Source, .Description), _
            vbCrLf), vbCritical
        'Hide form so you can show it as modal
        frmMacro.Hide
        'Show form as modal so this code waits for
        'the user to fix the macro
        frmMacro.Show vbModal, Me
        'Update the macro list after corrections are made
        UpdateMacros
    End With
End Sub

```

There's a lot going on in the preceding code. One of the first things happening is that any previous code or objects are cleared from the Script control using the Reset method. Then the Editor object is loaded, followed by the macro code. Any compiler errors cause MacroError to run, which displays the macro file in another editor window.

The Editor sample uses the same form to edit macros as it uses to edit other files. The MoveDown and MoveRight methods are defined in the Editor class module, along with all the other methods that are made available to macros. The Script control's Error object provides the Line and Column properties used by these methods to move the cursor to the line containing the compiler error, as shown in Figure 27-17 below.

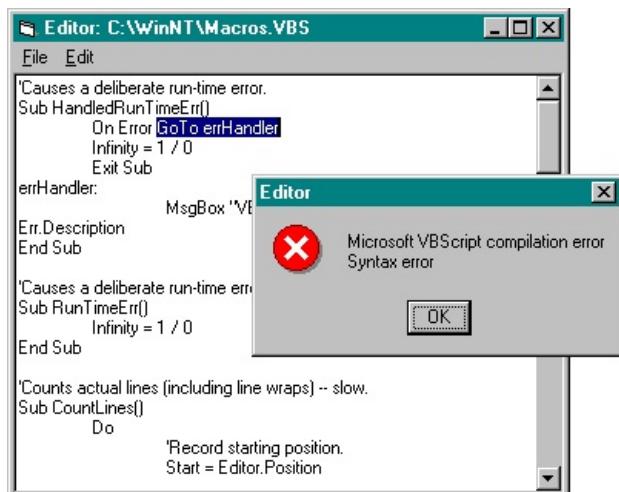


Figure 27-17. When the user modifies a macro, the Editor sample compiles the file and then moves the cursor to any errors that occurred.

The Editor sample first displays the macro edit window as modeless, so that the MoveDown and MoveRight methods can move the cursor within the window. Then the Editor sample hides the window and shows it again as modal so that UpdateMacros doesn't run until after the user fixes the compiler error and closes the window.

Running Macros

Use the Script control's Procedures collection to iterate through the list of loaded macros. The Procedures collection lets you get the name, number of arguments, and return value information for each Sub or Function. For example, the following code adds the macro names to the Macros menu in the Editor application:

```
'Unloads all the names on the Macro menu, then
'reloads names from the Procedures list of the
`Script control
Private Sub UpdateMenus()
    Dim proItem As Procedure
    Dim intCount As Integer
    `Unload Macro menu items
    For intCount = 1 To mnuProcedures.UBound
        Unload mnuProcedures(intCount)
    Next intCount
    intCount = 0
    `Add Macro names to the menu
    For Each proItem In scrVB.Procedures
        intCount = intCount + 1
        Load mnuProcedures(intCount)
        mnuProcedures(intCount).Caption = proItem.Name
    Next proItem
End Sub
```

Use the Script control's Run method to run a macro. The Run method takes the macro name and macro arguments as its own arguments and returns the results of the macro if the macro is a Function. The Editor sample doesn't deal with arguments or return values; it simply runs the macro, as shown here:

```
Private Sub mnuProcedures_Click(Index As Integer)
    `Turn on error handling
    On Error Resume Next
    `Run the selected macro
    scrVB.Run mnuProcedures(Index).Caption
    `If an error occurs, call MacroError
    If Err Then MacroError mnuProcedures(Index).Caption
End Sub
```

Any runtime errors in the macro occur when the Run method executes. The following section describes how to deal with runtime macro errors.

Handling Runtime Errors

Most macro runtime errors return full information about the error in the same way as do compile-time errors. In some cases, however, the Script control can't evaluate what went wrong. For those situations, be sure you pass the macro name to the error handler in order to display at least some information about the error.

The following code shows additions to the MacroError procedure discussed earlier in the section [Handling Compile-Time Errors](#). These changes let you display the procedure name when an Automation error occurs.

```
Sub MacroError(Optional Procedure As String = "")
    `Using the Script control's Error object
    With scrVB.Error
        `Create a new instance of this form
        Dim frmMacro As New frmEdit
        `Set MacroMode property
        frmMacro.MacroMode = True
        `Show modeless
        frmMacro.Show vbModeless, Me
        `If the error number is zero, then it's an
```

```

`Automation error, not a script error
If .Number = 0 Then
    `If it's an Automation error, display the
    `procedure that caused the error
    If Err Then
        MsgBox "The procedure " & Procedure &
                " caused an Automation error.", vbCritical
    End If
    `Otherwise, there is more specific information
    `about the error, so display the line with the error
Else
    `Move the cursor to the error line
    frmMacro.medtObject.MoveDown .Line
    `Move the cursor to the error column
    frmMacro.medtObject.MoveRight .Column
    `Display the error information
    MsgBox Join(Array(.Source, .Description), _
                vbCrLf), vbCritical
End If
`Hide form so you can show it as modal
frmMacro.Hide
`Show form as modal so that this code waits for
`the user to fix the macro
frmMacro.Show vbModal, Me
`Update the macro list after corrections are made
UpdateMacros
End With
End Sub

```

Passing Strings to Methods

All variables used by macros are **Variants**. You must take this into account when creating objects that will be used in macro code. For instance, the following method will cause an Automation error if you try to pass a variable to it from within a macro:

```

`~~~.Find - Finds a string within the current text and
`selects the found string
`Returns True if found, False if not found
Public Function Find(FindText As String) As Boolean
    Dim lngFound As Long
    lngFound = InStr(Position + 1, mfrmParent.txtSource, FindText)
    `If there is a current selection, limit the search
    If mfrmParent.txtSource.SelLength Then
        If lngFound > mfrmParent.txtSource.SelStart + _
            mfrmParent.txtSource.SelLength Then
            Find = False
            Exit Function
        End If
    ElseIf lngFound <> 0 Then
        mfrmParent.txtSource.SelStart = lngFound - 1
        mfrmParent.txtSource.SelLength = Len(FindText)
        Find = True
    Else
        Find = False
    End If
End Function

```

The problem here is subtle because it applies only to variables and not to literals. For instance, the first call to the **Find** method will work, but the macro fails on the second call:

```

`FindTest
Sub FindTest()
    FindString = "Howdy"
    Editor.Find "Howdy"      `Works as expected

```

```
Editor.Find FindString      `Causes Automation error!
End Sub
```

To fix this problem, remove the As String declaration from the definition of the Find method, as shown here:

```
Public Function Find(FindText) As Boolean  `Remove As String
```

Writing Macro Code

Writing macros in VBScript to run in the Script control is a little different than writing regular Visual Basic code. For one thing, VBScript doesn't support the line-continuation character or explicit data types. For another thing, VBScript lacks the file and directory functions included in Visual Basic. Switching between the two languages can give you headaches!

The following macro code was written using the Editor sample. It shows how to use the Editor methods and properties to modify files, list files, and handle errors within VBScript code:

```
`Macros.VBS
`Counts actual lines (including line wraps)
Sub CountLines()
    Do
        `Record starting position
        Start = Editor.Position
        `Move cursor down 1 line
        Editor.MoveDown
        `Keep track of number of lines
        NumLines = NumLines + 1
    `Repeat until cursor doesn't move
    Loop Until Editor.Position = Start
    `Show result
    MsgBox Editor.FileName & " has " & NumLines & " lines."
End Sub

`Changes tabs to 4 spaces
Sub TabsToSpaces()
    `Replace all tabs with spaces
    Editor.ReplaceAll vbTab, "    "
End Sub

`Displays a list of the files in the current directory
Sub BuildFileList()
    `Get a list of the files in the current directory
    Set CurrentDirFiles = FileList(Editor.Directory)
    For Each FileItem In CurrentDirFiles
        `Display the filename
        Editor.Insert FileItem.Name
        `Add a carriage return and line feed
        Editor.Insert vbCrLf
    Next
    `Move back 1
    Editor.MoveLeft
    `Delete the last carriage return and line feed
    Editor.Delete
End Sub

`Returns the collection of files in a specified directory
Function FileList(Directory)
    `Create an object to get folder and file information
    Set FileSys = CreateObject("Scripting.FileSystemObject")
    `Return the collection of files in the directory
    Set FileList = FileSys.GetFolder(Directory).Files
End Function
```

```
`Causes a deliberate runtime error
Sub HandledRunTimeErr()
    On Error Resume Next
    Infinity = 1 / 0
    If Err Then
        MsgBox "VBScript error: " & Err.Number & " " & Err.Description
    End If
End Sub
```

For information on the VBScript language, see the MSDN topic "VBScript Language Reference."

Dear John, How Do I... Pass a User-Defined Type to My Object?

Visual Basic now allows passing arrays and user-defined type (UDT) structures to and from functions and object properties. We already saw an example of passing arrays in the Loan class in Chapter 5, "[Object-Oriented Programming](#)." Now let's take a look at how to pass UDT structures.

Start a new ActiveX DLL project and set the class module's Name property to *MidPoint*. Add the following code to the class module and save it as MidPoint.cls.

```
'MIDPOINT.CLS
Option Explicit

Public Type typeCoordinate
    X As Double
    Y As Double
End Type

'Keep track of the most recent two coordinates
Private mcoordOne As typeCoordinate
Private mcoordTwo As typeCoordinate

'~~~Property (W/O): XY
Property Let XY(coordTest As typeCoordinate)
    'Bump previous coordinate
    mcoordTwo = mcoordOne
    'Store away this coordinate
    mcoordOne = coordTest
End Property

'~~~Property (R/O): XYMid
Property Get XYMid() As typeCoordinate
    'Return the midpoint
    XYMid.X = (mcoordOne.X + mcoordTwo.X) / 2
    XYMid.Y = (mcoordOne.Y + mcoordTwo.Y) / 2
End Property
```

Notice the public Type definition that defines the typeCoordinate UDT structure. All variables passed to and from the properties of the MidPoint object will be of this user-defined type. To keep it simple I defined one write-only property named XY, which is used to pass a coordinate to the MidPoint object, and one read-only property named XYMid, which returns a coordinate calculated to be halfway between the most recently entered two coordinates.

To test the MidPoint object, you need to create a group project by adding a Standard EXE project to the ActiveX DLL project. Select Add Project from the File menu and choose Standard EXE from the Add Project dialog box. In the Project Explorer window, right-click the Standard EXE project and select Set As Start Up from the pop-up menu. Rename the project's Form1 form to *frmPoint*, add the following code to it, set its Caption property to *Please Click on This Form*, and save the form as Point.frm.

```
'POINT.FRM
Option Explicit

Private Sub Form_Click()
    'Create a UDT variable
    Dim coordTest As typeCoordinate
    'Create an object
    Dim midpointTest As New MidPoint
    'Send the first coordinate to the object
    coordTest.X = 3
    coordTest.Y = 4
    midpointTest.XY = coordTest
    'Send the next coordinate to the object
    coordTest.X = 7
    coordTest.Y = 8
    midpointTest.XY = coordTest
```

```
'Get the midpoint from the object
coordTest = midpointTest.XYMid
Print "Midpoint coordinate is ";
Print coordTest.X;
Print ", ";
Print coordTest.Y
End Sub
```

Before this test application will work, the Standard EXE project needs a reference to the ActiveX DLL project. Select the Standard EXE project in the Project Explorer window and then choose References from the Project menu. In the References dialog box, check the box next to the ActiveX DLL project and then click OK.

When you run this test application and click on the form, a new MidPoint object named midpointTest is instantiated (created), along with a UDT structure variable named coordTest. The X and Y values of coordTest are filled with the first coordinate pair, and this variable is then passed to the midpointTest object's XY property. Values for a second coordinate pair are then also passed to the object, which internally remembers the two most recent coordinates passed to it.

The midpointTest.XYMid property returns a coordinate structure, which is then assigned to our coordTest variable. This coordinate is located halfway between the two coordinates remembered by the object, and these X and Y values are displayed on the form for verification, as shown in Figure 27-18.

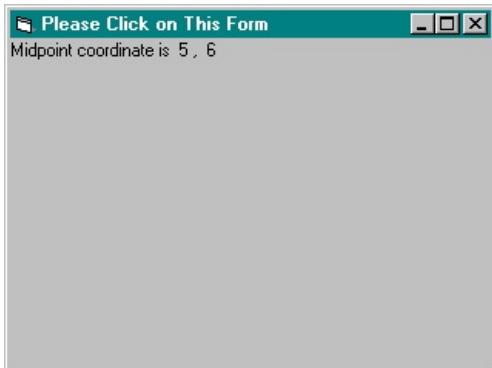


Figure 27-18. Output generated by using a MidPoint object.

The UDT structure passed to and from the object properties in this example is very simple, made up of just two numbers representing an X,Y coordinate pair. The real value of this technique can be seen when you consider how easy it is to pass much larger UDT structures.

Chapter Twenty-Eight

Miscellaneous Techniques

In this chapter, I cover a few odds and ends that don't quite fit anywhere else but that are valuable tricks for the Visual Basic programmer to know.

One of the complaints I've heard in the past about the Basic language in general is its lack of pointers and data structures, features that a competent C programmer couldn't do without. Earlier in this book, I showed how Variants allow great flexibility in the creation of structured data, and in the first section in this chapter, I will show you how the Collection object lets you create dynamic data structures that support very sophisticated data manipulation techniques. As a simple example, I will create the equivalent of a linked list.

Two other techniques that are useful to know are how to detect the operating system version and how to programmatically reboot the computer. You might want to do this, for example, as part of a setup or an initialization application.

The fourth section shows you how your application can dial a phone by sending commands to the modem through the serial port. This is a common programming task, and it turns out to be quite simple.

Finally, the last topic shows a generalized technique for inline error trapping that has advantages over the On Error GoTo Label technique of the past.

Dear John, How Do I... Create a Linked List?

Using a linked list to create, manage, and reorder a sequential series of data has long been a powerful C programming tool. When you create a linked list of data structures in C, you must rely on pointer variables to hold the addresses of linked members in the list. Visual Basic doesn't have explicit pointer variables, but it does provide several of its own techniques to handle references to objects and variables. Behind the scenes, Visual Basic does keep track of pointers, but you don't have to worry about those details. You can focus instead on higher-level concepts of data handling.

In this example, you will create the equivalent of a linked list of strings, in which you can insert strings as new members of the list while maintaining alphabetic order. The Collection object is a powerful tool for accomplishing this task efficiently. Start a new project, add a command button named *cmdBuildList* to a blank form, and add the following lines of code:

```
Option Explicit

Private colWords As New Collection

Sub Insert(V As Variant)
    Dim i As Variant
    Dim j As Variant
    Dim k As Variant
    'Determine whether this is first item to add
    If colWords.Count = 0 Then
        colWords.Add V
        Exit Sub
    End If
    'Get the range of the collection
    i = 1
    j = colWords.Count
    'Determine whether this should be inserted before first item
    If V <= colWords.Item(i) Then
        colWords.Add V, before:=i
        Exit Sub
    End If
    'Determine whether this should be inserted after last item
    If V >= colWords.Item(j) Then
        colWords.Add V, after:=j
        Exit Sub
    End If
    'Conduct binary search for insertion point
    Do Until j - i <= 1
        k = (i + j) \ 2
        If colWords.Item(k) < V Then
            i = k
        Else
            j = k
        End If
    Loop
    'Insert item where it belongs
    colWords.Add V, before:=j
End Sub

Private Sub cmdBuildList_Click()
    Dim i As Integer
    Insert "One"
    Insert "Two"
    Insert "Three"
    Insert "Four"
    Insert "Five"
    Insert "Six"
    Insert "Seven"
    Insert "Eight"
    Insert "Nine"
End Sub
```

```

Insert "Ten"
For i = 1 To colWords.Count
    Print colWords.Item(i)
Next i
End Sub

```

I've declared the `colWords` Collection object at the module level to ensure that it exists for the duration of this program. As with other local variables, if you declare a Collection object within a procedure, it is automatically removed from memory when that procedure ends. When you declare a Collection object at the module level, it exists as long as the form is loaded. Also, a common practice is to create several procedures that process the same collection; this technique allows all the procedures to share the same module-level Collection object.

Collection objects can contain two types of members: Objects and Variants. Of course, a Variant-type variable can contain a wide variety of data types, so a Collection object can actually contain just about anything you want it to contain. In this example, strings are passed and handled as Variants for insertion into the list. Visual Basic Books Online explains in detail how to wrap a Collection object within a class module to better control the type of data that can be added to a collection. This would be important, for instance, if you were to use a Collection object within an ActiveX component that you plan to distribute commercially, in which you might want the Collection object to contain only one type of data. In the example code above, there's nothing to prevent me from passing something other than a string to the `Insert` procedure, even though that would not make sense in the context of what my program is trying to accomplish.

Collection members can be directly accessed by means of a key string or an index number representing the member's position within the list. In this example, I use the index number to control the order in which the members are accessed. A Collection object's index allows you to insert members anywhere in the Collection, delete members from anywhere in the Collection, and generally perform the same kinds of manipulations that a true linked list allows. Like a linked list, the Collection object is a dynamic data structure that grows and shrinks as you add and delete members. Visual Basic handles the details of growing and shrinking the Collection object automatically, without wasting memory.

The Collection object's `Add` method is used to insert each string. Two named arguments of the `Add` method, `before` and `after`, allow you to add a new string just before or just after an indexed location in the object. I use both of these named arguments to insert each string into the object based on alphabetic order. This way the list of strings, when accessed sequentially through the Collection object's index, returns the strings in alphabetic order without requiring any further processing. Figure 28-1 shows the results of printing the Collection object's sorted contents on the form.

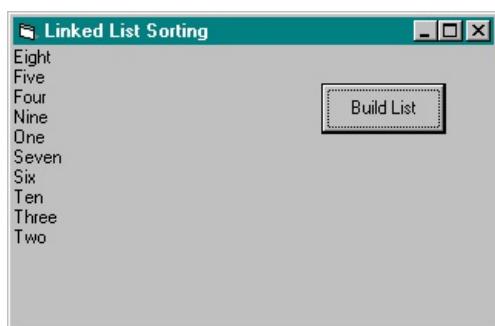


Figure 28-1. Results of using an insertion sort in a Collection object.

Be sure to study Visual Basic Books Online for more information about the Collection object. This powerful and flexible construct, when combined with the flexibility of Variant data types and class module objects, frees you from many of the constraints that Basic imposed in the past, allowing you to program in ways never before possible.

Dear John, How Do I... Respond to O/S Version Differences?

In an ideal world, elves would upgrade everyone's hardware and software silently and simultaneously in the night, all software would be 100 percent upwardly compatible, and everyone would buy *two* copies of this book. Unfortunately, there are still a lot of machines running 16-bit versions of Windows, and even my mom will only chip in for one copy of this book.

There are several strategies for managing applications that have to run on 16-bit and 32-bit systems, but I believe the best is really the simplest: freeze development of your 16-bit application, migrate the code to 32-bit, and incorporate the new features only in the 32-bit version. Trying to maintain equivalent versions of an application for both 16-bit and 32-bit platforms will either seriously hobble the 32-bit version or more than double your workload.

Now, with philosophy and humor out of the way, let's deal with how you detect operating system versions. To tell the difference between a 16-bit and a 32-bit system, you have to write a 16-bit application using Visual Basic 4 or earlier. (Later versions do not provide a 16-bit environment.) The following code shows how to create a stub application that launches the 16-bit or 32-bit version of a setup application based on the current operating system:

```

`LAUNCH.BAS
Option Explicit

#If Win16 Then
Declare Function GetVersion Lib "Kernel" () _
    As Long
#Else
Declare Function GetVersion Lib "Kernel32" () _
    As Long
#End If

`Demonstrates how to get Windows version information
`for 16-bit and 32-bit systems
Sub Main()
    Dim lWinInfo As Long
    Dim strWinVer As String
    Dim strDosVersion As String
    `Retrieve Windows version information
    lWinInfo = GetVersion()
    `Parse Windows version number from returned
    `Long integer value
    strWinVer = LoByte(LoWord(lWinInfo)) & "." & _
        HiByte(HiWord(lWinInfo))
    `If version number is earlier than 3.5 (Win NT 3.5).
    If Val(strWinVer) < 3.5 Then
        Shell "Setup1.EXE"           `Run 16-bit setup;
    Else
        `otherwise,
        Shell "Setup132.EXE"        `run 32-bit setup
    End If

End Sub
Function LoWord(lArg)
    LoWord = lArg And (lArg Xor &HFFFF0000)
End Function

Function HiWord(lArg)
    If lArg > &H7FFFFFFF Then
        HiWord = (lArg And &HFFFF0000) \ &H10000
    Else
        HiWord = ((lArg And &HFFFF0000) \ &H10000) Xor &HFFFF0000
    End If
End Function

Function HiByte(iArg)
    HiByte = (iArg And &HFF00) \ &H100

```

```
End Function
```

```
Function LoByte(iArg)
    LoByte = iArg Xor (iArg And &HFF00)
End Function
```

Notice that I've included conditional code for the 16-bit and 32-bit GetVersion API functions. This lets me develop and debug the application on either platform. The final version must be compiled using 16-bit Visual Basic to create a 16-bit application, however.

NOTE

To run a compiled 16-bit Visual Basic application on a 32-bit system, the 16-bit Visual Basic runtime dynamic link library (DLL) VB40016.DLL must be installed.

Dear John, How Do I... Exit and Restart Windows?

Restarting Windows 95 or Windows NT from your application is not something you're likely to do frequently, but there are times when the ability to restart can be useful. A detected security violation, for instance, can be counteracted with a programmatic restart of the system. Some specialized application installation and setup procedures require a restart to update current paths and Registry settings. It's easy to restart from your code; just be sure to save any open files, including this project, before you try the following code!

To try this example, add a command button named *cmdRestart* to a form, add this code, save the form, and then run it and click the command button:

```
Option Explicit

Const EWX_SHUTDOWN = 1
Const EWX_REBOOT = 2
Const EWX_LOGOFF = 0
Const EWX_FORCE = 4

Private Declare Function ExitWindowsEx Lib "user32" _
    (ByVal uFlags As Long, _
    ByVal dwReserved As Long) _
    As Long

Private Sub cmdRestart_Click()
    'Restart Windows (works on Windows 95/NT)
    ExitWindowsEx EWX_LOGOFF, 0
End Sub
```

Once you've clicked the command button, there's no turning back—the system will shut down and begin the restart process.

NOTE

Windows NT requires your application to have special permissions to use the EWX_SHUTDOWN, EWX_REBOOT, or EWX_FORCE flags. See the Windows API documentation for the *AdjustTokenPrivileges* function for information on how to set application privileges.

Dear John, How Do I... Dial a Phone from My Application?

The Microsoft Comm control is a complete, powerful, and easy-to-use control for handling all your serial communications requirements. One of the most common communications programming tasks is dialing a telephone. The NISTTime application in Chapter 31, "[Date and Time](#)," provides an example of a more sophisticated use of the Comm control, but here I provide the few lines of code required for a quick dial of the phone.

To make this example more useful, I've assumed that the phone number has been copied to the clipboard. This way, you can mark and copy a telephone number from just about anywhere and use it to dial the phone. To try this out, start a new project. If the Comm control is not available, choose Components from the Project menu, and check the Microsoft Comm Control 6.0 check box on the Controls tab of the Components dialog box. To a blank form, add a Comm control named *comOne* and a command button named *cmdDial*. Then add the following code:

```
Option Explicit
```

```
Private Sub cmdDial_Click()
    Dim strA As String
    strA = Clipboard.GetText(vbCFText)
    If strA = "" Then
        MsgBox "Mark and copy a number first."
        Exit Sub
    End If
    comOne.CommPort = 1
    comOne.Settings = "9600,N,8,1"
    comOne.PortOpen = True
    comOne.Output = "ATDT" & strA & vbCrLf
    MsgBox "Dialing " & strA & vbCrLf & "Pick up the phoneDear John, How Do I... ", _
        vbOKOnly, "Dial-A-Phone"
    comOne.PortOpen = False
End Sub
```

Figure 28-2 shows the program in action.

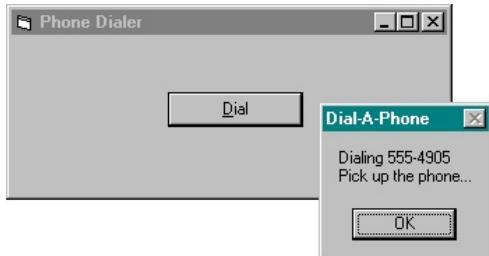


Figure 28-2. The phone-dialing program waiting for the user to pick up the phone.

You will need to change the CommPort property setting to 2 if your modem is installed on COM2 instead of COM1, and in rare cases you might need to change some of the other properties for your hardware. In the vast majority of situations, however, the above property settings will work fine.

The MsgBox command delays the program just before the PortOpen property is set to *False*. As soon as the user clicks the OK button in the message box, the phone is hung up by the Comm control. This gives the user time to pick up the phone before the program proceeds and disconnects the line at the modem.

SEE ALSO

- The NISTTime application in Chapter 31, "[Date and Time](#)," for a demonstration of a more sophisticated use of the Comm control

Dear John, How Do I... Use Inline Error Trapping?

The most commonly suggested way to set up error-trapping code in a Visual Basic program is to add a labeled section of code with an automatic On Error GoTo Label branch set up at the beginning of the procedure. Here's a simple example of this type of error trapping:

```
Private Sub cmdTest_Click()
    On Error GoTo ErrorTrap
    Print 17 / 0 ` (math error)
    Exit Sub

ErrorTrap:
    Print "Illegal to divide by zero."
    Resume Next
End Sub
```

There are several reasons why you might prefer a more inline approach to error trapping, though. First, the error-trap label above is reminiscent of the GOTO labels of ancient spaghetti coding days. Modern structured programming techniques, like those used for much of Visual Basic, have gotten us away from code containing discontinuous jumps and branches, a style that can be identified by the presence of a GOTO command. True, a simple error-trapping GoTo is not as confusing as a lot of Basic code I've seen in the past, but the action is still not as clear as it is when you use other techniques. Even more discouraging, especially when you are dealing with objects in your code, is the confusion you might experience when you are using this method and want to know exactly where the error occurred and which event in your procedure triggered the error. Although the old Err variable is now an enhanced object with properties of its own, the Err object provides limited information, especially in larger procedures.

Inline Error Trapping

The error-trapping approach taken in C programming is to check for returned error information immediately after each function call. The best example of this technique is the way you check for errors in Visual Basic after calling an API function. For example, immediately after you call the mciExecute API function to play a sound file, you can check the returned value to see whether the call was successful:

```
x = mciExecute("Play c:\windows\tada.wav")
If x = 0 Then MsgBox "There was a problem."
```

There is a way to set up generalized error trapping in your Visual Basic code to check for errors immediately after they occur. I like this technique much better than the standard error-trapping method, and Microsoft's documentation does suggest using this technique when you are working with objects. The trick is to use an On Error Resume Next statement at the start of your procedure and to check for possible errors immediately after the lines of code in which errors might occur. The following subprogram demonstrates this technique:

```
Option Explicit

Private Sub cmdTest_Click()
    Dim vntX As Variant
    Dim vntY As Variant
    For vntX = -3 To 3
        vntY = Reciprocal(vntX)
        Print "Reciprocal of "; vntX; " is "; vntY; ""
        If IsError(vntY) Then Print Err.Description
    Next vntX
End Sub

Function Reciprocal(vntX As Variant)
    Dim vntY As Variant
    On Error Resume Next
    vntY = 1 / vntX
    If Err.Number = 0 Then
        Reciprocal = vntY
    End If
End Function
```

```
    Else
        Reciprocal = CVErr(Err.Number)
    End If
End Function
```

Here I've taken advantage of the fact that a Variant can actually be set to a value of type Error using the CVErr conversion function, which is an excellent way to signal back to a calling procedure that an error has occurred. You can, of course, return a value from your functions to indicate an error, but the advantage of signaling an error implicitly through the returned data type is that the error can be indicated even if your function can theoretically return any numeric value.

Figure 28-3 shows the results of running this sample code on a form. Notice that I print the contents of the returned Variant *vntY* even in the case of an error. Because the returned value is a Variant, an error message instead of a number is returned.

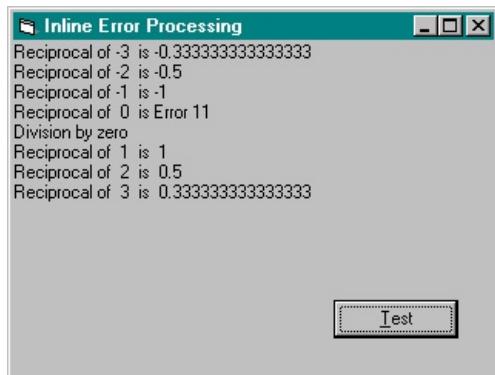


Figure 28-3. *Inline error trapping detecting and reporting a divide-by-zero condition.*

The Err object provides several properties and methods that greatly enhance what you can do with error handling. For example, the Err object's Raise method lets you create your own errors, a handy technique for passing errors back from objects you create. Be sure to review Visual Basic's online documentation to learn about all the error-handling features.

Chapter Twenty-Nine

Graphics

This chapter contains a handful of fun applications that demonstrate some of the graphics techniques and features discussed in Part II of this book. The HSVHSL application is a handy utility for selecting colors. The Animate application demonstrates a few ways to create simple animated graphics on your Visual Basic forms. The Lottery application might not guarantee that you'll win a million dollars, but it will at least provide some fun graphics experimentation, and the MySaver application provides a full-blown screen saver that includes multiple options.

The HSVHSL Application

The HSVHSL application is a simple utility to help you select any shade of color using the RGB (red, green, blue) system, the HSV (hue, saturation, value) system, or the HSL (hue, saturation, luminosity) system of color definition. This application creates instances of the HSV and HSL objects presented in Chapter 14, "[Graphics Techniques](#)," in order to make the necessary conversions between the two color systems. This conversion is accomplished by adding the HSV.CLS and HSL.CLS class modules to this project and creating an instance of each object.

Like many of the programs in Part III, this application includes menu items. Some of the items are not enabled and appear grayed out, waiting for you to add code to give them purpose. All the items on the Help menu, however, are enabled, giving you access to my standard About dialog box and a help file created for this collection of programs. Use of the About dialog box is described in Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)." Notice in the source code that the path to the help file is a relative path. By starting with the application's path, working upward through a couple of directories, and then down to the HELP directory, you can locate the help file regardless of the drive you use for your CD-ROM. You can find the help file even if you copied the contents of the CD-ROM into a directory on your hard drive without changing the basic directory structure.

One line of code might cause you to do a double take. In the Update procedure, the hexadecimal value of the currently selected color value is displayed in a Label control named *lblColor*, just above the picture box that displays the color itself. To create the standard Visual Basic notation for a hexadecimal number, I added an ampersand (&) and an uppercase H prefix to the displayed hexadecimal value. Notice, however, that the string in my program line contains two ampersands in a row:

```
lblColor = "Color = " & "&H" & Hex$(RGBColor)
```

A single ampersand just before the *H* causes the label to display an underlined *H*, a handy feature when you want it, but a nuisance when you want to actually display an ampersand. When you use two ampersands in a row, Visual Basic knows to display a single ampersand instead of an underlined character.

Figures 29-1 through 29-3 show runtime and development-time details of the HSVHSL application. Figure 29-1 shows the HSVHSL application in action. As each slider is moved, HSVHSL displays the color within the picture box to match the slider position and all other sliders are adjusted programmatically. Figure 29-2 shows the contents of the Project window, which provides a list of all the forms and modules that make up this application. Figure 29-3 shows the HSVHSL form during the development process. The numbers on the form identify the form's objects, as listed in the "HSVHSL.FRM Objects and Property Settings" table later in this section.

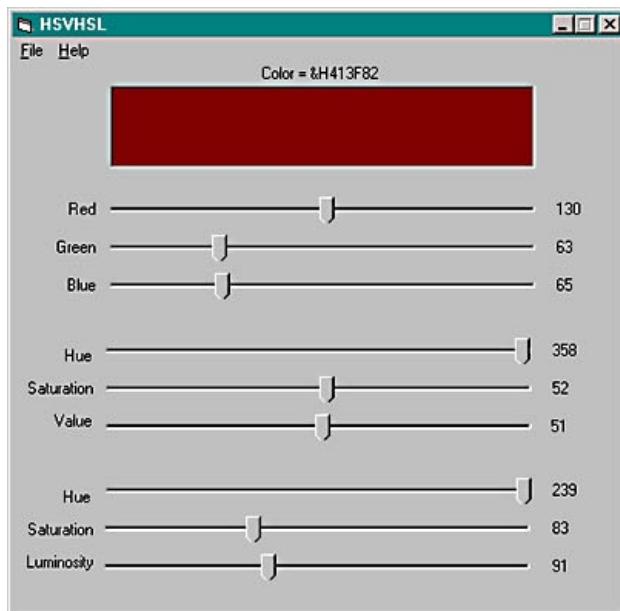


Figure 29-1. The HSVHSL application in action.

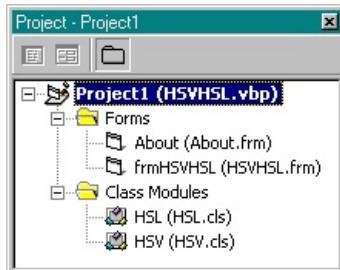


Figure 29-2. The HSVHSL project list.

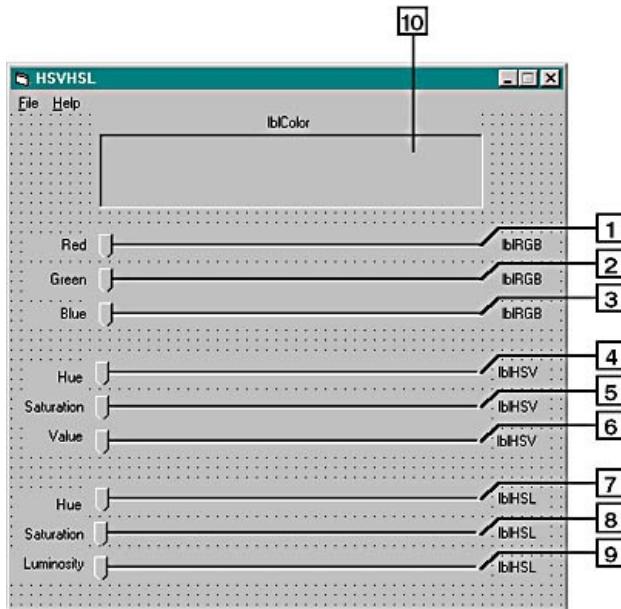


Figure 29-3. HSVHSL.FRM during development.

To create this application, use the tables and source code below to add the appropriate controls, set any nondefault properties as indicated, and enter the source code as shown. I've included the source code for the About form and the HSV and HSL class modules for easy reference, but refer to Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)," for a complete description of the About form and Chapter 14, "[Graphics Techniques](#)," for a complete description of the HSV and HSL classes.

HSVHSL.FRM Menu Editor Settings

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&Open Dear John, How Do I...	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &As Dear John, How Do I...	mnuSaveAs	1	<i>False</i>
—	mnuFileDash1	1	<i>True</i>
E&xit	mnuExit	1	<i>True</i>
&Help	mnuHelp	0	<i>True</i>
&Contents	mnuContents	1	<i>True</i>
&Search For Help On Dear John, How Do I...	mnuSearch	1	<i>True</i>
—	mnuHelpDash1	1	<i>True</i>
&About Dear John, How Do I...	mnuAbout	1	<i>True</i>

HSVHSL.FRM Objects and Property Settings

ID No.*	Property	Value
Slider		
1	Name	<i>sliRGB</i>
	Index	0
	Max	255
	LargeChange	10
Slider		
2	Name	<i>sliRGB</i>
	Index	1
	Max	255
	LargeChange	10
Slider		
3	Name	<i>sliRGB</i>
	Index	2
	Max	255
	LargeChange	10
Slider		
4	Name	<i>sliHSV</i>
	Index	0
	Max	359
Slider		
5	Name	<i>sliHSV</i>
	Index	1
	Max	100
Slider		
6	Name	<i>sliHSV</i>
	Index	2
	Max	100
Slider		
7	Name	<i>sliHSL</i>
	Index	0
	Max	239
Slider		
8	Name	<i>sliHSL</i>
	Index	1

	Max	240
Slider		
9	Name	sliHSL
	Index	2
	Max	240
Label		
	Name	<i>Label1</i>
	Index	0
	Caption	<i>Red</i>
Label		
	Name	<i>Label1</i>
	Index	1
	Caption	<i>Green</i>
Label		
	Name	<i>Label1</i>
	Index	2
	Caption	<i>Blue</i>
Label		
	Name	<i>Label2</i>
	Index	0
	Caption	<i>Hue</i>
Label		
	Name	<i>Label2</i>
	Index	1
	Caption	<i>Saturation</i>
Label		
	Name	<i>Label2</i>
	Index	2
	Caption	<i>Value</i>
Label		
	Name	<i>Label3</i>
	Index	0
	Caption	<i>Hue</i>
Label		
	Name	<i>Label3</i>
	Index	1
	Caption	<i>Saturation</i>

Label

	Name	<i>Label3</i>
	Index	2
	Caption	<i>Luminosity</i>

Label

	Name	<i>lblRGB</i>
	Index	0

Label

	Name	<i>lblRGB</i>
	Index	1

Label

	Name	<i>lblRGB</i>
	Index	2

Label

	Name	<i>lblHSV</i>
	Index	0

Label

	Name	<i>lblHSV</i>
	Index	1

Label

	Name	<i>lblHSV</i>
	Index	2

Label

	Name	<i>lblHSL</i>
	Index	0

Label

	Name	<i>lblHSL</i>
	Index	1

Label

	Name	<i>lblHSL</i>
	Index	2

Label

	Name	<i>lblColor</i>
10	Name	<i>picColor</i>

* The number in the ID No. column corresponds to the number in Figure 29-3 that identifies the location of the object on the form.

Source Code for HSVHSL.FRM

```

Option Explicit

Private Declare Function WinHelp _  

Lib "user32" Alias "WinHelpA" ( _  

    ByVal hwnd As Long, _  

    ByVal lpHelpFile As String, _  

    ByVal wCommand As Long, _  

    ByVal dwData As Long _  

) As Long

Dim RGBColor  

Dim hsvDemo As New HSV  

Dim hslDemo As New HSL

Private Sub Form_Load()  

    'Set a gray starting color  

    With hsvDemo  

        .Red = 127  

        .Green = 127  

        .Blue = 127  

    End With  

    With hslDemo  

        .Red = 127  

        .Green = 127  

        .Blue = 127  

    End With  

    Update  

End Sub

Private Sub mnuAbout_Click()  

    'Set properties  

    About.Application = "HSVHSL"  

    About.Heading =  

        "Microsoft Visual Basic 6.0 Developer's Workshop"  

    About.Copyright = "1998 John Clark Craig and Jeff Webb"  

    'Call a method  

    About.Display
End Sub

Private Sub mnuContents_Click()  

    WinHelp hwnd, App.Path & "\..\..\Help\Mvbdw.hlp", _  

        cdlHelpContents, 0
End Sub

Private Sub mnuExit_Click()  

    Unload Me
End Sub

Private Sub mnuSearch_Click()  

    WinHelp hwnd, App.Path & "\..\..\Help\Mvbdw.hlp", _  

        cdlHelpPartialKey, 0
End Sub

Sub Update()  

    sliRGB(0).Value = hsvDemo.Red  

    sliRGB(1).Value = hsvDemo.Green  

    sliRGB(2).Value = hsvDemo.Blue  

    sliHSV(0).Value = hsvDemo.Hue  

    sliHSV(1).Value = hsvDemo.Saturation  

    sliHSV(2).Value = hsvDemo.Value  

    sliHSL(0).Value = hslDemo.Hue  

    sliHSL(1).Value = hslDemo.Saturation  

    sliHSL(2).Value = hslDemo.Luminosity

```

```

`Update RGB color labels
lblRGB(0).Caption = Format$(hsvDemo.Red, "##0")
lblRGB(1).Caption = Format$(hsvDemo.Green, "##0")
lblRGB(2).Caption = Format$(hsvDemo.Blue, "##0")
`Update HSV color labels
lblHSV(0).Caption = Format$(hsvDemo.Hue, "##0")
lblHSV(1).Caption = Format$(hsvDemo.Saturation, "##0")
lblHSV(2).Caption = Format$(hsvDemo.Value, "##0")
`Update HSL color labels
lblHSL(0).Caption = Format$(sliHSL(0).Value, "##0")
lblHSL(1).Caption = Format$(sliHSL(1).Value, "##0")
lblHSL(2).Caption = Format$(sliHSL(2).Value, "##0")
`Update the displayed color
RGBColor = RGB(hsvDemo.Red, hsvDemo.Green, hsvDemo.Blue)
picColor.BackColor = RGBColor
`Update the color's number
lblColor = "Color = " & "(&H" & Hex$(RGBColor)
End Sub

Private Sub sliHSL_Scroll(Index As Integer)
    hslDemo.Hue = sliHSL(0).Value
    hslDemo.Saturation = sliHSL(1).Value
    hslDemo.Luminosity = sliHSL(2).Value
    hsvDemo.Red = hslDemo.Red
    hsvDemo.Green = hslDemo.Green
    hsvDemo.Blue = hslDemo.Blue
    Update
End Sub

Private Sub sliRGB_Scroll(Index As Integer)
    hsvDemo.Red = sliRGB(0).Value
    hsvDemo.Green = sliRGB(1).Value
    hsvDemo.Blue = sliRGB(2).Value
    hslDemo.Red = hsvDemo.Red
    hslDemo.Green = hsvDemo.Green
    hslDemo.Blue = hsvDemo.Blue
    Update
End Sub

Private Sub sliHSV_Scroll(Index As Integer)
    hsvDemo.Hue = sliHSV(0).Value
    hsvDemo.Saturation = sliHSV(1).Value
    hsvDemo.Value = sliHSV(2).Value
    hslDemo.Red = hsvDemo.Red
    hslDemo.Green = hsvDemo.Green
    hslDemo.Blue = hsvDemo.Blue
    Update
End Sub

```

Source Code for HSV.CLS

```

`HSV.CLS
Option Explicit

`RGB color properties
Private mintRed As Integer
Private mintGreen As Integer
Private mintBlue As Integer

`HSV color properties
Private msngHue As Single
Private msngSaturation As Single

```

```
Private msngValue As Single

`Keep track of implied conversion
Private mintCalc As Integer
Private Const RGB2HSV = 1
Private Const HSV2RGB = 2

`~~~ Hue
Property Let Hue(intHue As Integer)
    msngHue = intHue
    mintCalc = HSV2RGB
End Property
Property Get Hue() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Hue = msngHue
End Property

`~~~ Saturation
Property Let Saturation(intSaturation As Integer)
    msngSaturation = intSaturation
    mintCalc = HSV2RGB
End Property
Property Get Saturation() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Saturation = msngSaturation
End Property

`~~~ Value
Property Let Value(intValue As Integer)
    msngValue = intValue
    mintCalc = HSV2RGB
End Property
Property Get Value() As Integer
    If mintCalc = RGB2HSV Then CalcHSV
    Value = msngValue
End Property

`~~~ Red
Property Let Red(intRed As Integer)
    mintRed = intRed
    mintCalc = RGB2HSV
End Property

Property Get Red() As Integer
    If mintCalc = HSV2RGB Then CalcRGB
    Red = mintRed
End Property

`~~~ Green
Property Let Green(intGreen As Integer)
    mintGreen = intGreen
    mintCalc = RGB2HSV
End Property
Property Get Green() As Integer
    If mintCalc = HSV2RGB Then CalcRGB
    Green = mintGreen
End Property

`~~~ Blue
Property Let Blue(intBlue As Integer)
    mintBlue = intBlue
    mintCalc = RGB2HSV
End Property
Property Get Blue() As Integer
```

```

If mintCalc = HSV2RGB Then CalcRGB
Blue = mintBlue
End Property

`Converts RGB to HSV
Private Sub CalcHSV()
    Dim sngRed As Single
    Dim sngGreen As Single
    Dim sngBlue As Single
    Dim sngMx As Single
    Dim sngMn As Single
    Dim sngDelta As Single
    Dim sngVa As Single
    Dim sngSa As Single
    Dim sngRc As Single
    Dim sngGc As Single
    Dim sngBc As Single
    sngRed = mintRed / 255
    sngGreen = mintGreen / 255
    sngBlue = mintBlue / 255
    sngMx = sngRed
    If sngGreen > sngMx Then sngMx = sngGreen
    If sngBlue > sngMx Then sngMx = sngBlue
    sngMn = sngRed
    If sngGreen < sngMn Then sngMn = sngGreen
    If sngBlue < sngMn Then sngMn = sngBlue
    sngDelta = sngMx - sngMn
    sngVa = sngMx
    If sngMx Then
        sngSa = sngDelta / sngMx
    Else
        sngSa = 0
    End If
    If sngSa = 0 Then
        msngHue = 0
    Else
        sngRc = (sngMx - sngRed) / sngDelta
        sngGc = (sngMx - sngGreen) / sngDelta
        sngBc = (sngMx - sngBlue) / sngDelta
        Select Case sngMx
        Case sngRed
            msngHue = sngBc - sngGc
        Case sngGreen
            msngHue = 2 + sngRc - sngBc
        Case sngBlue
            msngHue = 4 + sngGc - sngRc
        End Select
        msngHue = msngHue * 60
        If msngHue < 0 Then msngHue = msngHue + 360
    End If
    msngSaturation = sngSa * 100
    msngValue = sngVa * 100
    mintCalc = 0
End Sub

`Converts HSV to RGB
Private Sub CalcRGB()
    Dim sngSaturation As Single
    Dim sngValue As Single
    Dim sngHue As Single
    Dim intI As Integer
    Dim sngF As Single
    Dim sngP As Single
    Dim sngQ As Single

```

```

Dim sngT As Single
Dim sngRed As Single
Dim sngGreen As Single
Dim sngBlue As Single
sngSaturation = msngSaturation / 100
sngValue = msngValue / 100
If msngSaturation = 0 Then
    sngRed = sngValue
    sngGreen = sngValue
    sngBlue = sngValue
Else
    sngHue = msngHue / 60
    If sngHue = 6 Then sngHue = 0
    intI = Int(sngHue)
    sngF = sngHue - intI
    sngP = sngValue * (1! - sngSaturation)
    sngQ = sngValue * (1! - (sngSaturation * sngF))
    sngT = sngValue * (1! - (sngSaturation * (1! - sngF)))
    Select Case intI
        Case 0
            sngRed = sngValue
            sngGreen = sngT
            sngBlue = sngP
        Case 1
            sngRed = sngQ
            sngGreen = sngValue
            sngBlue = sngP
        Case 2
            sngRed = sngP
            sngGreen = sngValue
            sngBlue = sngT
        Case 3
            sngRed = sngP
            sngGreen = sngQ
            sngBlue = sngValue
        Case 4
            sngRed = sngT
            sngGreen = sngP
            sngBlue = sngValue
        Case 5
            sngRed = sngValue
            sngGreen = sngP
            sngBlue = sngQ
    End Select
End If
mintRed = Int(255.9999 * sngRed)
mintGreen = Int(255.9999 * sngGreen)
mintBlue = Int(255.9999 * sngBlue)
mintCalc = 0
End Sub

```

Source Code for HSL.CLS

```

`HSL.CLS
Option Explicit

`RGB color properties
Private mintRed As Integer
Private mintGreen As Integer
Private mintBlue As Integer

`HSL color properties

```

```
Private msngHue As Single
Private msngSaturation As Single
Private msngLuminosity As Single

`Keep track of implied conversion
Private mintCalc As Integer
Private Const RGB2HSL = 1
Private Const HSL2RGB = 2

`~~~ Hue
Property Let Hue(intHue As Integer)
    msngHue = (intHue / 240!) * 360!
    mintCalc = HSL2RGB
End Property
Property Get Hue() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Hue = (msngHue / 360!) * 240!
End Property

`~~~ Saturation
Property Let Saturation(intSaturation As Integer)
    msngSaturation = intSaturation / 240!
    mintCalc = HSL2RGB
End Property
Property Get Saturation() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Saturation = msngSaturation * 240!
End Property

`~~~ Luminosity
Property Let Luminosity(intLuminosity As Integer)
    msngLuminosity = intLuminosity / 240!
    mintCalc = HSL2RGB
End Property
Property Get Luminosity() As Integer
    If mintCalc = RGB2HSL Then CalcHSL
    Luminosity = msngLuminosity * 240!
End Property

`~~~ Red
Property Let Red(intRed As Integer)
    mintRed = intRed
    mintCalc = RGB2HSL
End Property

Property Get Red() As Integer
    If mintCalc = HSL2RGB Then CalcRGB
    Red = mintRed
End Property

`~~~ Green
Property Let Green(intGreen As Integer)
    mintGreen = intGreen
    mintCalc = RGB2HSL
End Property
Property Get Green() As Integer
    If mintCalc = HSL2RGB Then CalcRGB
    Green = mintGreen
End Property

`~~~ Blue
Property Let Blue(intBlue As Integer)
    mintBlue = intBlue
    mintCalc = RGB2HSL
```

```

End Property
Property Get Blue() As Integer
    If mintCalc = HSL2RGB Then CalcRGB
    Blue = mintBlue
End Property

Private Sub CalcHSL()
    Dim sngMx As Single
    Dim sngMn As Single
    Dim sngDelta As Single
    Dim sngPctRed As Single
    Dim sngPctGrn As Single
    Dim sngPctBlu As Single
    sngPctRed = mintRed / 255
    sngPctGrn = mintGreen / 255
    sngPctBlu = mintBlue / 255
    sngMx = sngMaxOf(sngMaxOf(sngPctRed, sngPctGrn), sngPctBlu)
    sngMn = sngMinOf(sngMinOf(sngPctRed, sngPctGrn), sngPctBlu)
    sngDelta = sngMx - sngMn
    msngLuminosity = (sngMx + sngMn) / 2
    If sngMx = sngMn Then
        msngSaturation = 0
    Else
        msngSaturation = 1
    End If
    If msngLuminosity <= 0.5 Then
        If msngSaturation > 0 Then
            msngSaturation = sngDelta / (sngMx + sngMn)
        End If
    Else
        If msngSaturation > 0 Then
            msngSaturation = sngDelta / (2 - sngMx - sngMn)
        End If
    End If
    If msngSaturation Then
        If sngPctRed = sngMx Then
            msngHue = (sngPctGrn - sngPctBlu) / sngDelta
        End If
        If sngPctGrn = sngMx Then
            msngHue = 2 + (sngPctBlu - sngPctRed) / sngDelta
        End If
        If sngPctBlu = sngMx Then
            msngHue = 4 + (sngPctRed - sngPctGrn) / sngDelta
        End If
        msngHue = msngHue * 60
    End If
    If msngHue < 0 Then msngHue = msngHue + 360
    mintCalc = 0
End Sub

Private Sub CalcRGB()
    Dim sngM1 As Single
    Dim sngM2 As Single
    Dim sngPctRed As Single
    Dim sngPctGrn As Single
    Dim sngPctBlu As Single
    If msngLuminosity <= 0.5 Then
        sngM2 = msngLuminosity * (1! + msngSaturation)
    Else
        sngM2 = (msngLuminosity + msngSaturation) -
            (msngLuminosity * msngSaturation)
    End If
    sngM1 = 2! * msngLuminosity - sngM2
    If msngSaturation = 0! Then

```

```

        sngPctRed = msngLuminosity
        sngPctGrn = msngLuminosity
        sngPctBlu = msngLuminosity
    Else
        sngPctRed = rgbVal(sngM1, sngM2, msngHue + 120!)
        sngPctGrn = rgbVal(sngM1, sngM2, msngHue)
        sngPctBlu = rgbVal(sngM1, sngM2, msngHue - 120!)
    End If
    mintRed = Int(255.9999 * sngPctRed)
    mintGreen = Int(255.9999 * sngPctGrn)
    mintBlue = Int(255.9999 * sngPctBlu)
    mintCalc = 0
End Sub

Private Function rgbVal(sngN1 As Single, sngN2 As Single, _
sngHue As Single) As Single
    If sngHue > 360 Then
        sngHue = sngHue - 360
    ElseIf sngHue < 0 Then
        sngHue = sngHue + 360
    End If
    If sngHue < 60 Then
        rgbVal = sngN1 + (sngN2 - sngN1) * sngHue / 60
    ElseIf sngHue < 180 Then
        rgbVal = sngN2
    ElseIf sngHue < 240 Then
        rgbVal = sngN1 + (sngN2 - sngN1) * (240 - sngHue) / 60
    Else
        rgbVal = sngN1
    End If
End Function

Private Function sngMaxOf(sngV1 As Single, sngV2 As Single) As Single
    sngMaxOf = IIf(sngV1 > sngV2, sngV1, sngV2)
End Function

Private Function sngMinOf(sngV1 As Single, sngV2 As Single) As Single
    sngMinOf = IIf(sngV1 < sngV2, sngV1, sngV2)
End Function

```

Source Code for ABOUT.FRМ

```

Option Explicit

Private Sub cmdOK_Click()
    `Cancel About form
    Unload Me
End Sub

Private Sub Form_Load()
    `Center this form
    Left = (Screen.Width - Width) \ 2
    Top = (Screen.Height - Height) \ 2
    `Set defaults
    lblApplication.Caption = "- Application -"
    lblHeading.Caption = "- Heading -"
    lblCopyright.Caption = "- Copyright -"
End Sub

Public Sub Display()
    `Display self as modal
    Show vbModal

```

```
End Sub

Property Let Application(Application As String)
    'Define string property for Application
    lblApplication.Caption = Application
End Property

Property Let Heading(Heading As String)
    'Define string property for Heading
    lblHeading.Caption = Heading
End Property

Property Let Copyright(Copyright As String)
    'Build complete Copyright string property
    lblCopyright.Caption = "Copyright © " & Copyright
End Property
```

The Animate Application

The Animate application demonstrates a couple of graphics techniques you might find useful in designing your own applications. In the code module ANIMATE.BAS, Sub Main displays two forms, each demonstrating a unique graphics technique. First let's take a look at the source code for the entire code module, and then we'll dive right into the two forms in this project.

Source Code for ANIMATE.BAS

```

Option Explicit
DefDbl A-Z           `<<< NOTICE!!!

Public Const PI = 3.14159265358979
Public Const RADPERDEG = PI / 180

Sub Main()
    App.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    frmClock.Show vbModeless
    frmGlobe.Show vbModeless
End Sub

Sub RotateX(X, Y, Z, Angle)
    Dim Radians, Ca, Sa, Ty
    Radians = Angle * RADPERDEG
    Ca = Cos(Radians)
    Sa = Sin(Radians)
    Ty = Y * Ca - Z * Sa
    Z = Z * Ca + Y * Sa
    Y = Ty
End Sub

Sub RotateY(X, Y, Z, Angle)
    Dim Radians, Ca, Sa, Tx
    Radians = Angle * RADPERDEG
    Ca = Cos(Radians)
    Sa = Sin(Radians)
    Tx = X * Ca + Z * Sa
    Z = Z * Ca - X * Sa
    X = Tx
End Sub

Sub RotateZ(X, Y, Z, Angle)
    Dim Radians, Ca, Sa, Tx
    Radians = Angle * RADPERDEG
    Ca = Cos(Radians)
    Sa = Sin(Radians)
    Tx = X * Ca - Y * Sa
    Y = Y * Ca + X * Sa
    X = Tx
End Sub

Sub PolToRec(Radius, Angle, X, Y)
    Dim Radians
    Radians = Angle * RADPERDEG
    X = Radius * Cos(Radians)
    Y = Radius * Sin(Radians)
End Sub

Sub RectoPol(X, Y, Radius, Angle)
    Dim Radians
    Radius = Sqr(X * X + Y * Y)
    If X = 0 Then
        Select Case Y
            Case Is > 0

```

```

        Angle = 90
    Case Is < 0
        Angle = -90
    Case Else
        Angle = 0
    End Select
ElseIf Y = 0 Then
    Select Case X
    Case Is < 0
        Angle = 180
    Case Else
        Angle = 0
    End Select
Else
    If X < 0 Then
        If Y > 0 Then
            Radians = Atn(Y / X) + PI
        Else
            Radians = Atn(Y / X) - PI
        End If
    Else
        Radians = Atn(Y / X)
    End If
    Angle = Radians / RADPERDEG
End If
End Sub

```

Notice that I've used the DefDbl A-Z statement in all of the forms and modules of this project to cause all variables to default to double-precision floating-point values. This program will run just fine if you delete these statements and let all variables default to Variants, but operation is slightly faster if all variables are defined as Double, as shown. I decided to depart in this one application from the Hungarian Notation I've adhered to fairly strictly elsewhere in the book. It's perfectly acceptable and works well, but if it confuses you, a good exercise would be to convert all variables and their declarations explicitly to carry Hungarian prefixes.

Unlike most of the applications in Part III, these graphics demonstration forms don't have menus; hence, they have no menu selections to access the help file. However, this is a convenient place to demonstrate how to connect the F1 key to this application's help file so that the user can activate the Help system at any time by pressing the F1 key. To do this, I simply set the App object's HelpFile property in Sub Main to the path and filename of my help file.

An alternative to defining the HelpFile property in your code is to set the filename and optionally the full path of the help file in your project file. To do this, choose Project Properties from the Project menu to display the Project Properties dialog box. In the Help File Name text box, enter the name of the help file, and leave the Project Help Context ID text box set to 0, which displays the Contents topic in the help file.

I set the program to start with Sub Main instead of one of the forms. To do this, I chose Project Properties from the Project menu to open the Project Properties dialog box. From the Startup Object drop-down list in this dialog box, I selected Sub Main. The Sub Main code block is very short; its purpose is to show both of the animation demonstration forms and set the App object's HelpFile property. The rest of the code in this module is a collection of handy procedures for rotating Cartesian coordinates around each of the three axes and for converting coordinates between rectangular and polar. These procedures are useful for three-dimensional graphics computations, such as those I use to display a spinning globe. We will get into that in more detail later, but first let's look at the animated clock form.

ANICLOCK.FRM

This form creates a real-time clock using only a single Line control and a Timer control. Figure 29-4 shows these two controls on the form at design time, and Figure 29-5 shows the clock at runtime.

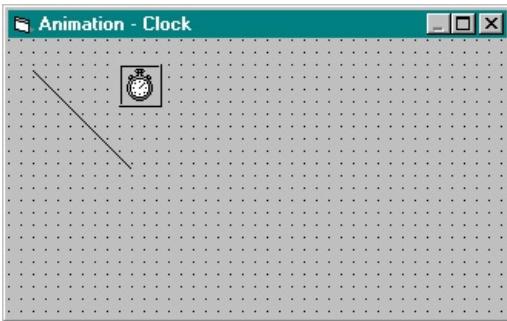


Figure 29-4. The clock form at design time, with only two controls.



Figure 29-5. The clock form at runtime.

You might be wondering how in the world all the straight-line elements of the clock face are drawn. The trick is to make 14 copies of the original Line control using the Load statement, with the endpoint-coordinate properties of each instance of this control array set to properly place each line on the clock face. Twelve of these copies are placed once, to mark the hour positions on the clock face, and three of the Line controls, the hands of the clock, are updated each second to give the illusion of movement.

Notice that no Line methods were used to create this clock and that no lines are ever erased directly by our application code. All the technical work of erasing and redrawing each hand as it moves is taken care of by Visual Basic when we update the endpoints of each Line control. Sometimes you don't need fancy graphics statements and commands—even a simple Line control can do amazing things.

You can change the appearance of the clock by adjusting property settings in the code. For example, you can create thinner or fatter lines by changing the setting of each Line control's BorderWidth property. To create this application, use the following table to add the appropriate controls, and set any nondefault properties as indicated.

ANICLOCK.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmClock</i>
Caption	<i>Animation - Clock</i>
MinButton	<i>False</i>
Timer	
Name	<i>tmrClock</i>
Interval	<i>100</i>
Line	
Name	<i>linClock</i>

Notice that I set the timer's Interval property to 100, which is $1/10$ of a second. It might seem that logic would dictate a setting of 1000 milliseconds, or once per second, as ideal for updating the hands of the clock. However, this timing interval is not precise because Visual Basic's timers are designed to provide a delay that is at least as long as the specified interval, but there is no guarantee that every delay will be of exactly the same duration. Because of the slight variability in delay intervals, occasionally the second hand will "miss a beat," observable as a jerky movement of the second hand. To solve this timing unpredictability, I chose to check at the rate of 10 times per second to see whether a new second has arrived. This results in a maximum error of approximately $1/10$ of a second in the precision of each movement of the second hand, which is well within the tolerance of smooth operation for visual perception. About 9 out of 10 times that the timer event procedure runs, it bails out immediately because the current second hasn't changed, wasting very little of the system's time.

Much of the work of *tmrClock*'s Timer event procedure is in recalculating and resetting the endpoint-coordinate properties X1, Y1, X2, and Y2. The source code for the AniClock form is shown here.

Source Code for ANICLOCK.FRM

```

Option Explicit
DefDbl A-Z           '=<<< NOTICE!!!

Private Sub Form_Load()
    Width = 4000
    Height = 4000
    Left = Screen.Width \ 2 - 4100
    Top = (Screen.Height - Height) \ 2
End Sub

Private Sub Form_Resize()
    Dim i, Angle
    Static Flag As Boolean
    If Flag = False Then
        Flag = True
        For i = 0 To 14
            If i > 0 Then Load linClock(i)
            linClock(i).Visible = True
            linClock(i).BorderWidth = 5
            linClock(i).BorderColor = RGB(0, 128, 0)
        Next i
    End If
    For i = 0 To 14
        Scale (-1, -1)-(1, 1)
        Angle = i * 2 * Atn(1) / 3
        linClock(i).x1 = 0.9 * Cos(Angle)
        linClock(i).y1 = 0.9 * Sin(Angle)
        linClock(i).x2 = Cos(Angle)
        linClock(i).y2 = Sin(Angle)
    Next i
End Sub

Private Sub tmrClock_Timer()
    Const HourHand = 0
    Const MinuteHand = 13

    Const SecondHand = 14
    Dim Angle
    Static LastSecond
    'Position hands only on the second
    If Second(Now) = LastSecond Then Exit Sub
    LastSecond = Second(Now)
    'Position hour hand

```

```

Angle = -0.5236 * (15 - (Hour(Now) + Minute(Now) / 60))
linClock(HourHand).x1 = 0
linClock(HourHand).y1 = 0
linClock(HourHand).x2 = 0.3 * Cos(Angle)
linClock(HourHand).y2 = 0.3 * Sin(Angle)
`Position minute hand
Angle = -0.1047 * (75 - (Minute(Now) + Second(Now) / 60))
linClock(MinuteHand).x1 = 0
linClock(MinuteHand).y1 = 0
linClock(MinuteHand).x2 = 0.7 * Cos(Angle)
linClock(MinuteHand).y2 = 0.7 * Sin(Angle)
`Position second hand
Angle = -0.1047 * (75 - Second(Now))
linClock(SecondHand).x1 = 0
linClock(SecondHand).y1 = 0
linClock(SecondHand).x2 = 0.8 * Cos(Angle)
linClock(SecondHand).y2 = 0.8 * Sin(Angle)
End Sub

```

ANIGLOBE.FRM

This form displays sequential images of a sphere's lines of latitude and longitude, creating the illusion of a spinning globe. The technique used here is an ImageList control that stores the sequential images and provides them for quick, animated copying to a PictureBox control. Figure 29-6 below shows the AniGlobe form during development, with Timer, PictureBox, and ImageList controls for creating the animation effect.

This form updates itself each time its *tmrGlobe* Timer event fires. For the first 15 iterations, a new image is drawn, using the various 3D graphics procedures you'll find in the ANIMATE.BAS module. As each image is drawn, it is quickly saved in an ImageList control declared at the module level. After all 15 images have been tucked away for safekeeping, the program begins to display them sequentially by updating the PictureBox's Picture property to refer to each image, as shown in Figure 29-7 below. This image transfer operation is fast and smooth, providing a great general technique for creating animated sequences.

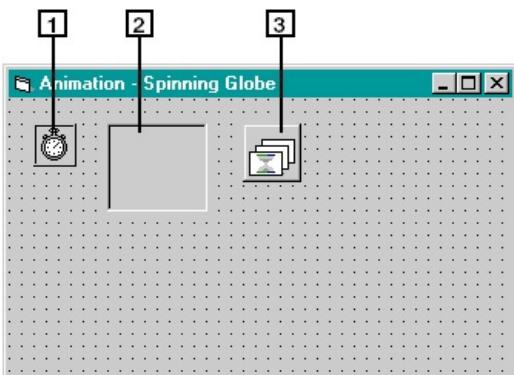


Figure 29-6. The AniGlobe form at development time.

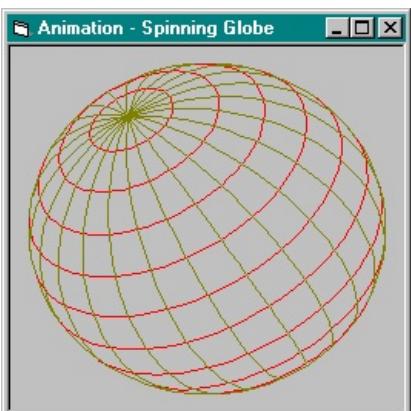


Figure 29-7. The AniGlobe form in action.

The following table specifies the AniGlobe form's settings.

ANIGLOBE.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmGlobe</i>
	Caption	<i>Animation - Spinning Globe</i>
Timer		
1	Name	<i>tmrGlobe</i>
	Interval	1
PictureBox		
2	Name	<i>picGlobe</i>
	AutoRedraw	<i>True</i>
ImageList		
3	Name	<i>imlGlobe</i>

* The number in the ID No. column corresponds to the number in Figure 29-6 that identifies the location of the object on the form.

In the following source code, I've isolated constants that define the forward and sideways tilt angles of the globe. Try changing the *TILTSOUTH* and *TILTEAST* constants to see the dramatic changes in the appearance of the spinning globe.

The ImageList control is especially useful when associated with other controls, such as the ListView, ToolBar, TabStrip, and TreeView controls. For more information about this powerful image manipulation tool, see the Visual Basic online documentation.

The source code for the AniGlobe form is shown below.

Source Code for ANIGLOBE.FRM

```

Option Explicit
DefDbl A-Z           `<<< NOTICE!!!

Const TILTSOUTH = 47
Const TILTEAST = -37

Private Sub Form_Load()
    Width = 4000
    Height = 4000
    Left = Screen.Width \ 2 + 100
    Top = (Screen.Height - Height) \ 2
End Sub

Private Sub tmrGlobe_Timer()
    Dim Lat, Lon, Radians
    Dim R, A, i
    Dim x1, y1, x2, y2
    Dim Xc(72), Yc(72), Zc(72)
    Dim imgX As ListImage
    Static ImageIndex, ImageNum
    Select Case ImageNum
        `Pump next image to display
    Case -1

```

```

        ImageIndex = (ImageIndex Mod 15) + 1
        Set picGlobe.Picture = imlGlobe.ListImages _
            (ImageIndex).Picture
        Exit Sub
    `Initialize PictureBox
Case 0
    picGlobe.Move 0, 0, ScaleWidth, ScaleHeight
    picGlobe.Scale (-1.1, 1.1)-(1.1, -1.1)
    Caption = "Animation Dear John, How Do I... PREPARATION"
    ImageNum = ImageNum + 1
    Exit Sub
`Set flag when last image has been
`drawn and saved in image list
Case 16
    Caption = "Animation - Spinning Globe"
    ImageNum = -1
    Exit Sub
End Select
`Erase any previous picture in PictureBox control
Set picGlobe.Picture = Nothing
`Draw edge of globe
picGlobe.ForeColor = vbBlue
For i = 0 To 72
    PolToRec 1, i * 5, Xc(i), Yc(i)
Next i
For i = 1 To 72
    picGlobe.Line (Xc(i - 1), Yc(i - 1))-(Xc(i), Yc(i))
Next i
`Calculate and draw latitude lines
picGlobe.ForeColor = vbRed
For Lat = -75 To 75 Step 15
    `Convert latitude to radians
    Radians = Lat * RADPERDEG
`Draw circle size based on latitude
    For i = 0 To 72
        PolToRec Cos(Radians), i * 5, Xc(i), Zc(i)
        Yc(i) = Sin(Radians)
        `Tilt globe's north pole toward us
        RotateX Xc(i), Yc(i), Zc(i), TILTSOUTH

        `Tilt globe's north pole to the right
        RotateY Xc(i), Yc(i), Zc(i), TILTEAST
    Next i
    `Draw front half of rotated circle
    For i = 1 To 72
        If Zc(i) >= 0 Then
            picGlobe.Line (Xc(i - 1), Yc(i - 1))-(Xc(i), Yc(i))
        End If
    Next i
Next Lat
`Calculate and draw longitude lines
picGlobe.ForeColor = vbBlue
For Lon = 0 To 165 Step 15
    `Start with xy-plane circle
    For A = 0 To 72
        PolToRec 1, A * 5, Xc(A), Yc(A)
        Zc(A) = 0
    Next A
    `Rotate points for current line of longitude
    For i = 0 To 72
        RotateY Xc(i), Yc(i), Zc(i), Lon + ImageNum
        `Tilt globe's north pole toward us
        RotateX Xc(i), Yc(i), Zc(i), TILTSOUTH
        `Tilt globe's north pole to the right
    Next i
Next Lon

```

```
    RotateY Xc(i), Yc(i), Zc(i), TILTEAST
Next i
`Draw front half of rotated circle
For i = 1 To 72
    If Zc(i) >= 0 Then
        picGlobe.Line (Xc(i - 1), Yc(i - 1))-(Xc(i), Yc(i))
    End If
Next i
Next Lon
`Update PictureBox state
picGlobe.Refresh
picGlobe.Picture = picGlobe.Image
`Add this image to our image list
Set imgX = imlGlobe.ListImages.Add(, , picGlobe.Picture)
`Prepare to draw next image
ImageNum = ImageNum + 1
End Sub
```

The Lottery Application

The Lottery application is modeled after the Colorado lottery, in which a cylindrical basket tumbles numbered Ping-Pong balls. With minor modifications, this application can model other similar games of chance. The concept is fairly simple: 42 Ping-Pong balls, numbered 1 through 42, are tossed into a basket and jumbled. Six balls are then selected, one at a time, in a random manner. The six numbers drawn from the set of numbers 1 through 42 represent the winning ticket. The prize money can be a huge amount, but the chance of hitting all six numbers is extremely small. The only guaranteed winner here is the state. In the Lottery application, each time you click the Next Ball command button, one of the balls is selected and placed at the bottom of the screen. Figure 29-8 below shows the application in action, after three of the numbers have been selected. After you've finished selecting six balls, you can see how lucky you are—I've added an option to simulate the purchase of 1000 lottery tickets so that you can see how hard it is to hit the jackpot even if you buy a lot of tickets. As you'll discover, it's very rare that you'll match even five out of the six numbers.

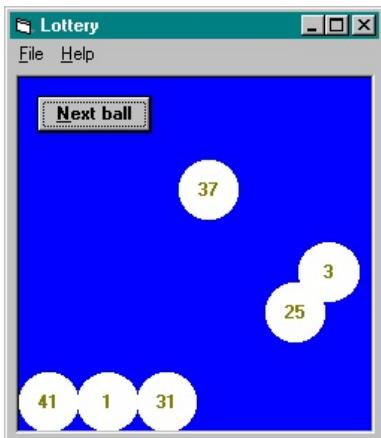


Figure 29-8. The Lottery program in action.

This application demonstrates a Random object I've created that generates random numbers with a much greater sequence length than the generator built into Visual Basic. The application also demonstrates one of the simplest methods of animation, which is simply to redraw a picture repeatedly with enough speed to make it appear that the action is continuous.

In addition to the main Lottery form, which displays the tumbling Ping-Pong balls, this project contains a class module named RANDOM.CLS and the standard About form I described in Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)." Figure 29-9 shows the Lottery project list.

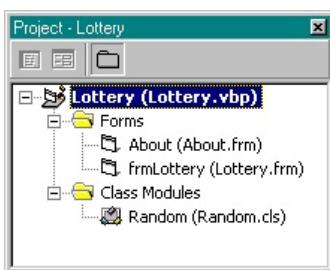


Figure 29-9. The Lottery project list.

LOTTERY.FRM

LOTTERY.FRM is the main startup form for this application. The *picTumble* PictureBox control displays the tumbling balls and the selected balls, the two command buttons control the action, and the *tmrPingPong* Timer control triggers each update of the action. Figure 29-10 shows this form during development, and the tables and source code below define the form's design.

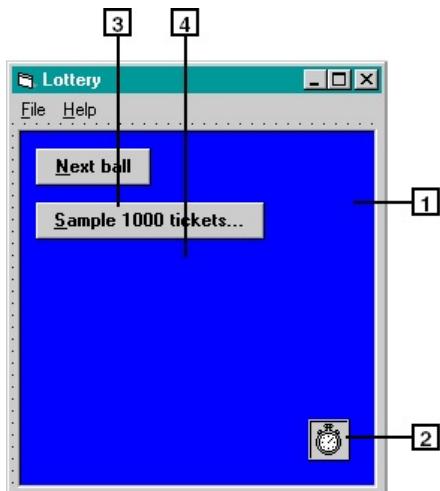


Figure 29-10. The Lottery form during development.

LOTTERY.FRM Menu Editor Settings

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&Open Dear John, How Do I...	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &As Dear John, How Do I...	mnuSaveAs	1	<i>False</i>
—	mnuFileDash1	1	<i>True</i>
E&xit	mnuExit	1	<i>True</i>
&Help	mnuHelp	0	<i>True</i>
&Contents	mnuContents	1	<i>True</i>
&Search for Help on Dear John, How Do I...	mnuSearch	1	<i>True</i>
—	mnuHelpDash1	1	<i>True</i>
&About Dear John, How Do I...	mnuAbout	1	<i>True</i>

LOTTERY.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmLottery</i>
	Caption	<i>Lottery</i>
PictureBox		
1	Name	<i>picTumble</i>
	AutoRedraw	<i>True</i>
	BackColor	<i>&H00FF0000&</i>
	Height	<i>3600</i>
	Width	<i>3600</i>
Timer		

2	Name	<i>tmrPingPong</i>
	Interval	50
CommandButton		
3	Name	<i>cmdNextBall</i>
	Caption	<i>&Next ball</i>
<i>CommandButton</i>		
4	Name	<i>cmdSample</i>
	Caption	<i>&Sample 1000 ticketsDear John, How Do I...</i>

* The number in the ID No. column corresponds to the number in Figure 29-10 that identifies the location of the object on the form.

Source Code for LOTTERY.FRM

```

Option Explicit

Private Declare Function WinHelp _
Lib "user32" Alias "WinHelpA" ( _
    ByVal hwnd As Long, _
    ByVal lpHelpFile As String, _
    ByVal wCommand As Long, _
    ByVal dwData As Long _
) As Long

Const MAXNUM = 42

Dim intPPBall(6) As Integer
Dim randDemo As New Random

Private Sub cmdNextBall_Click()
    Dim intI As Integer
    Dim intJ As Integer
    'Set command button caption
    cmdNextBall.Caption = "&Next ball"
    cmdSample.Visible = False
    'Get current count of selected balls
    intI = intPPBall(0)
    'If all balls were grabbed, start over
    If intI = 6 Then
        For intI = 0 To 6
            intPPBall(intI) = 0
        Next intI
        Exit Sub
    End If
    'Select next unique Ping-Pong ball
    GrabNext intPPBall()
    'Change command button caption,
    'and show sample command button
    If intPPBall(0) = 6 Then
        cmdNextBall.Caption = "Start &over"
        cmdSample.Visible = True
    End If
End Sub

Private Sub cmdSample_Click()
    Dim intI As Integer
    Dim intJ As Integer
    Dim intK As Integer
    Dim intN As Integer
    Dim intTicket(6) As Integer

```

```

Dim Hits(6) As Integer
Dim strMsg As String
`Display hourglass mouse pointer
MousePointer = vbHourglass
`Now simulate a thousand "quick pick" tickets
For intI = 1 To 1000
    `Generate a ticket
    intTicket(0) = 0
    For intJ = 1 To 6
        GrabNext intTicket()
    Next intJ
    `Tally the hits
    intN = 0
    For intJ = 1 To 6
        For intK = 1 To 6
            If intTicket(intJ) = intPPBall(intK) Then
                intN = intN + 1
            End If
        Next intK
    Next intJ
    `Update statistics
    Hits(intN) = Hits(intN) + 1
Next intI
`Display default mouse pointer
MousePointer = vbDefault
`Display summarized statistics
strMsg = "Sample of 1000 ticketsDear John, How Do I... " & vbCrLf & vbCrLf
strMsg = strMsg & Space$(10) & "Hits      Tally" & vbCrLf
For intI = 0 To 6
    strMsg = strMsg & Space$(12) & Format$(intI) & Space$(6)
    strMsg = strMsg & Format$(Hits(intI)) & vbCrLf
Next intI
MsgBox strMsg, , "Lottery"
End Sub

Private Sub Form_Load()
    `Seed new random numbers
    Randomize
    randDemo.Shuffle Rnd
    `Set range of random integers
    randDemo.MinInt = 1
    randDemo.MaxInt = MAXNUM
    `Center form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
    `Hide sample command button for now
    cmdSample.Visible = False
    `Prepare tumble animation
    picTumble.Scale (0, 0)-(12, 12)
    picTumble.FillStyle = vbSolid
    picTumble.FillColor = vbWhite
    picTumble.ForeColor = vbRed
End Sub

Private Sub picTumble_Paint()
    Dim intI As Integer
    Dim sngX As Single
    Dim sngY As Single
    Dim strN As String
    `Erase previous tumble animation
    picTumble.Cls
    For intI = 1 To 6
        `Determine whether ball has been selected
        If intPPBall(intI) > 0 Then
            sngX = intI * 2 - 1
            sngY = 11
        End If
    Next intI
    `Draw the balls
    For intI = 1 To 6
        If intPPBall(intI) > 0 Then
            picTumble.PSet (sngX, sngY)
        End If
    Next intI
End Sub

```

```

        strN = Format$(intPPBall(intI))
    Else
        sngX = Rnd * 10 + 1
        sngY = Rnd * 8 + 3
        strN = Format$(randDemo.RandomInt)
    End If
    'Draw each Ping-Pong ball
    picTumble.Circle (sngX, sngY), 1, vbWhite
    picTumble.CurrentX = sngX - picTumble.TextWidth(strN) / 2
    picTumble.CurrentY = sngY - picTumble.TextHeight(strN) / 2
    'Label each Ping-Pong ball
    picTumble.Print strN
    Next intI
End Sub

Private Sub tmrPingPong_Timer()
    picTumble_Paint
End Sub

Private Sub mnuAbout_Click()
    'Set properties
    About.Application = "Lottery"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    WinHelp hwnd, App.Path & "\..\..\Help\Mvbdw.hlp", _
        cdlHelpContents, 0
End Sub

Private Sub mnuSearch_Click()
    WinHelp hwnd, App.Path & "\..\..\Help\Mvbdw.hlp", _
        cdlHelpPartialKey, 0
End Sub

Private Sub GrabNext(intAry() As Integer)
    Dim intI As Integer
    Dim intJ As Integer
    'Store index in first array element
    intAry(0) = intAry(0) + 1
    intI = intAry(0)
    'Get next unique Ping-Pong ball number
    Do
        intAry(intI) = randDemo.RandomInt
        If intI > 1 Then
            For intJ = 1 To intI - 1
                If intAry(intI) = intAry(intJ) Then
                    intAry(intI) = 0
                End If
            Next intJ
        End If
    Loop Until intAry(intI)
End Sub

```

NOTE

The WinHelp API function is used to activate the help file when the appropriate menu items are clicked. In some of the applications in Part III, I will add a CommonDialog control to the project to provide the connection to the help file, even in cases in which the menu is identical to the menu shown here. Both techniques produce identical results, and it's largely a matter of personal preference as to the method you choose.

Notice in the Form_Load event procedure that I use an object of type Random. This type of object is defined by the Random class module, which I'll describe next.

RANDOM.CLS

The Random class module provides the template for creating objects of type Random. At the core of the Random object is a technique that greatly expands the sequence length of the random-number generator built into Visual Basic. I've used an array of type Double to add a shuffling and mixing action to the generation of the random numbers.

The following public properties are defined in RANDOM.CLS:

- MinInt: Minimum value for the range of generated random integers
- MaxInt: Maximum value for the range of generated random integers
- Random: Random number in the range 0 through 1
- RandomInt: Random integer in the range defined by MinInt and MaxInt

The one public method defined in RANDOM.CLS is Shuffle, which initializes the random-number sequence.

The random-number generator in Visual Basic is very good, but I've heard of some concerns. First, it's not at all clear how to initialize Visual Basic's random-number generator to a repeatable sequence. OK, I'll let you in on a little secret—there is a simple, albeit tricky, way to reinitialize Visual Basic's random-number generator to a repeatable sequence. You need to call Randomize immediately after passing a negative value to the Rnd function, as in the following example:

```
Randomize Rnd (-7)
```

Every time you pass -7 to these two functions, as shown, you'll initialize Visual Basic's random numbers to the same sequence. I've used this technique, modified slightly, in the Shuffle method of my Random object to initialize the object's sequence. For flexibility, if you pass a negative value to Shuffle, a repeatable sequence is initialized. A positive or zero value results in a completely unpredictable sequence.

Another concern I've heard, especially from cryptographers, is that the random numbers generated by Visual Basic don't take into account such phenomena as subtle patterns and entropy, reducing the value of the generated sequences for high-quality cryptography work. My Random object maintains an array of double-precision numbers that effectively shuffle and randomize Visual Basic's sequence by many orders of magnitude, while maintaining a good distribution of numbers in the range 0 through 1. Study the Random public property procedure to see how this is accomplished.

The RandomInt property procedure modifies a value returned by the Random procedure to provide an integer in the range defined by the user-set properties MinInt and MaxInt. The distribution of these pseudorandom integers is as good as the distribution of the values returned by Random.

There are two private procedures in the Random class module. The Zap procedure initializes the array and its indexes to a known state during the initialization performed by the Shuffle method. The Stir private method helps warm up the pseudorandom number generator.

Source Code for RANDOM.CLS

```
Option Explicit
```

```
Const ARYCNT = 17
```

```

`Two simple R/W properties
Public MinInt As Long
Public MaxInt As Long

`Module-level variables
Private dblSeed(ARYCNT - 1) As Double
Private intP As Integer
Private intQ As Integer

`Method
Public Sub Shuffle(dblX As Double)
    Dim strN As String
    Dim intI As Integer
    Zap
    strN = Str$(dblX)
    For intI = 1 To Len(strN)
        Stir 1 / Asc(Mid(strN, intI, 1))
    Next intI
    Randomize Rnd(mdblSeed(intP) * Sgn(dblX))
    For intI = 1 To ARYCNT * 2.7
        Stir Rnd
    Next intI
End Sub

Property Get Random() As Double
    intP = (intP + 1) Mod ARYCNT
    intQ = (intQ + 1) Mod ARYCNT
    dblSeed(intP) = dblSeed(intP) + dblSeed(intQ) + Rnd
    dblSeed(intP) = dblSeed(intP) - Int(dblSeed(intP))
    Random = dblSeed(intP)
End Property

Property Get RandomInt() As Long
    RandomInt = Int(Random() * (MaxInt - MinInt + 1)) + MinInt
End Property

Private Sub Zap()
    Dim intI As Integer
    For intI = 1 To ARYCNT - 1
        dblSeed(intI) = 1 / intI
    Next intI
    intP = ARYCNT \ 2
    intQ = ARYCNT \ 3
    If intP = intQ Then
        intP = intP + 1
    End If
End Sub

Private Sub Stir(dblX As Double)
    intP = (intP + 1) Mod ARYCNT
    intQ = (intQ + 1) Mod ARYCNT
    dblSeed(intP) = dblSeed(intP) + dblSeed(intQ) + dblX
    dblSeed(intP) = dblSeed(intP) - Int(dblSeed(intP))
End Sub

```

The MySaver Application

The MySaver application expands on the examples presented in Chapter 25, "[Screen Savers](#)." I've added a larger set of graphics options that let the user set a wide variety of effects without having to add a lot of code. Figure 29-11 shows this screen saver in action, although you should try the many combinations of settings to get the full effect of its many graphics variations.

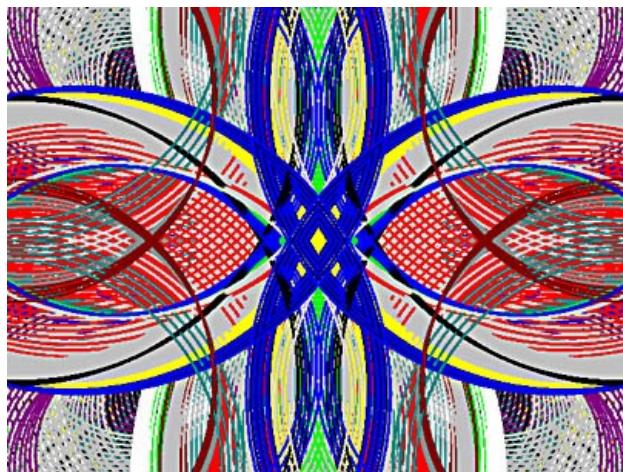


Figure 29-11. The MySaver application in action.

Perhaps the most important difference between this screen saver and the examples in [Chapter 25](#) is the addition of code to automatically show the screen saver in the little preview window that looks like a desktop on a small monitor. The operating system passes a command line parameter of `/P nnnn` when this type of preview is indicated, where the `nnnn` value is the hWnd of the small window designated for the preview output. It takes several API functions to correctly set up and handle this type of output, which this application demonstrates. At several places in the code, you'll see tests of the global variable `gblnShow` to check whether the graphics are currently expected to be displayed in the "Show" mode (the normal full-screen operation of the screen saver) or in the "Preview" mode (in the small preview window). These two modes are treated slightly differently, hence the need for these checks. Figure 29-12 shows the MySaver screen saver as it's displayed in the small preview window.



Figure 29-12. The MySaver screen saver in action in the small preview window.

As shown in Figure 29-13, this project contains two form modules and one standard module. MYSAVER.BAS contains the Sub Main starting point and associated control code, MYSAVER.FRM is the main display for the screen saver graphics, and MYSETUP.FRM is activated when the user clicks the

Settings button on the Screen Saver tab of the Display Properties dialog box.

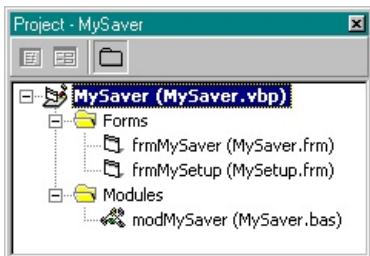


Figure 29-13. The MySaver project list.

MYSAVER.BAS

The starting point, Sub Main, is in this standard module, along with several API function declarations and calls used to control the location of the graphics output, depending on whether the screen saver is to output full-screen graphics, or to output only to the small preview screen in the image of the monitor in the Display Properties dialog box. This module's main purpose is to decode the command line parameters passed to this screen saver by the system upon activation, and to take appropriate action.

Here are the contents of MYSAVER.BAS:

```

`MySaver.bas
Option Explicit

`Rectangle data structure
Private Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type

`Constants for some API functions
Private Const WS_CHILD = &H40000000
Private Const GWL_HWNDPARENT = (-8)
Private Const GWL_STYLE = (-16)
Private Const HWND_TOP = 0&
Private Const SWP_NOZORDER = &H4
Private Const SWP_NOACTIVATE = &H10
Private Const SWP_SHOWWINDOW = &H40

`--- API functions
Private Declare Function GetClientRect _
Lib "user32" (
    ByVal hwnd As Long,
    lpRect As RECT
) As Long

Private Declare Function GetWindowLong _
Lib "user32" Alias "GetWindowLongA" (
    ByVal hwnd As Long,
    ByVal nIndex As Long
) As Long

Private Declare Function SetWindowLong _
Lib "user32" Alias "SetWindowLongA" (
    ByVal hwnd As Long,
    ByVal nIndex As Long,
    ByVal dwNewLong As Long
) As Long

```

```

Private Declare Function SetWindowPos _  

Lib "user32" ( _  

    ByVal hwnd As Long, _  

    ByVal hWndInsertAfter As Long, _  

    ByVal X As Long, _  

    ByVal Y As Long, _  

    ByVal cx As Long, _  

    ByVal cy As Long, _  

    ByVal wFlags As Long _  

) As Long

Private Declare Function SetParent _  

Lib "user32" ( _  

    ByVal hWndChild As Long, _  

    ByVal hWndNewParent As Long _  

) As Long

`Global Show/Preview flag
Public gblnShow As Boolean

`Module level variables
Private mlnghwnd As Long
Private recDisplay As RECT

`Starting pointDear John, How Do I... .
Public Sub Main()
    Dim strCmd As String
    Dim strTwo As String
    Dim lngStyle As Long
    Dim lngPreviewHandle As Long
    Dim lngParam As Long
    `Process the command line
    strCmd = UCASE(Trim(Command))
    strTwo = Left(strCmd, 2)
    Select Case strTwo
        `Preview screen saver in small display window
        Case "/P"
            `Get HWND of display window
            mlnghwnd = Val(Mid(strCmd, 4))
            `Get display rectangle dimensions
            GetClientRect mlnghwnd, recDisplay
            `Load form for preview
            gblnShow = False
            Load frmMySaver
            `Get HWND for display form
            lngPreviewHandle = frmMySaver.hwnd
            `Get current window style
            lngStyle = GetWindowLong(lngPreviewHandle, GWL_STYLE)
            `Append "WS_CHILD" style to the current window style
            lngStyle = lngStyle Or WS_CHILD
            `Add new style to display window
            SetWindowLong lngPreviewHandle, GWL_STYLE, lngStyle
            `Set display window as parent window
            SetParent lngPreviewHandle, mlnghwnd
            `Save the parent hWnd in the display form's window structure.
            SetWindowLong lngPreviewHandle, GWL_HWNDFPARENT, _
                mlnghwnd
            `Preview screensaver in the windowDear John, How Do I...
            SetWindowPos lngPreviewHandle, _
                HWND_TOP, 0&, 0&, recDisplay.Right, recDisplay.Bottom, _
                SWP_NOZORDER Or SWP_NOACTIVATE Or SWP_SHOWWINDOW
            Exit Sub
        `Allow user to set up screen saver
        Case "/C"

```

```

        Load frmMySetup
        Exit Sub
    'Password - not implemented here
    Case "/A"
        MsgBox "No password is necessary for this Screen Saver", _
            vbInformation, "Password Information"
        Exit Sub
    'Show screen saver in normal full screen mode
    Case "/S"
        gblnShow = True
        Load frmMySaver
        frmMySaver.Show
        Exit Sub
    'Unknown command line parameters
    Case Else
        Exit Sub
    End Select
End Sub

```

MYSAVER.FRM

The MYSAYER.FRM form is where all the graphical screen saver action takes place. Notice that I turned off all visible parts of the form—for example, setting the MinButton and MaxButton properties to *False*, and the BorderStyle property to *0 - None*. This lets the form provide a drawing surface covering the entire screen.

The only control on this form, as shown below in Figure 29-14, is a Timer control. The graphics update happens in a continuous loop during the form's Paint event. Visual Basic generates an error if your form tries to unload itself from within the Paint event, so the timer is a tricky way to allow the form to unload quickly after exiting from the graphics loop within the Paint event. I wouldn't normally suggest structuring a program to continuously loop within the Paint event in this way, but I did it because it's slightly faster than letting a timer trigger each update of the graphics and because the system doesn't interact with the user while a screen saver is running.

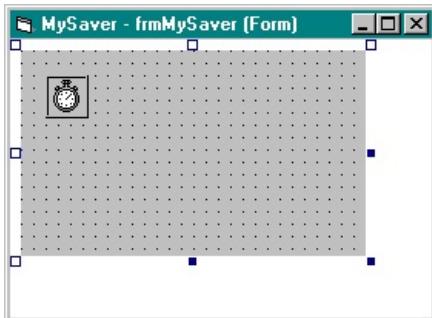


Figure 29-14. MYSAYER.FRM during development.

The following table and source code define the form's design.

MYSAYER.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmMySaver</i>
BorderStyle	<i>0 - None</i>
ControlBox	<i>False</i>
MaxButton	<i>False</i>
MinButton1	<i>False</i>

Timer

Name	<i>tmrExitNotify</i>
Interval	1
Enabled	<i>False</i>

Source Code for MYSAYER.FRM

```

`MySaver.frm
Option Explicit

`API function to hide/show the mouse pointer
Private Declare Function ShowCursor _
Lib "user32" ( _
    ByVal bShow As Long _
) As Long

`API function to signal activity to system
Private Declare Function SystemParametersInfo _ 
Lib "user32" Alias "SystemParametersInfoA" ( _
    ByVal uAction As Long, _
    ByVal uParam As Long, _
    ByRef lpvParam As Any, _
    ByVal fuWinIni As Long _
) As Long

`Constant for API function
Private Const SPI_SETSCREENSAVEACTIVE = 17

`Declare module-level variables
Dim mlngXai As Long
Dim mlngYai As Long
Dim mlngXbi As Long
Dim mlngYbi As Long
Dim mlngLineCount As Long
Dim mlngLineWidth As Long
Dim mlngActionType As Long
Dim mlngXmax As Long
Dim mlngYmax As Long
Dim mlngInc As Long
Dim mlngColorNum() As Long
Dim mlngDx1() As Double
Dim mlngDx2() As Double
Dim mlngDy1() As Double
Dim mlngDy2() As Double
Dim mlngXa() As Long
Dim mlngXb() As Long
Dim mlngYa() As Long
Dim mlngYb() As Long
Dim mbInQuit As Boolean

Private Sub Form_Load()
    Dim lngRet As Long
    `Tell system that screen saver is active
    lngRet = SystemParametersInfo(
        SPI_SETSCREENSAVEACTIVE, 0, ByVal 0&, 0)
    `Go full screen if not in preview mode
    If gblnShow = True Then
        Me.WindowState = vbMaximized
    End If
End Sub

```

```

Private Sub Form_Paint()
    Dim lngX As Long
    'In preview mode, set AutoRedraw to True
    If gblnShow = False Then
        Me.AutoRedraw = True
    End If
    'Create different display each time
    Randomize
    'Set control values
    mlngInc = 5
    mlngXmax = 300
    mlngYmax = 300
    'Get current user settings from Registry
    mlngActionType = Val(GetSetting("MySaver", "Options", _
        "Action", "1"))
    mlngLineCount = Val(GetSetting("MySaver", "Options", _
        "LineCount", "1"))
    mlngLineWidth = Val(GetSetting("MySaver", "Options", _
        "LineWidth", "1"))
    'Initialize graphics
    With Me
        .BackColor = vbBlack
        .DrawWidth = mlngLineWidth
    End With
    Scale (-mlngXmax, -mlngYmax)-(mlngXmax, mlngYmax)
    'Size arrays
    ReDim mlngColorNum(0 To mlngLineCount)
    ReDim mlngXa(1 To mlngLineCount), mlngXb(1 To mlngLineCount)
    ReDim mlngYa(1 To mlngLineCount), mlngYb(1 To mlngLineCount)
    'Action types above 4 are a little different
    If mlngActionType < 5 Then
        ReDim mlngDx1(1 To mlngLineCount), _
            mlngDx2(1 To mlngLineCount)
        ReDim mlngDy1(1 To mlngLineCount), _
            mlngDy2(1 To mlngLineCount)
    Else
        ReDim mlngDx1(0), mlngDx2(0)
        ReDim mlngDy1(0), mlngDy2(0)
        mlngDx1(0) = Rnd * mlngInc
        mlngDx2(0) = Rnd * mlngInc
        mlngDy1(0) = Rnd * mlngInc
        mlngDy2(0) = Rnd * mlngInc
    End If
    'Hide mouse pointer, unless in preview mode
    If gblnShow = True Then
        lngX = ShowCursor(False)
    End If
    'Do main processing as a loop
    Do
        'Update display
        DoGraphics
        'Yield execution
        DoEvents
    Loop Until mbInQuit = True
    'Show mouse pointer, unless in preview mode
    If gblnShow = True Then
        lngX = ShowCursor(True)
    End If
    'Can't quit in this context; let timer do it
    tmrExitNotify.Enabled = True
End Sub

Private Sub Form_QueryUnload(Cancel As Integer, _
    UnloadMode As Integer)

```

```

`Using End here appears to prevent memory leaks
End
End Sub

Private Sub Form_Click()
    `Quit if mouse is clicked, unless in preview mode
    If gblnShow = True Then mblnQuit = True
End Sub

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    `Quit if any key is pressed, unless in preview mode
    If gblnShow = True Then mblnQuit = True
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Static sngTimer As Single
    `Bail out quickly if in preview mode
    If gblnShow = False Then Exit Sub
    `Quit any time after first .25 seconds
    If sngTimer = 0 Then
        sngTimer = Timer
    ElseIf Timer > sngTimer + 0.25 Then
        mblnQuit = True
    End If
End Sub

Private Sub tmrExitNotify_Timer()
    Dim lngRet As Long
    `Tell system that screen saver is done
    lngRet = SystemParametersInfo( _
        SPI_SETSCREENSAVEACTIVE, 1, ByVal 0&, 0)
    `Time to quit
    End
End Sub

`~~~~~
`This is where the real graphics drawing takes place
`~~~~~

Sub DoGraphics()
    Dim intI
    Static dblColorTime As Double
    `Shuffle line colors every so often
    If Timer > dblColorTime Then
        ColorReset
        If mlngLineCount < 5 Then
            dblColorTime = Timer + mlngLineCount * Rnd + 0.3
        Else
            dblColorTime = Timer + 5 * Rnd + 0.3
        End If
    End If
    `Process based on count of lines
    For intI = 1 To mlngLineCount
        `Handle action types below 5 with special procedures
        If mlngActionType < 5 Then
            `Keep ends of lines within bounds
            If mlngXa(intI) <= 0 Then
                mlngDx1(intI) = mlngInc * Rnd
            End If
            If mlngXb(intI) <= 0 Then
                mlngDx2(intI) = mlngInc * Rnd
            End If
            If mlngYa(intI) <= 0 Then
                mlngDy1(intI) = mlngInc * Rnd
            End If
        End If
    Next
End Sub

```

```

        End If
        If mlnyYb(intI) <= 0 Then
            mlnyDy2(intI) = mlnyInc * Rnd
        End If
        If mlnyXa(intI) >= mlnyXmax Then
            mlnyDx1(intI) = -mlnyInc * Rnd
        End If
        If mlnyXb(intI) >= mlnyXmax Then
            mlnyDx2(intI) = -mlnyInc * Rnd
        End If
        If mlnyYa(intI) >= mlnyYmax Then
            mlnyDy1(intI) = -mlnyInc * Rnd
        End If
        If mlnyYb(intI) >= mlnyYmax Then
            mlnyDy2(intI) = -mlnyInc * Rnd
        End If
        `Increment the coordinates of the line endpoints
        mlnyXa(intI) = mlnyXa(intI) + mlnyDx1(intI)
        mlnyXb(intI) = mlnyXb(intI) + mlnyDx2(intI)
        mlnyYa(intI) = mlnyYa(intI) + mlnyDy1(intI)
        mlnyYb(intI) = mlnyYb(intI) + mlnyDy2(intI)
        `Draw each line with a unique color
        ForeColor = mlnyColorNum(intI)
    Else
        `Set action types 5 and 6 with the same color
        ForeColor = mlnyColorNum(0)
    End If
    `Draw lines according to action type
    Select Case mlny ActionType
    Case 1
        Line (mlnyXa(intI), mlnyYa(intI)) - _
              (mlnyXb(intI), mlnyYb(intI))
        Line (-mlnyXa(intI), -mlnyYa(intI)) - _
              (-mlnyXb(intI), -mlnyYb(intI))
        Line (-mlnyXa(intI), mlnyYa(intI)) - _
              (-mlnyXb(intI), mlnyYb(intI))
        Line (mlnyXa(intI), -mlnyYa(intI)) - _
              (mlnyXb(intI), -mlnyYb(intI))
    Case 2
        Line (mlnyXa(intI), mlnyYa(intI)) - _
              (mlnyXb(intI), mlnyYb(intI)), , B
        Line (-mlnyXa(intI), -mlnyYa(intI)) - _
              (-mlnyXb(intI), -mlnyYb(intI)), , B
        Line (-mlnyXa(intI), mlnyYa(intI)) - _
              (-mlnyXb(intI), mlnyYb(intI)), , B
        Line (mlnyXa(intI), -mlnyYa(intI)) - _
              (mlnyXb(intI), -mlnyYb(intI)), , B
    Case 3
        Circle (mlnyXa(intI), mlnyYa(intI)), _
                 mlnyXb(intI)
        Circle (-mlnyXa(intI), -mlnyYa(intI)), _
                 mlnyXb(intI)
        Circle (-mlnyXa(intI), mlnyYa(intI)), _
                 mlnyXb(intI)
        Circle (mlnyXa(intI), -mlnyYa(intI)), _
                 mlnyXb(intI)
    Case 4
        Line (mlnyXa(intI), mlnyYa(intI)) - _
              (mlnyXb(intI), -mlnyYb(intI))
        Line -(-mlnyXa(intI), -mlnyYa(intI))
        Line -(-mlnyXb(intI), mlnyYb(intI))
        Line -(mlnyXa(intI), mlnyYa(intI))
    `Handle action types above 4 a little differently
    Case 5, 6

```

```

        If mlngActionType = 5 Then
            Line (mlngXa(intI), mlngYa(intI))-_
                  (mlngXb(intI), mlngYb(intI)), _
                  BackColor
        Else
            Line (mlngXa(intI), mlngYa(intI))-_
                  (mlngXb(intI), mlngYb(intI)), _
                  BackColor, B
        End If
        If mlngXai <= -mlngXmax Then
            mlngDx1(0) = mlngInc * Rnd + 1
        End If
        If mlngXbi <= -mlngXmax Then
            mlngDx2(0) = mlngInc * Rnd + 1
        End If
        If mlngYai <= -mlngYmax Then
            mlngDy1(0) = mlngInc * Rnd + 1
        End If
        If mlngYbi <= -mlngYmax Then
            mlngDy2(0) = mlngInc * Rnd + 1
        End If
        If mlngXai >= mlngXmax Then
            mlngDx1(0) = -mlngInc * Rnd + 1
        End If
        If mlngXbi >= mlngXmax Then
            mlngDx2(0) = -mlngInc * Rnd + 1
        End If
        If mlngYai >= mlngYmax Then
            mlngDy1(0) = -mlngInc * Rnd + 1
        End If
        If mlngYbi >= mlngYmax Then
            mlngDy2(0) = -mlngInc * Rnd + 1
        End If
        mlngXai = mlngXai + mlngDx1(0)
        mlngXbi = mlngXbi + mlngDx2(0)
        mlngYai = mlngYai + mlngDy1(0)
        mlngYbi = mlngYbi + mlngDy2(0)
        mlngXa(intI) = mlngXai
        mlngXb(intI) = mlngXbi
        mlngYa(intI) = mlngYai
        mlngYb(intI) = mlngYbi
        If mlngActionType = 5 Then
            Line (mlngXa(intI), mlngYa(intI))-_
                  (mlngXb(intI), mlngYb(intI))
        Else
            Line (mlngXa(intI), mlngYa(intI))-_
                  (mlngXb(intI), mlngYb(intI)), , B
        End If
    End Select
    Next intI
End Sub

Sub ColorReset()
    Dim intI
    `Randomize set of colors
    If mlngActionType <= 4 Then
        For intI = 1 To mlngLineCount
            mlngColorNum(intI) =
                RGB(Rnd * 256, Rnd * 256, Rnd * 256)
        Next intI
    `Use bright colors for action types 5 or 6
    Else
        mlngColorNum(0) = QBColor(Int(8 * Rnd) + 8)
    End If

```

```
End Sub
```

I've set up six unique types of graphics animations, with variations on the number of lines and the thickness of each line in pixels. Much of the code varies only slightly among these various modes, although the differences can be dramatic. For example, the *B* parameter that is added to the Line method causes the same command to draw a series of boxes instead of diagonal lines.

MYSETUP.FRM

As mentioned, the MySetup form is activated when the user clicks the Settings button after selecting this screen saver. You can select a screen saver by right-clicking on the Windows 95 desktop, choosing Properties, and then clicking on the Screen Saver tab in the Display Properties dialog box that appears.

MYSETUP.FRM is a dialog box that lets you select one of six types of graphics displays and lets you modify each of these by selecting the number of lines and the thickness in pixels of each line. These settings are read from and written to the system Registry using the GetSetting and SaveSetting statements. The dialog box always displays the current settings when it opens. Figure 29-15 shows the MySetup form during the development process.

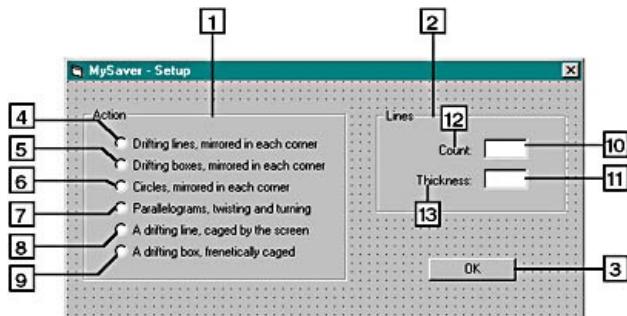


Figure 29-15. The MySetup form during development.

The following table and source code define the form:

MYSETUP.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmMySetup</i>
	BorderStyle	3 - Fixed Dialog
	Caption	MySaver - Setup
	ScaleMode	3 - Pixel
Frame		
1	Name	<i>Frame1</i>
	Caption	<i>Action</i>
Frame		
2	Name	<i>Frame2</i>
	Caption	<i>Lines</i>
CommandButton		
3	Name	<i>cmdOK</i>
	Caption	<i>OK</i>

OptionButton

4	Name	<i>optAction</i>
	Index	<i>0</i>
	Caption	<i>Drifting lines, mirrored in each corner</i>

OptionButton

5	Name	<i>optAction</i>
	Index	<i>1</i>
	Caption	<i>Drifting boxes, mirrored in each corner</i>

OptionButton

6	Name	<i>optAction</i>
	Index	<i>2</i>
	Caption	<i>Circles, mirrored in each corner</i>

OptionButton

7	Name	<i>optAction</i>
	Index	<i>3</i>
	Caption	<i>Parallelograms, twisting and turning</i>

OptionButton

8	Name	<i>optAction</i>
	Index	<i>4</i>
	Caption	<i>A drifting line, caged by the screen</i>

OptionButton

9	Name	<i>optAction</i>
	Index	<i>5</i>
	Caption	<i>A drifting box, frenetically caged</i>

TextBox

10	Name	<i>txtLineCount</i>

TextBox

11	Name	<i>txtLineWidth</i>

Label

12	Name	<i>Label1</i>
	Caption	<i>Count:</i>

Label

13	Name	<i>label2</i>
	Caption	<i>Thickness:</i>

* The number in the ID No. column corresponds to the number in Figure 29-15 that identifies the location of the object on the form.

Source Code for MYSETUP.FRM

```

`MySetup.frm
Option Explicit

Dim mstrAction As String

Private Sub Form_Load()
    'Center this form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
    'Get current settings from the Registry
    mstrAction = GetSetting("MySaver", "Options", "Action", "1")
    optAction(Val(mstrAction) - 1).Value = True
    txtLineCount.Text = GetSetting("MySaver", "Options",
        "LineCount", "5")
    txtLineWidth.Text = GetSetting("MySaver", "Options",
        "LineWidth", "1")
    Me.Show
End Sub

Private Sub cmdOK_Click()
    Dim lngN As Long
    'Check line count option
    lngN = Val(txtLineCount.Text)
    If lngN < 1 Or lngN > 1000 Then
        MsgBox "Line count should be a small positive integer", _
            vbExclamation, "MySaver"
        Exit Sub
    End If
    'Check line thickness option
    lngN = Val(txtLineWidth.Text)
    If lngN < 1 Or lngN > 100 Then
        MsgBox _
            "Line thickness should be a small positive integer", _
            vbExclamation, "MySaver"
        Exit Sub
    End If
    'Save the settings
    SaveSetting "MySaver", "Options", "Action", mstrAction
    SaveSetting "MySaver", "Options", "LineCount", _
        txtLineCount.Text
    SaveSetting "MySaver", "Options", "LineWidth", _
        txtLineWidth.Text
    'Close the Setup dialog box
    Unload Me
End Sub

Private Sub optAction_Click(Index As Integer)
    mstrAction = Format(Index + 1)
End Sub

```

Most of the code in this form is used to read and write settings to and from the Registry. I used defaults in the GetSetting statements to guarantee a valid setting even if the setting doesn't yet exist in the Registry.

To complete the screen saver, you must compile it as an executable file with an SCR extension—for example, MySaver.scr. Copy the resulting SCR file into your Windows directory. Your screen saver should then be listed in the drop-down list on the Screen Saver tab of the Display Properties dialog box. For more information about screen savers and how to compile a screen saver, see Chapter 25, "[Screen Savers](#)."

Chapter Thirty

Development Tools

This chapter contains three applications that will help you control your Visual Basic programming environment, and one to enhance your experimentation with the VBScript and JScript scripting languages. The ColorBar application helps you adjust your monitor so that you can see all the color characteristics your users will see. The APIAddin application is an add-in to the Visual Basic development environment that lets you locate, copy, and paste constants, types, and declarations for Windows 32-bit API functions. The Metric application demonstrates one way to extend your set of application development tools—in this case, by putting much of the functionality of the application in a help file and using Visual Basic to do the tasks it does best.

The ColorBar Application

The ColorBar application is quite simple, but I've found it very useful for adjusting my monitor to balance the colors and brightness. It also demonstrates a couple of programming techniques you'll probably find useful elsewhere. Figure 30-1 below shows the ColorBar application in action, and Figure 30-2 shows the project list.

When you run ColorBar, you see a form filled with 16 rectangles, each containing one of the 16 primary colors defined by the QBColor function. Make sure the yellow block doesn't look brown. (I had a monitor like that a few years ago, and it drove me nuts!) Also make sure each color is distinct from all the others—for example, you should see two distinct shades of gray. Click with the left mouse button on the form to rotate the color blocks one way, and click with the right button to rotate them the other way. After 16 clicks in either direction, the blocks return to their original locations.

COLORBAR.FRM

Notice a couple of details about this form. The MinButton property is set to *False* to prevent an error condition. Each time the user resizes the form, the surface of the form is rescaled to simplify the drawing of the blocks. The Scale statement generates an error if the form is minimized to an icon. Because this utility serves little purpose in the minimized state, I decided to simply eliminate the MinButton option. As shown in Figure 30-1, the button is still visible even when the form's MinButton property is set to *False*. Note, however, that the button is grayed and inactive.

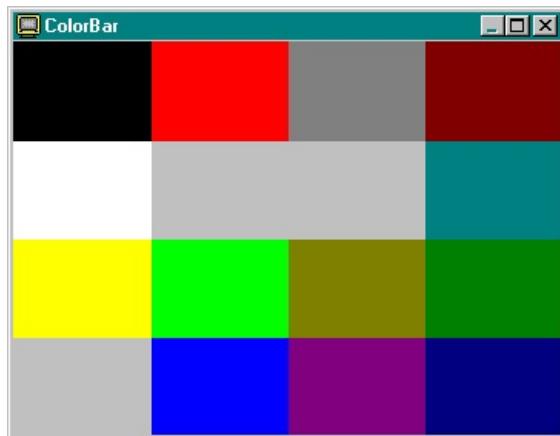


Figure 30-1. The ColorBar application in action.

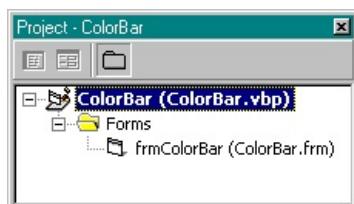


Figure 30-2. The ColorBar project list.

The only control on the form is a timer, as shown in Figure 30-3. Originally I put the block drawing code in the form's Paint event procedure, a technique that worked great as long as I stretched the form to a larger size at runtime, triggering a Paint event. But when I shrank the form a little, the Paint event was not called. To get around this result, I activated the timer at the various places in my program where I wanted to redraw the blocks of color. The timer is set up as a one-shot event, which means its code is activated once and then the timer shuts itself off. This one-shot action is triggered by setting the timer's Enabled property to *True* whenever the blocks are to be updated. By setting this property within the Resize event, I effectively enabled the program to redraw the blocks whenever the form is resized either larger or smaller. This also makes it easy to redraw the blocks when the mouse is clicked to rotate the order of the blocks.

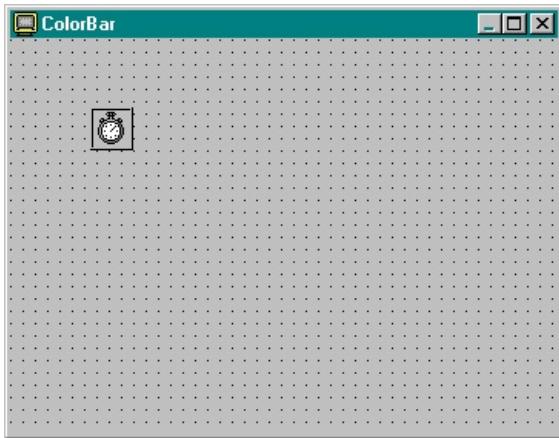


Figure 30-3. The ColorBar form during development.

To create this application, use the table and source code below to add the Timer control, set any nondefault properties as indicated, and enter the source code lines as shown.

COLORBAR.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmColorBar</i>
Caption	<i>ColorBar</i>
MinButton	<i>False</i>
Icon	<i>Monitr01.ico</i>
Timer	
Name	<i>tmrDrawBars</i>
Interval	<i>1</i>

Source Code for COLORBAR.FRM

```

Option Explicit

Dim mintColorShift As Integer

Private Sub Form_Load()
    'Center form on screen
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
End Sub

Private Sub Form_MouseDown(Button As Integer, _
Shift As Integer, X As Single, Y As Single)
    'Shift color bars based on mouse button
    mintColorShift = (mintColorShift - Button * 2 + 19) Mod 16
    'Activate timer to draw color bars
    tmrDrawBars.Enabled = True
End Sub

Private Sub Form_Resize()
    'Activate timer to draw color bars
    tmrDrawBars.Enabled = True
End Sub

Private Sub tmrDrawBars_Timer()

```

```
Dim intX As Integer
Dim intY As Integer
`Deactivate timer so that color bars are drawn only once
tmrDrawBars.Enabled = False
`Scale form for convenience
Scale (4, 4)-(0, 0)
`Fill in colors
For intX = 0 To 3
    For intY = 0 To 3
        mintColorShift = (mintColorShift + 1) Mod 16
        Line (intX, intY)-(intX + 1, intY + 1), _
            QBColor(mintColorShift), BF
    Next intY
Next intX
End Sub
```

The APIAddin Application

The APIAddin application is an add-in to the Visual Basic development environment. I include it to provide a working example of an add-in and because I like the way I've structured the API declarations better than the way this information is presented in WIN32API.TXT, a file you'll find in your Visual Studio folders. On my system this file is located at C:\PROGRAM FILES\MICROSOFT VISUAL STUDIO\COMMON\TOOLS\WINAPI.

The line continuation character makes it easy to format declarations in an easier-to-read, multiline layout. Throughout this book, I've taken advantage of this capability, and this application lets you easily add API functions to your applications in the same format. Figure 30-4 shows the APIAddin application in action, displaying a few of the multiline API function declarations. Notice that once the application is installed as an add-in, the dialog box that this application creates is accessed directly from the Add-Ins menu, letting you quickly and easily locate API functions and copy and paste them into your applications.

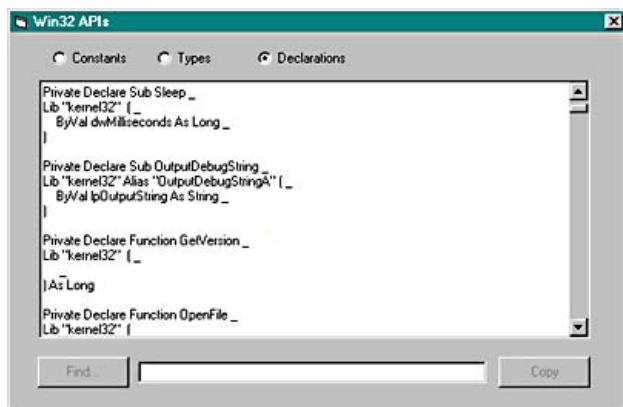


Figure 30-4. The APIAddin application in action.

Converting the WIN32API.TXT File

Before I describe the APIAddin application itself, I want to make you aware that the original WIN32API.TXT file must be modified to provide three new working files that the APIAddin application loads at runtime. This conversion process is performed only once, and because I've provided the resulting files on the companion CD-ROM, you won't even need to run it once. However, I'll include the source code for the conversion process here in case you are interested in modifying the format even further, or you have a newer version of the WIN32API.TXT file and want to perform an update conversion.

The code below can be plopped into a fresh, blank form. I've provided no controls or other modifications. Simply click once on the running form, and wait until a "Done" message is displayed on the form. (This demonstrates a handy way to create a small utility program for your own use when there's no need for a fancy user interface.) You'll need to either move the WIN32API.TXT file to the directory containing your application or modify the path in the code so that the program can find the file. This code can be found in the CVTAPITX.VBP project on the companion CD-ROM.

Source Code for WIN32API.TXT Conversion

```
Option Explicit

Private Sub Form_Click()
    Dim strA As String
    Dim strT As String
    Dim intState As Integer
    Dim intI As Integer
    Dim intJ As Integer
    Dim intK As Integer
    Dim intN As Integer
    Print "WorkingDear John, How Do I... "
    Open App.Path & "\Win32api.txt" For Input As #1
    Open App.Path & "\W32cons.txt" For Output As #2
    Open App.Path & "\W32type.txt" For Output As #3

```

```

Open App.Path & "\W32decl.txt" For Output As #4
Do Until EOF(1)
    Line Input #1, strA
    If InStr(strA, "Const ") Then intState = 2
    If InStr(LTrim(strA), "Type ") = 1 Then
        strA = "Private " & strA
        intState = 3
    End If
    If InStr(LTrim(strA), "Declare ") = 1 Then
        intState = 4
    End If
    If intState = 2 Then
        intI = InStr(strA, "`")
        If intI > 0 Then
            strA = Trim(Left(strA, intI - 1))
        End If
        If strA <> "" Then
            intI = InStr(strA, "Public")
            If intI = 1 Then
                Print #2, strA
            Else
                Print #2, "Private " & strA
            End If
        End If
    End If
    If intState = 3 Then
        If Left(strA, 1) = " " Then
            strA = Space(4) & Trim(strA)
        End If
        intJ = InStr(strA, "`")
        If intJ > 0 Then
            strA = RTrim(Left(strA, intJ - 1))
        End If
        Print #3, strA
    End If
    If intState = 4 Then
        intN = 2
        If Trim(strA) = "" Then
            Print #4, ""
        Else
            `Lop off comments
            intI = InStr(strA, ")")
            intJ = InStr(intI, strA, "`")
            If intJ > intI Then strA = Trim(Left(strA, intJ - 1))
            `Drop Alias if not different from original function
            intI = InStr(strA, "Alias")
            If intI Then
                intJ = InStr(intI, strA, Chr(34))
                intK = InStr(intJ + 1, strA, Chr(34))
                strT = Mid(strA, intJ + 1, intK - intJ - 1)
                strT = Space(1) & strT & Space(1)
                If InStr(strA, strT) Then
                    strA = Left(strA, intI - 1) & _
                        Mid(strA, intK + 1)
                End If
            End If
            `Locate "Lib"
            intI = InStr(strA, " Lib")
            `Insert "Private"
            Print #4, "Private Declare " & _
                Mid(strA, 9, intI - 8) & " _"
            strA = Mid(strA, intI + 1)
            `Locate left parenthesis
            intI = InStr(strA, "(")

```

```

        Print #4, Left(strA, intI) & " _"
        strA = Mid(strA, intI + 1)
        `Locate each parameter
        Do
            intI = InStr(strA, ", ")
            If intI = 0 Then Exit Do
            Print #4, Space(4) & Left(strA, intI) & " _"
            intN = intN + 1
            strA = Mid(strA, intI + 2)
        Loop;
        `Locate right parenthesis
        intI = InStr(strA, ")")
        Print #4, Space(4) & Left(strA, intI - 1) & " _"
        Print #4, Mid(strA, intI)
    End If

    End If
    If intState = 2 Then intState = 0
    If intState = 3 And InStr(strA, "End Type") > 0 Then
        intState = 0
        Print #3, ""
    End If
    If intState = 4 Then
        intState = 0
        Print #4, ""
    End If
Loop
Close #1
Close #2
Close #3
Close #4
Print "Done"
End Sub

```

When you run this code, WIN32API.TXT is split into three files: W32CONS.TXT contains a list of all constants, W32TYPE.TXT contains all UDT structure definitions, and W32DECL.TXT contains all the declarations for all API functions. The program removes all extraneous comments and extra blank lines in order to keep the files small and quick to load. The Private declaration is added to minimize the scope of all declarations. For explanations and descriptions of these functions, you'll need to consult the Win32 SDK Reference Help or another source. These trimmed-down files are designed to provide the declarations as efficiently as possible and not to explain their use.

I discovered an interesting fact about Visual Basic while creating these files. On one of the first iterations using these files, I left all the Alias modifiers within the Declare statements. However, when you enter a Declare statement in which the aliased function name is identical to the original name, Visual Basic automatically deletes the Alias part. This is cool, except that the extra Alias text, which is going to get zapped automatically by your Visual Basic application anyway, takes up some space in the W32DECL.TXT file, and my goal is to condense this file as much as possible. So my preparation program strips out the Alias modifiers when the original name is identical to the aliased name. Compare the contents of W32DECL.TXT and WIN32API.TXT to see the difference in the functions when this modification is performed.

One final note: keep these three text files in the same directory as the APIAddin application's executable file. I've used the App.Path property to locate these files as they are needed, simplifying the housekeeping.

Building the APIAddin Application

The easiest way to begin building an add-in application is to start a new project and double-click the Addin icon in the New Project dialog box. An almost complete application is created, containing one form and one add-in designer—the framework for your new add-in application. That's how I first created the

APIAddin application. From this point on, I'll describe the changes that were required to modify this application to display the API data files.

First rename and save the form that was automatically created and named for you. Name the form *frmAPIAddin*, and save it as APIADDIN.FRM. Save the Connect designer as CONNECT.DSR. Finally, name the project APIAddin, and save it as APIADDIN.VBP. Figure 30-5 shows the project list after these changes have been made.

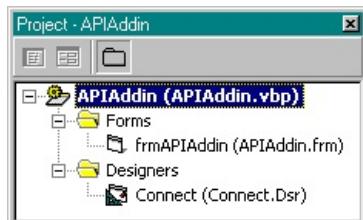


Figure 30-5. The project list for the APIAddin application.

CONNECT.DSR

There's not a lot of code in this project, and the designer requires just a few changes from the code that was created automatically. The biggest change to the Connect designer code is a global replacement of all occurrences of *frmAddIn* with *frmAPIAddin*. I did make a few other changes, though, so check each line of your code carefully to be sure it matches the following listing. Here's the code as it should appear in the edited CONNECT.DSR designer module.

Source Code for CONNECT.DSR

```
Option Explicit

Public FormDisplayed As Boolean
Public VBInstance As VBIDE.VBE
Dim mcbMenuCommandBar As Office.CommandBarControl
Dim mfrmAPIAddin As New frmAPIAddin
Public WithEvents MenuHandler As CommandBarEvents

Sub Hide()
    On Error Resume Next
    FormDisplayed = False
    mfrmAPIAddin.Hide
End Sub

Sub Show()
    On Error Resume Next
    If mfrmAPIAddin Is Nothing Then
        Set mfrmAPIAddin = New frmAPIAddin
    End If
    Set mfrmAPIAddin.VBInstance = VBInstance
    Set mfrmAPIAddin.Connect = Me
    FormDisplayed = True
    mfrmAPIAddin.Show
End Sub

'-----
`This method adds the add-in to VB
'-----

Private Sub AddinInstance_OnConnection(ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant _
)
    On Error GoTo error_handler
    `Save the VB instance
    Set VBInstance = Application
    `This is a good place to set a breakpoint and
```

```

`test various add-in objects, properties, and methods
Debug.Print VBInstance.FullName
If ConnectMode = ext_cm_External Then
    `Used by the wizard toolbar to start this wizard
    Me.Show
Else
    Set mcbMenuCommandBar = AddToAddInCommandBar("API Addin")
    `Sink the event
    Set Me.MenuHandler =
        VBInstance.Events.CommandBarEvents(mcbMenuCommandBar)
End If

If ConnectMode = ext_cm_AfterStartup Then
    If GetSetting(App.Title, "Settings",
        "DisplayOnConnect", "0") = "1" Then
        `Set this to display the form on connect
        Me.Show
    End If
End If
Exit Sub
error_handler:
    MsgBox Err.Description
End Sub

`-----
`This method removes the add-in from VB
`-----

Private Sub AddinInstance_Disconnection(ByVal RemoveMode _
    As AddInDesignerObjects.ext_DisconnectMode, _
    custom() As Variant _
)
    On Error Resume Next
    `Delete the command bar entry
    mcbMenuCommandBar.Delete
    `Shut down the add-in
    If FormDisplayed Then
        SaveSetting App.Title, "Settings", "DisplayOnConnect", "1"
        FormDisplayed = False
    Else
        SaveSetting App.Title, "Settings", "DisplayOnConnect", "0"
    End If
    Unload mfrmAPIAddin
    Set mfrmAPIAddin = Nothing
End Sub

Private Sub IDTExtensibility_StartupComplete(custom() As Variant)
    If GetSetting(App.Title, "Settings",
        "DisplayOnConnect", "0") = "1" Then
        `Set this to display the form on connect
        Me.Show
    End If
End Sub

`This event fires when the menu is clicked in the IDE
Private Sub MenuHandler_Click(ByVal CommandBarControl As Object, _
    handled As Boolean, CancelDefault As Boolean _
)
    Me.Show
End Sub

Function AddToAddInCommandBar(sCaption As String) As _
    Office.CommandBarControl
    Dim cbMenuCommandBar As Office.CommandBarControl `command bar object
    Dim cbMenu As Object

```

```

On Error GoTo AddToAddInCommandBarErr
`See if we can find the Add-Ins menu
Set cbMenu = VBInstance.CommandBars("Add-Ins")
If cbMenu Is Nothing Then
    `Not available, so we fail
    Exit Function
End If
`Add it to the command bar
Set cbMenuCommandBar = cbMenu.Controls.Add(1)
`Set the caption
cbMenuCommandBar.Caption = sCaption
Set AddToAddInCommandBar = cbMenuCommandBar
Exit Function
AddToAddInCommandBarErr:
End Function

```

The Connect designer's purpose is to provide the connections to the Visual Basic integrated development environment (IDE). This designer provides the code that connects actions to menus, toolbar buttons, or to other events in your Visual Basic environment. Be sure to study the explanations and examples in Visual Basic Books Online to gain a thorough understanding of all the ways the Connect designer can be modified to suit your purposes.

APIADDIN.FRM

The APIAddin form displays one of three files in a RichTextBox control, depending on which option button is selected along the top of the form. For efficiency, each of the three files is loaded into a string variable only if the file is requested by the user, and it's loaded only once per run of the application. When selected, the contents of the appropriate string are copied into the rich text box's Text property. Figure 30-6 shows the APIAddin form during the development process, and Figure 30-7 shows the form in action, with the list of constants displayed.

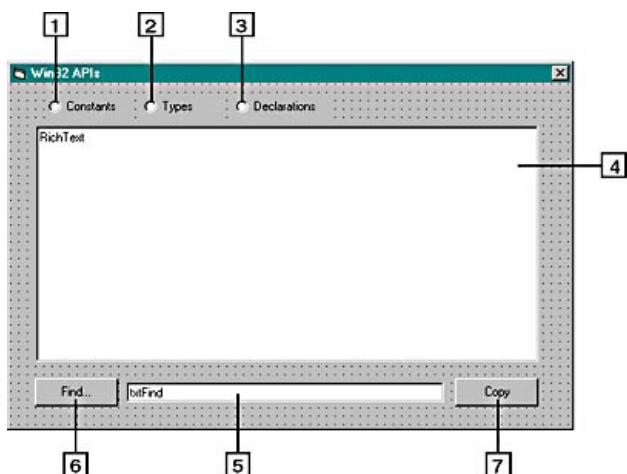


Figure 30-6. The APIAddin form during development.

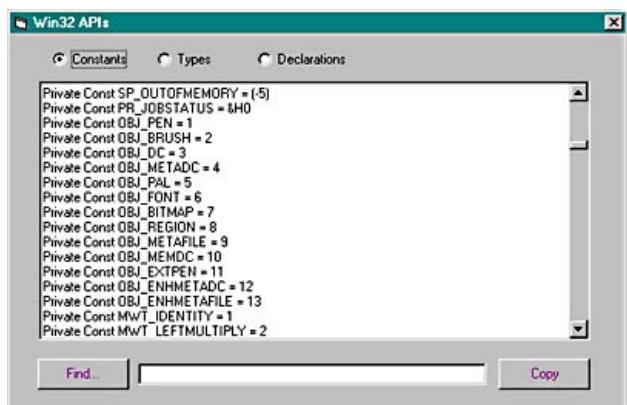


Figure 30-7. The APIAddin form, displaying the list of constants.

The APIAddin form creates an instance of the Connect object to make the connection to Visual Basic's Add-Ins menu and displays the API constants, types, and declarations when the new menu item is selected. To create this form, use the following table and source code to add the appropriate controls, set any nondefault properties for APIADDIN.FRM as indicated, and enter the source code lines as shown after the following table.

APIADDIN.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmAPIAddin</i>
	Caption	<i>Win32 APIs</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
OptionButton		
1	Name	<i>optAPI</i>
	Caption	<i>Constants</i>
	Index	<i>0</i>
OptionButton		
2	Name	<i>optAPI</i>
	Caption	<i>Types</i>
	Index	<i>1</i>
OptionButton		
3	Name	<i>optAPI</i>
	Caption	<i>Declarations</i>
	Index	<i>2</i>
RichTextBox		
4	Name	<i>rtfAPI</i>
	HideSelection	<i>False</i>
	Scrollbars	<i>3 - rtfBoth</i>
TextBox		
5	Name	<i>txtFind</i>
CommandButton		
6	Name	<i>cmdFind</i>
	Caption	<i>FindDear John, How Do I...</i>
CommandButton		
7	Name	<i>cmdCopy</i>
	Caption	<i>Copy</i>

* The number in the ID No. column corresponds to the number in Figure 30-6 that identifies the location of the object on the form.

Source Code for APIADDIN.FRM

Option Explicit

```

Public VBInstance As VBIDE.VBE
Public Connect As Connect

Private mstrCon As String
Private mstrTyp As String
Private mstrDec As String

Private Sub cmdCopy_Click()
    'Copy selected text to clipboard
    Clipboard.SetText rtfAPI.SelText
    'Return to user's project
    Unload Me
End Sub

Private Sub cmdFind_Click()
    Dim lngPtr As Long
    Dim strFind As String
    'Put focus on rich text box
    rtfAPI.SetFocus
    'Grab search string
    strFind = txtFind.Text
    'Determine where to begin search
    If rtfAPI.SelLength Then
        lngPtr = rtfAPI.SelStart + rtfAPI.SelLength
    Else
        lngPtr = 0
    End If
    'Use rich text box's Find method
    lngPtr = rtfAPI.Find(strFind, lngPtr)
    If lngPtr = -1 Then
        MsgBox "Search text not found"
    End If
End Sub

Private Sub Form_Load()
    'Startup showing declarations
    optAPI(2).Value = True
    txtFind.Text = ""
    Me.Show
    Me.ZOrder
End Sub

Private Sub optAPI_Click(Index As Integer)
    Select Case Index
        'Constants
        Case 0
            If mstrCon = "" Then
                LoadUp mstrCon, "W32cons.txt"
            End If
            rtfAPI.Text = mstrCon
        'Type structures
        Case 1
            If mstrTyp = "" Then
                LoadUp mstrTyp, "W32type.txt"
            End If
            rtfAPI.Text = mstrTyp
        'Declarations
        Case 2
            If mstrDec = "" Then
                LoadUp mstrDec, "W32decl.txt"
            End If
    End Select
End Sub

```

```

        End If
        rtfAPI.Text = mstrDec
    End Select
End Sub

Private Sub LoadUp(sA As String, sFile As String)
    Open App.Path & "\" & sFile For Binary As #1
    sA = Space(LOF(1))
    Get #1, , sA
    Close #1
End Sub

Private Sub rtfAPI_SelChange()
    If rtfAPI.SelLength Then
        cmdCopy.Enabled = True
    Else
        cmdCopy.Enabled = False
    End If
End Sub

Private Sub txtFind_Change()
    If Len(txtFind.Text) Then
        cmdFind.Enabled = True
    Else
        cmdFind.Enabled = False
    End If
End Sub

```

The rich text box turns out to be very easy to use for this application. Scrollbars allow manual scanning of the large amount of text, and the RichTextBox control provides its own Find method. This Find method implements functionality that would be complicated to program yourself. For example, once a fragment of text is searched for and found, the rich text box automatically scrolls to the appropriate spot and highlights the text, just as the user would expect. Likewise, the special properties related to text selection simplify the task of identifying a block of text to be copied to the clipboard. See the descriptions of the SelStart, SelLength, and SelText properties in the Visual Basic online help for more information about text selection within a RichTextBox control.

Compiling the Add-In

Before compiling this or any other add-in project, be sure to enable the references to Microsoft Visual Basic 6.0 Extensibility and to the Microsoft Office 8.0 Object Library. To do so, choose References from the Project menu and verify that these items are checked in the References dialog box.

An add-in is an ActiveX component and can be created as either a dynamic link library (DLL) or an EXE file. In most cases, you'll want to create an ActiveX DLL. From the Project menu, choose APIAddin Properties, and verify that Project Type is set to *ActiveX DLL* in the Project Properties dialog box.

To compile the add-in, choose Make APIADDIN.DLL from the File menu. Save the resulting APIADDIN.DLL file in the same directory in which you saved the three API text files. I've used the App.Path property to locate these files, and the application assumes that they're located in the same directory as the DLL. During the compiling process, an entry in the Registry is automatically created for you, and the Registry remembers where the DLL is stored.

To try out the APIAddin add-in, start a new Standard EXE project. From the Add-Ins menu, choose Add-In Manager. You should see the new API Add-In item in the Add-In Manager dialog box. (The items are in alphabetic order.) Check the Loaded/Unloaded check box to load the new add-in, and click OK. Open the Add-Ins menu again, and notice that there's now an API Addin menu option. Choose the new menu option, and you should see the new Win32 APIs dialog box.

The purpose of the Win32 APIs dialog box is to provide quick and easy access to the 32-bit API declarations, constants, and type structures. Click the three option buttons at the top of the dialog box to see how each of these types of API data lists are displayed. To find a specific entry, type part or all of the

text in the text box at the bottom of the dialog box and click the Find button. Once you've found what you're looking for, highlight the lines and click the Copy button. The dialog box will disappear, and you'll be returned to your project, ready to paste the selected text from the clipboard.

The Metric Application

The Metric application is rather simple, but it demonstrates the powerful technique of combining a Visual Basic application with a help file in a symbiosis that incorporates the best features of each. I've created a very short example tutorial on the metric system of weights and measures using this technique. The METRIC.HLP file displays easy-to-read text and has pop-up windows and hypertext hot spots; METRIC.EXE, the Visual Basic half of the team, plays sound files and video clips, performs metric conversions for the user, and displays a quiz to test the user's understanding. Help files in Windows 95 can do much more than help files in earlier versions of Windows, but when you connect to Visual Basic the possibilities are virtually limitless.

When you run METRIC.EXE, it first checks to see whether you used any command line parameters. Assuming that you start the program by double-clicking on its name or its icon, there won't be any parameters, and the program immediately runs the METRIC.HLP file. I've provided several buttons in the main topic window of the help file that jump back into the METRIC.EXE program, each passing a different set of command line parameters. The METRIC.EXE program takes action based on these parameters and then terminates, allowing the user to interact once again with the help file.

METRIC.EXE effectively lets a help file play a sound file or a video clip, display a full-blown dialog box, perform mathematical computations, or do just about anything else you can imagine. The Metric application demonstrates each of these techniques. For example, when you click the Video Clip button in the main topic window of the help file, a sample video clip plays. Likewise, when you click the Quiz button, a dialog box full of quiz questions is displayed, as shown in Figure 30-8. Study the source code listings to see how easily this was accomplished. Note that both METRIC.EXE and METRIC.HLP should be in the same directory at runtime. Also, to view the video clip CLOUDS.AVI, copy the file from the companion CD-ROM to the directory containing the Metric application.

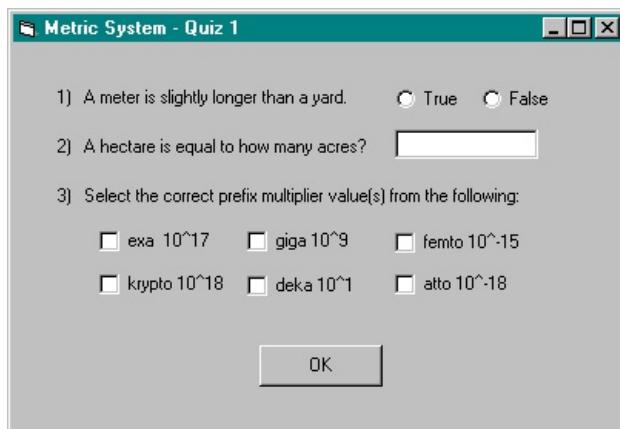


Figure 30-8. The quiz activated from within the METRIC.HLP file.

Building the Metric Application

The Visual Basic part of the Metric application consists of three files, as shown in the project list in Figure 30-9.

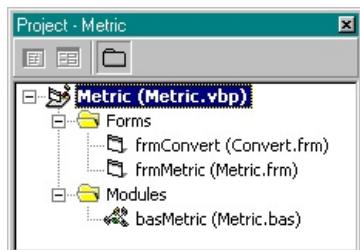


Figure 30-9. The Metric project list.

The METRIC.BAS code module contains most of the code for this application, including the startup routine Sub Main, in which action decisions based on the command line parameters are made. The METRIC.FRM form displays the sample quiz, and the CONVERT.FRM form lets the user perform conversions from meters to feet and from centimeters to inches.

METRIC.BAS

The METRIC.BAS code module contains the Sub Main startup point for the Visual Basic half of this application. Sub Main calls a useful procedure named GetParms, which analyzes and parses an application's command line, returning an array of parameters that's often much easier to process than Visual Basic's Command function. The public dynamic array *gstrParm* returns each parameter, and even in the case in which no command line parameters were given, *gstrParm(0)* returns the full pathname and filename of the executable file. The GetParms procedure uses the new string function Split to efficiently parse the command line.

Based on the evaluated command line parameters, Sub Main plays a multimedia file, displays a quiz, or performs metric conversions. You can easily add other capabilities by modifying the Select Case statements to intercept and process command line parameters of your own design.

Source Code for METRIC.BAS

```
Option Explicit

Private Declare Function WinExec _
Lib "kernel32" (
    ByVal lpCmdLine As String,
    ByVal nCmdShow As Long
) As Long

Private Declare Function mciExecute _
Lib "winmm.dll" (
    ByVal lpstrCommand As String
) As Long

Public gstrParm() As String

Sub Main()
    GetParms
    Select Case UBound(gstrParm)
        Case 0
            TakeAction
        Case 1
            TakeAction gstrParm(1)
        Case 2
            TakeAction gstrParm(1), gstrParm(2)
    End Select
End Sub

Private Sub TakeAction(Optional vntCmd, Optional vntFil)
    'If no parameters, open help file
    If IsMissing(vntCmd) Then
        WinExec "Winhelp.exe " & App.Path & "\Metric.hlp", 1
        Exit Sub
    End If
    'First parameter determines action to take
    Select Case UCase(vntCmd)
        'Display units conversion form
        Case "M2F", "C2I"
            frmConvert.Show
        'Play a sound file
        Case "WAV"
            Select Case vntFil
                Case 1
                    mciExecute "Play Sound1.wav"
            End Select
        'Play a video clip
        Case "AVI"
            Select Case vntFil
```

```

Case 1
    mciExecute "Play Clouds.avi"
End Select
`Display a quiz form
Case "QUIZ"
    frmMetric.Show
End Select
End Sub

Private Sub GetParms()
    Dim strCmd As String
    strCmd = Trim("X" & Space(1) & Command)
    gstrParm() = Split(strCmd)
    If Right(App.Path, 1) <> "\" Then
        gstrParm(0) = App.Path & "\" & App.EXENAME
    Else
        gstrParm(0) = App.Path & App.EXENAME
    End If
End Sub

```

METRIC.BAS uses two API functions: one to play the multimedia files and one to start the METRIC.HLP file. Notice that I've used yet another technique to start up a help file. (See Chapter 17, "[User Assistance](#)," for other ways to do this.) The WinExec API function lets your Visual Basic application start any Windows application—WinHelp, in this case. You could use this function to start the Calculator application, for instance, which might be a handy addition to a tutorial.

The TakeAction procedure demonstrates the use of optional parameters by accepting none, one, or two parameters.

METRIC.FRM

This form displays a sample quiz when the Quiz button in the help file is clicked. (Read on to see how the help file handles this button click.) To keep this sample application simple, I've created a nongraded quiz that displays one true/false question, one fill-in-the-blank question, and one multiple-choice question. It's only a start, and I'm sure you'll want to enhance a real quiz dialog box beyond this simple example. Figure 30-10 below shows the Metric form during development; Figure 30-8 above shows the form in action after it has been activated within the help file.

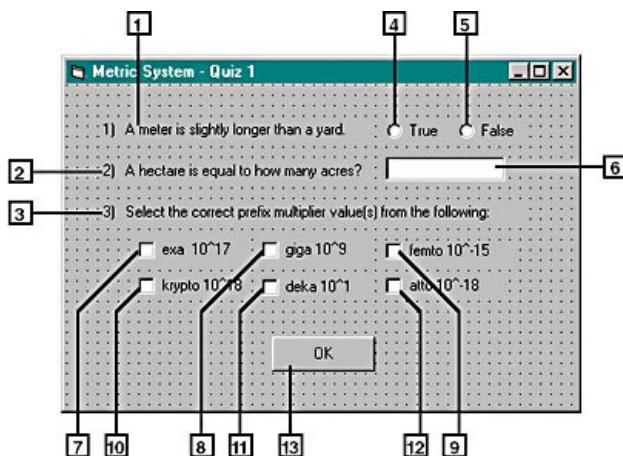


Figure 30-10. The Metric form during development.

METRIC.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	frmMetric

	Caption	<i>Metric System - Quiz 1</i>
Label		
1	Name	<i>Label1</i>
	Caption	<i>1) A meter is slightly longer than a yard.</i>
Label		
2	Name	<i>Label2</i>
	Caption	<i>2) A hectare is equal to how many acres?</i>
Label		
3	Name	<i>Label3</i>
	Caption	<i>3) Select the correct prefix multiplier value(s) from the following:</i>
Option1		
4	Caption	<i>True</i>
Option2		
5	Caption	<i>False</i>
Text1		
6	Text	<i>(blank)</i>
Check1		
7	Caption	<i>exa 10^{17}</i>
Check2		
8	Caption	<i>giga 10^9</i>
Check3		
9	Caption	<i>femto 10^{-15}</i>
Check4		
10	Caption	<i>krypto 10^{18}</i>
Check5		
11	Caption	<i>deka 10^1</i>
Check6		
12	Caption	<i>atto 10^{-18}</i>
CommandButton		
13	Name	<i>cmdOK</i>

*The number in the ID No. column corresponds to the number in Figure 30-10 that identifies the location of the object on the form.

Source Code for METRIC.FRM

```
Option Explicit

Private Sub cmdOK_Click()
    MsgBox "This example quiz is not graded."
    Unload Me
End Sub
```

CONVERT.FRM

The Convert form displays a dialog box that lets the user perform mathematical calculations—in this case, conversions of meters to feet and centimeters to inches. Once again, this example is greatly simplified, and a full suite of conversions would probably be in order for a real application along these lines. I've included enough in this dialog box to get you started and to demonstrate the core technique of using Visual Basic to add computational abilities to a help file. Figure 30-11 shows the CONVERT.FRM form during development, and Figure 30-12 shows the form at runtime, after the user has clicked one of the buttons in the Conversions section of the help file.

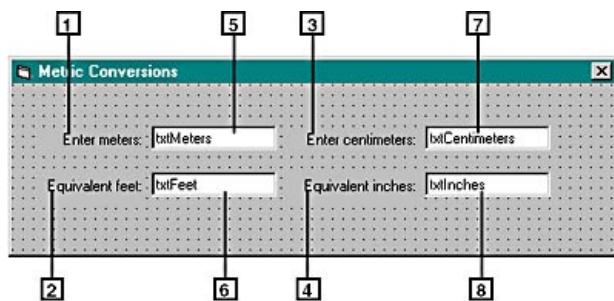


Figure 30-11. The Convert form during development.

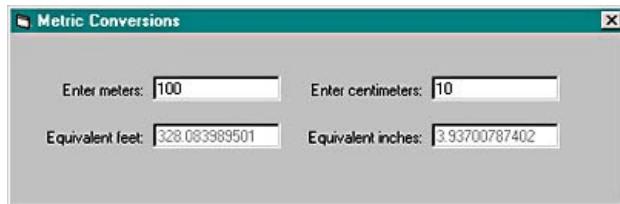


Figure 30-12. The Metric Conversions dialog box when activated by a button in the Metric help file.

To create this application, use the table and source code below to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

CONVERT.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmConvert</i>
	Caption	<i>Metric Conversions</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
Label		
1	Name	<i>lblMeters</i>
	Caption	<i>Enter meters:</i>
Label		
2	Name	<i>lblFeet</i>
	Caption	<i>Equivalent feet:</i>
Label		
3	Name	<i>lblCentimeters</i>
	Caption	<i>Enter centimeters:</i>
Label		

4	Name	<i>lblInches</i>
	Caption	<i>Equivalent inches:</i>
TextBox		
5	Name	<i>txtMeters</i>
TextBox		
6	Name	<i>txtFeet</i>
	Enabled	<i>False</i>
TextBox		
7	Name	<i>txtCentimeters</i>
TextBox		
8	Name	<i>txtInches</i>
	Enabled	<i>False</i>

* The number in the ID No. column corresponds to the number in Figure 30-11 that identifies the location of the object on the form.

Source Code for CONVERT.FRM

```
Option Explicit

Private Sub Form_Load()
    txtMeters.Text = "0"
    txtCentimeters.Text = "0"
End Sub

Private Sub txtCentimeters_Change()
    txtInches.Text = CStr(CDbl(txtCentimeters.Text) * 0.393700787402)
End Sub

Private Sub txtMeters_Change()
    txtFeet.Text = CStr(CDbl(txtMeters.Text) * 3.28083989501)
End Sub
```

METRIC.HLP

Perhaps you've been wondering how the METRIC.EXE application is activated, complete with a variety of command line parameters, within the METRIC.HLP file. It's beyond the scope of this book to go into all the details of building a help file, but I will describe this critical part of the process. I used RoboHelp, from Blue Sky Software, to create the METRIC.HLP file. (RoboHelp is a great product for enhancing your 32-bit Visual Basic applications with full-blown 32-bit help files using all the latest techniques.) RoboHelp makes it easy to add macros to buttons or hot spots, and these macros are the key to activating an external application such as METRIC.EXE. You can also insert the buttons and macros using other help-building tools, such as the Microsoft Help Workshop that comes with Visual Basic (HCW.EXE), or you can code everything directly using footnotes in a Rich Text File (RTF) document. See the online help for the Microsoft Help Workshop for the button and macro syntax. For reference, here are the macros embedded in the METRIC.RTF source code for the buttons that appear in the main topic window of the help file:

Conversions:

```
{button Meters to feet, ExecProgram("Metric.exe M2F")}
{button Centimeters to inches, ExecProgram("Metric.exe C2I")} To hear a {button Sound, ExecProgram("Metric.exe wav 1")} click this button, and to see a sample {button Video clip, ExecProgram("Metric.exe avi 1")} click here. When you're ready,
```

```
click this {button Quiz, ExecProgram("Metric.exe QUIZ 1") } button.
```

For each button, everything between the braces is processed by the help file compiler to create a button that activates an external application named METRIC.EXE, passing the necessary command line parameters. Macros such as these are a standard feature of help files. RoboHelp explains the use of macros in detail, but no matter what tool you use to edit and compile help files, the documentation should provide an explanation of the inclusion of macros. Figure 30-13 shows this primary help topic in action. All the buttons providing macros leading back to the METRIC.EXE application are shown in this first help topic.

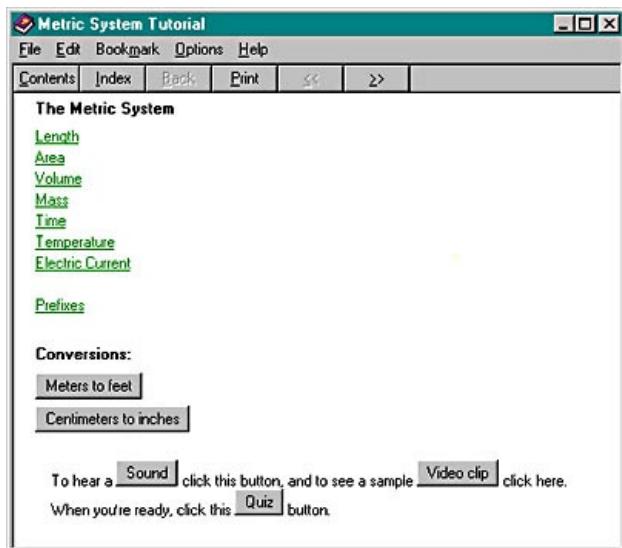


Figure 30-13. The METRIC.HLP file displaying the first help topic.

The ScripDem Application

The Script control isn't included with Visual Basic, but you can download it for free from the Microsoft Web site (<http://www.microsoft.com>). Once it's installed, you can add the Script control to your application by choosing Components from the Project menu and checking Microsoft Script Control 1.0 in the Components dialog box.

The Script control lets you run either VBScript or JScript source code, which provides a way to create a stand-alone VBScript development environment as a Visual Basic application. Although not an ideal full-featured development environment, the ScripDem application shows how this can be done, and it provides all the basics to let you try out simple VBScript routines.

The main form in this application contains a TextBox control to enter source code, and a Script control to run the code interactively. Syntax errors are handled to show you what the problem is and what line is at fault, but runtime errors can be a little harder to track down, depending on their nature. Be sure to study the VBScript language itself first, to avoid as many errors as possible in your source.

This application provided a good opportunity to demonstrate use of a Toolbar control with button images stored in an ImageList control. These buttons duplicate the functionality of the menus, and in the case of the Run button, the menu is actually used as a popup when the button is clicked. I've set the ToolTipText property and the Key property of each button to the same value. The ToolTipText property displays what each button does, and the Key property identifies which button is clicked.

The Run menu (and the Run button) let you select either VBScript or JScript as the scripting language, with VBScript as the default. Figure 30-14 shows the ScripDem form in action, displaying the source code for a test procedure, and the resulting message box displayed just after the Run button is clicked. Figure 30-15 shows the project list for the ScripDem application.

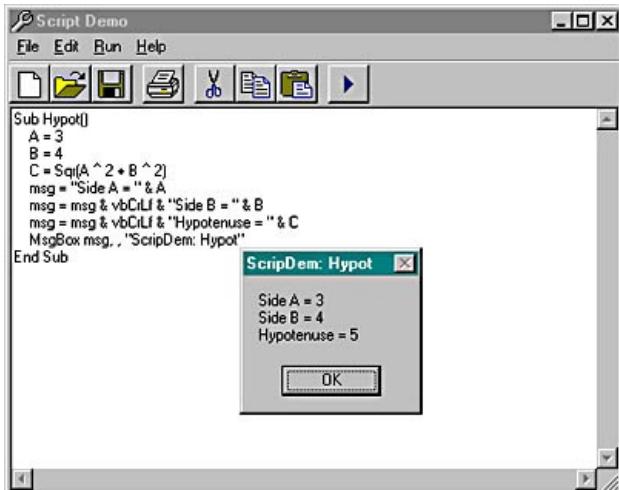


Figure 30-14. The ScripDem application in action, running a VBScript procedure.

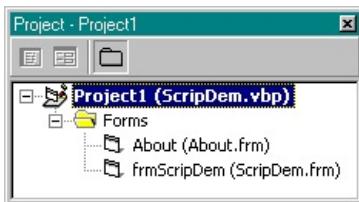


Figure 30-15. The ScripDem project list.

SCRIPDEM.FRM

As shown in Figure 30-16, the ScripDem form consists of coordinated Toolbar and ImageList controls, a TextBox control and a Script control for editing and running the VBScript source code, and a CommonDialog control for loading and saving VBScript source code files.

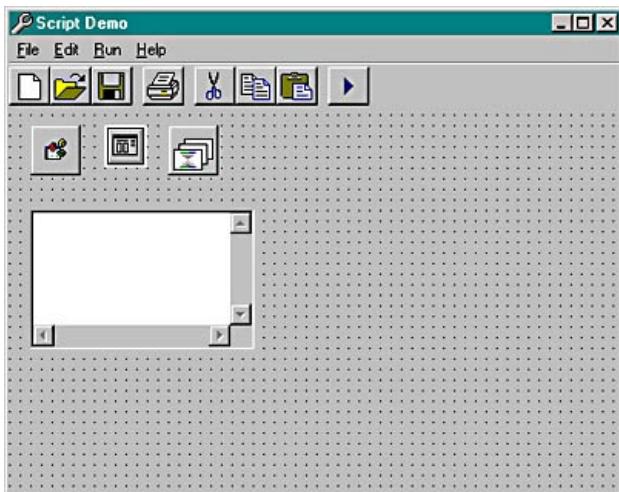


Figure 30-16. The ScripDem form during development.

To create this application, use the following tables and source code to add the controls, set any nondefault properties as indicated, and enter the source code lines as shown. Be sure to set the Toolbar's ImageList property to `ilToolbar`, and then add the buttons to `tlbToolbar` in the correct order. I added some separator "buttons" to visually group the buttons, but this is an optional step.

SCRIPDEM.FRM Menu Design Window Entries

Caption	Name	Indentation	Checked	Enabled
&File	<i>mnuFile</i>	0		<i>True</i>
&New	<i>mnuNew</i>	1		<i>True</i>

&Open Dear John, How Do I...	mnuOpen	1		True
&Save	mnuSave	1		True
-	mnuFileSep1	1		True
&Print	mnuPrint	1		True
-	mnuFileSep2	1		True
E&xit	mnuExit	1		True
&Edit	mnuEdit	0		True
Cu&t	mnuCut	1		True
&Copy	mnuCopy	1		True
&Paste	mnuPaste	1		True
&Run	mnuRun	0		True
&VisualBasic	mnuVB	1	Yes	True
&Java	mnuJava	1	No	True
-	mnuRunSep1	1		True
&Update Procedures	mnuUpdate	1		True
-	mnuRunSep2	1		True
&Help	mnuHelp	0		True
&Contents	mnuContents	1		True
&Search For Help On Dear John, How Do I...	mnuSearch	1		True
-	mnuHelpSep1	1		True
&About Dear John, How Do I...	mnuAbout	1		True

SCRIPDEM.FRM Objects and Property Settings

Property	Value
Form	
Name	frmScripDem
Caption	Script Demo
Icon	Wrench.ico
Toolbar	
Name	tlbToolbar
ImageList	ilsToolbar
ImageList	
Name	ilsToolbar
Images	New, Open, Save, Print, Cut, Copy, Paste, and Run
CommonDialog	
Name	cdlOne
TextBox	

Name	<i>txtModule</i>
Scrollbars	<i>3 - Both</i>
MultiLine	<i>True</i>
Script	
Name	<i>scrVB</i>
Language	<i>VBScript</i>

Source Code for SCRIPDEM.FRM

```

`SCRIPDEM
Option Explicit

Dim mfilScript As New Scripting.FileSystemObject
Dim mtexScript As TextStream
Dim mblnFileChanged As Boolean

Private Sub Form_Load()
    'Center this form
    Me.Move (Screen.Width - Me.Width) \ 2, _
        (Screen.Height - Me.Height) \ 2
End Sub

Private Sub Form_Resize()
    'Resize edit area to fill form
    txtModule.Move 0, tlbToolbar.Height,
        Me.ScaleWidth, Me.ScaleHeight - tlbToolbar.Height
End Sub

Private Sub mnuAbout_Click()
    'Set properties and display About form
    About.Application = "ScripDem"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuJava_Click()
    mnuVB.Checked = False
    mnuJava.Checked = True
    scrVB.Language = "JScript"
End Sub

Private Sub mnuOpen_Click()
    'If previous file has changed, prompt user to save
    If mblnFileChanged Then mnuNew_Click
    'Display Open File common dialog box
    cdlOne.DialogTitle = "Open File"
    cdlOne.ShowOpen
    'If user doesn't select a file, exit procedure
    If cdlOne.FileName = "" Then Exit Sub
    'Create text stream from FileSystemObject object
    Set mtexScript = mfilScript.OpenTextFile(
        cdlOne.FileName, ForReading, True)
    'If file is empty or doesn't exist, don't read it
    If mtexScript.AtEndOfFile Then
        'Clear text box
        txtModule.Text = ""
    Else
        'Load text from text stream into text box

```

```

        txtModule.Text = mtexScript.ReadAll
End If
`Close file and text stream
Set mtexScript = Nothing
`Reset mblnFileChanged flag
mblnFileChanged = False
`Update list of procedures
mnuUpdate_Click
End Sub

Private Sub mnuPrint_Click()
`Simple dump to printer
Printer.Print txtModule.Text
Printer.EndDoc
End Sub

Private Sub mnuSave_Click()
`Display Save File common dialog box
cdlOne.DialogTitle = "Save File"
cdlOne.ShowSave
`If user cancels, exit
If cdlOne.CancelError Then Exit Sub
`Create text stream using FileSystemObject object
Set mtexScript = mfilScript.OpenTextFile( _
    cdlOne.FileName, ForWriting, True)
`Write text box text to text stream
mtexScript.Write txtModule.Text
`Close text stream and FileSystemObject object
Set mtexScript = Nothing
`Reset mblnFileChanged flag
mblnFileChanged = False
`Update procedures
mnuUpdate_Click
End Sub

Private Sub mnuRunSep2_Click(Index As Integer)
`Catch runtime errors
On Error GoTo RunErr
`Run selected procedure
scrVB.Run scrVB.Procedures(Index).Name
Exit Sub
RunErr:
    HandleError
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

Private Sub mnuUpdate_Click()
`Check for errors when code is parsed during
`loading
On Error GoTo LoadErr
`Clear code
scrVB.Reset
`Add new code to Script control
scrVB.AddCode txtModule.Text
Dim proItem As Procedure
Dim intCount As Integer
`Unload menu items
For intCount = 1 To mnuRunSep2.UBound
    Unload mnuRunSep2(intCount)

```

```
Next intCount
intCount = 0
`Add procedure names to menu
For Each proItem In scrVB.Procedures
    intCount = intCount + 1
    Load mnuRunSep2(intCount)
    mnuRunSep2(intCount).Caption = proItem.Name
Next proItem
Exit Sub

LoadErr:
    HandleError
End Sub

Private Sub mnuNew_Click()
    Dim intSave As Integer
    `Check whether file has changed
    CheckForChange
    txtModule.Text = ""
    mblnFileChanged = False
    `Update procedure list
    mnuUpdate_Click
End Sub

Private Sub mnuExit_Click()
    Dim intSave As Integer
    `Check whether file has changed
    CheckForChange
    `Unload form (ends application)
    Unload Me
End Sub

Private Sub mnuVB_Click()
    mnuVB.Checked = True
    mnuJava.Checked = False
    scrVB.Language = "VBScript"
End Sub

Private Sub tlbToolbar_ButtonClick(ByVal Button As ComctlLib.Button)
    Select Case Button.Key
        Case "New"
            mnuNew_Click
        Case "Open"
            mnuOpen_Click
        Case "Save"
            mnuSave_Click
        Case "Print"
            mnuPrint_Click
        Case "Cut"
            txtModule.SetFocus
            SendKeys "^x"
        Case "Copy"
            txtModule.SetFocus
            SendKeys "^c"
        Case "Paste"
            txtModule.SetFocus
            SendKeys "^v"
        Case "Run"
            PopupMenu mnuRun
        Case Else
            MsgBox "Unknown toolbar button clicked!", vbCritical
    End Select
End Sub
```

```
Private Sub txtModule_Change()
    'Set mblnFileChanged flag
    mblnFileChanged = True
End Sub

Private Sub CheckForChange()
    If mblnFileChanged Then
        Dim intSave As Integer
        'If file has changed, prompt user to save changes
        intSave = MsgBox("File has changed, save changes?", _
            vbYesNo, "Save Changes?")
        'If user decides to save, run Save menu procedure
        If intSave = vbYes Then
            mnuSave_Click
        End If
    End If
End Sub

Private Sub HandleError()
    Dim intCount As Integer
    With scrVB.Error
        'Move cursor to beginning
        txtModule.SelStart = 0
        'Move cursor to line with error
        For intCount = 1 To .Line
            SendKeys "{Down}", True
        Next intCount
        'Move cursor to column with error
        For intCount = 1 To .Column
            SendKeys "{Right}", True
        Next intCount
        'Display error
        If .Number <> 0 Then
            MsgBox Join(Array(.Source, .Number, .Description)), _
                vbCritical
        End If
    End With
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub
```

Chapter Thirty-One

Date and Time

This chapter provides several applications to handle date and time tasks: a date selection demonstration (VBCal) that enhances the way users of your applications can select a date, a visually appealing analog clock (VBClock) that displays the current system time, and a utility that dials in to the National Institute of Standards and Technology (NIST) to set your computer's clock precisely.

The VBCal Application

A common task in many business applications is to let the user select a date. The easiest way to do this is simply to have the user type a date into a text box. This method, however, requires careful attention to validation of the entered date and to internationalization issues—for example, does the string 3/5/99 indicate March 5 or May 3? A better approach is to display one of the three new calendar-related controls and let the user select the date of choice. The VBCal application demonstrates these three different controls.

It's a Wizard

I've designed this application as a simple wizard, which is a useful concept in itself. The wizard metaphor, first used by Microsoft in several products, is a convenient and standard way to perform interactions with the user in a linear, sequential manner. The VBCal project contains five forms of nearly identical appearance that are used to step through the wizard action. The code module VBCALWIZ.BAS contains only three lines of code, which make the dates the user selected available to all the forms. These forms have two command buttons, a visual element in the top half of each form, and guide the user by displaying step-by-step instructions. Three of the forms prompt the user to enter or select a date using one of three different but similar controls. In general, most wizards are laid out in the same general way to guide the user through a sequence of steps. Figures 31-1 through 31-5 show these wizard forms in action.



Figure 31-1. Step 1 displays the introductory screen VBCal Wizard.



Figure 31-2. Step 2 of the VBCal Wizard, prompting for the first date using the DTPicker control.

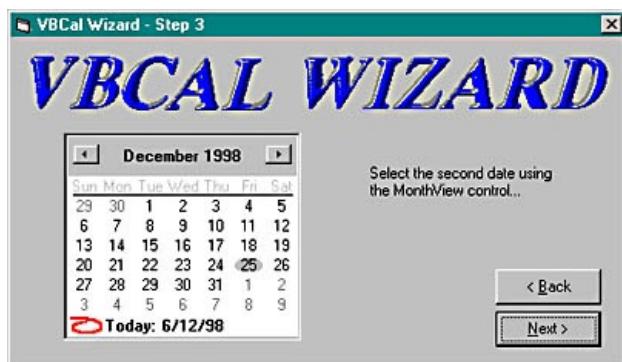
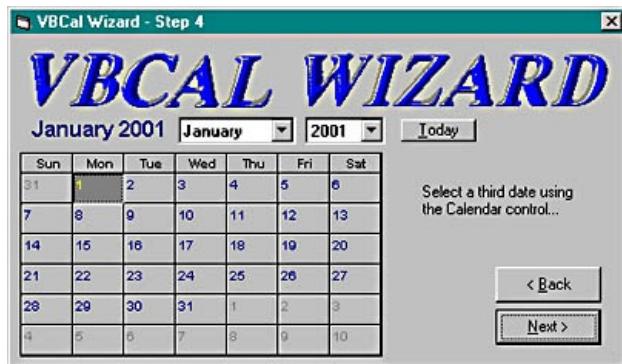
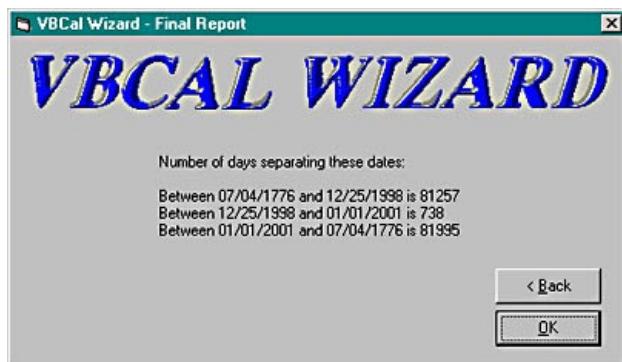


Figure 31-3. Step 3 of the VBCal Wizard, prompting for the second date using the MonthView control.**Figure 31-4.** Step 4 of the VBCal Wizard, prompting for the third date using the Calendar control.**Figure 31-5.** The report display for the VBCal Wizard, displaying the final results.

DTPicker Control

In the second step of the wizard action, after the introductory screen, the user is prompted for a date using the DTPicker control. This control lets the user type in a date by means of the keyboard, or click the drop-down arrow to select a date from a small calendar window. The control's CustomFormat property lets you define the date (and/or time) format in a completely flexible way, and the control handles all the details of highlighting the fields, automatically validating the entries and displaying the small calendar. As shown in Figure 31-2, I used a custom date format that includes a prompt, and includes all four digits for entering the year in a manner that will avoid the Year 2000 (Y2K) problem which arises from processing years as two-digit numbers.

MonthView Control

The third step of this wizard uses the MonthView control to select a date. You'll notice that this small one-month calendar has a default appearance much like the small calendar displayed by the DTPicker control. Figure 31-3 shows the MonthView control in action as a date is selected.

This control has a number of options to control its appearance and functionality. For example, you can set the MonthColumns and MonthRows properties to values greater than 1 to show several months at a time, side by side, or stacked vertically. Be sure to experiment with the Font property, as a wide variety of calendar styles can be created by simply changing the control's font.

Calendar Control

The fourth wizard step, as shown in Figure 31-4, uses the new Calendar control to let the user select a date in yet another way. This control also displays a one-month calendar, but it has more formatting properties to control the details of its appearance, and the calendar can be resized as desired.

I added a small button next to the Calendar control, labeled *Today*, to instantly jump to the current date. You'll notice that I've added code to the double-click event to allow the user to double-click any date to select it. Figure 31-6 shows the Calendar control in action as a date of January 1, 2001, is selected.



Figure 31-6. The Calendar control in action.

VBCALWIZ.BAS

The VBCal forms interact with one another through three public variables declared in the VBCALWIZ.BAS module. The Date variables *gdtmDate1*, *gdtmDate2*, and *gdtmDate3* hold the three dates selected using each of the three types of date selection controls. As the user proceeds or reverses to each step of the wizard, these variables keep track of the last date selected by each control.

Figure 31-7 shows the project list for the VBCal application. The form *frmVBCalWiz1* is set as the startup form. Most of the files in this project define the steps of the wizard action. Each of the forms that constitutes a wizard step is sized the same; the command buttons are all in the same place, and the graphic captions are identical. This creates the illusion that each step of the wizard is accomplished on a single form. The source code, tables, and illustrations in this chapter describe the construction of each of these forms.

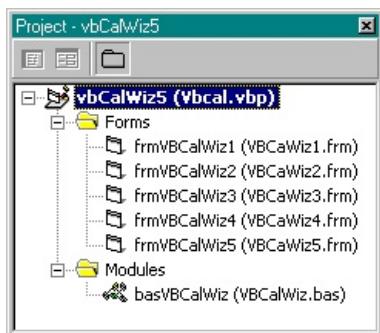


Figure 31-7. The VBCal project list.

To create the forms for this application, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

VBCAWIZ1.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmVBCalWiz1</i>
Caption	VBCal Wizard - Step 1
BorderStyle	3 - Fixed Dialog
Image	
Name	<i>imgCal</i>
Picture	<i>VBCAL.BMP</i>
Label	
Name	<i>lblPrompt</i>

CommandButton

Name	<i>cmdNext</i>
Caption	<i>&Next ></i>
Default	<i>True</i>

CommandButton

Name	<i>cmdCancel</i>
Caption	<i>&Cancel</i>
Cancel	<i>True</i>

Source Code for VBCAWIZ1.FRM

```

Option Explicit

Private Sub cmdCancel_Click()
    Unload Me
End Sub

Private Sub cmdNext_Click()
    frmVBCalWiz2.Show
    Unload Me
End Sub

Private Sub Form_Load()
    'Preset today's date
    If gdtmDate1 = 0 Then
        gdtmDate1 = Date
        gdtmDate2 = Date
        gdtmDate3 = Date
    End If
    'Display the prompting text
    lblPrompt.Caption = "This example wizard " & _
        "demonstrates three controls that let " & _
        "you select dates as it helps you " & _
        "calculate the number of days " & _
        "between two dates."
    'Center form
    Move (Screen.Width - Width) \ 2, _
        (Screen.Height - Height) \ 2
    Show
    cmdNext.SetFocus
End Sub

```

VBCAWIZ2.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmVBCalWiz2</i>
Caption	<i>VBCal Wizard - Step 2</i>
BorderStyle	<i>3 - Fixed Dialog</i>
Image	
Name	<i>imgCal</i>

Picture	<i>VBCAL.BMP</i>
Label	
Name	<i>lblPrompt</i>
Caption	<i>Select the first date using the DTPicker control</i> <i>Dear John, How Do I...</i>
CommandButton	
Name	<i>cmdNext</i>
Caption	<i>&Next ></i>
Default	<i>True</i>
CommandButton	
Name	<i>cmdBack</i>
Caption	<i>< &Back</i>
DTPicker	
Name	<i>dtpDemo</i>
Format	<i>3 - dtpCustom</i>
CustomFormat	<i>'Date format: mm/dd/yyyy</i> <i>Dear John, How Do I... ' MM/dd/yyyy</i>

Source Code for VBCAWIZ2.FRM

```

Option Explicit

Private Sub cmdBack_Click()
    gdtmDate1 = dtpDemo
    frmVBCalWiz1.Show
    Unload Me
End Sub

Private Sub cmdNext_Click()
    gdtmDate1 = dtpDemo
    frmVBCalWiz3.Show
    Unload Me
End Sub

Private Sub Form_Load()
    `Center form
    Move (Screen.Width - Width) \ 2, -
          (Screen.Height - Height) \ 2
    Show
    dtpDemo = gdtmDate1
    cmdNext.SetFocus
End Sub

```

VBCAWIZ3.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmVBCalWiz3</i>
Caption	<i>VBCal Wizard - Step 3</i>

BorderStyle	3 - Fixed Dialog
-------------	------------------

Image

Name	<i>imgCal</i>
Picture	<i>VBCAL.BMP</i>

Label

Name	<i>lblPrompt</i>
Caption	<i>Select the second date using the MonthView control</i> <i>Dear John, How Do I...</i>

CommandButton

Name	<i>cmdNext</i>
Caption	<i>&Next ></i>
Default	<i>True</i>

CommandButton

Name	<i>cmdBack</i>
Caption	<i>< &Back</i>

MonthView

Name	<i>mvwDemo</i>
------	----------------

Source Code for VBCAWIZ3.FRM

```

Option Explicit

Private Sub cmdBack_Click()
    gdtmDate2 = mvwDemo.Value
    frmVBCalWiz2.Show
    Unload Me
End Sub

Private Sub cmdNext_Click()
    gdtmDate2 = mvwDemo.Value
    frmVBCalWiz4.Show
    Unload Me
End Sub

Private Sub Form_Load()
    `Center form
    Move (Screen.Width - Width) \ 2,
          (Screen.Height - Height) \ 2
    Show
    mvwDemo.Value = gdtmDate2
    cmdNext.SetFocus
End Sub

Private Sub mvwDemo_DateDblClick(ByVal DateDblClicked As Date)
    `Allow double-click to select a date
    cmdNext_Click
End Sub

```

VBCAWIZ4.FRM Objects and Property Settings

Property	Value
----------	-------

Form	
Name	<i>frmVBCalWiz4</i>
Caption	<i>VBCal Wizard - Step 4</i>
BorderStyle	<i>3 - Fixed Dialog</i>
Image	
Name	<i>imgCal</i>
Picture	<i>VBCAL.BMP</i>
Label	
Name	<i>lblPrompt</i>
Caption	<i>Select a third date using the Calendar control/Dear John, How Do I...</i>
CommandButton	
Name	<i>cmdNext</i>
Caption	<i>&Next ></i>
Default	<i>True</i>
CommandButton	
Name	<i>cmdBack</i>
Caption	<i>< &Back</i>
CommandButton	
Name	<i>cmdToday</i>
Caption	<i>&Today</i>
Calendar	
Name	<i>calDemo</i>

Source Code for VBCAWIZ4.FRM

```

Option Explicit

Private Sub calDemo_DblClick()
    'Allow double-click selection of date
    cmdNext_Click
End Sub

Private Sub cmdBack_Click()
    gdtmDate3 = calDemo.Value
    frmVBCalWiz3.Show
    Unload Me
End Sub

Private Sub cmdNext_Click()
    gdtmDate3 = calDemo.Value
    frmVBCalWiz5.Show
    Unload Me
End Sub

Private Sub cmdToday_Click()
    'Reset selected date to today
    calDemo.Value = Date

```

```

End Sub

Private Sub Form_Load()
    'Center form
    Move (Screen.Width - Width) \ 2,
          (Screen.Height - Height) \ 2
    calDemo.Value = gdtmDate3
End Sub

```

VBCAWIZ5.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmVBCalWiz5</i>
Caption	<i>VBCal Wizard - Final Report</i>
BorderStyle	<i>3 - Fixed Dialog</i>
Label	
Name	<i>lblReport</i>
CommandButton	
Name	<i>cmdBack</i>
Caption	<i>< &Back</i>
CommandButton	
Name	<i>cmdOK</i>
Caption	<i>&OK</i>

Source Code for VBCAWIZ5.FRM

```

Option Explicit

Private Sub cmdOK_Click()
    Unload Me
End Sub

Private Sub cmdBack_Click()
    frmVBCalWiz4.Show
    Unload Me
End Sub

Private Sub Form_Load()
    'Center this form
    Move (Screen.Width - Width) \ 2,
          (Screen.Height - Height) \ 2
    lblReport.Caption = "Number of days separating " _
        & "these dates:" & vbCrLf & vbCrLf & "Between" " -
        & Format$(gdtmDate1, "mm/dd/yyyy") & " and " -
        & Format$(gdtmDate2, "mm/dd/yyyy") & " is " -
        & Abs(gdtnDate1 - gdtnDate2) & vbCrLf & "Between" " -
        & Format$(gdtnDate2, "mm/dd/yyyy") & " and " -
        & Format$(gdtnDate3, "mm/dd/yyyy") & " is " -
        & Abs(gdtnDate2 - gdtnDate3) & vbCrLf & "Between" " -
        & Format$(gdtnDate3, "mm/dd/yyyy") & " and " -
        & Format$(gdtnDate1, "mm/dd/yyyy") & " is "

```

```
& Abs(gdtmDate3 - gdtmDate1)  
End Sub
```

Source Code for VBCALWIZ.BAS

```
Option Explicit
```

```
Public gdtmDate1 As Date  
Public gdtmDate2 As Date  
Public gdtmDate3 As Date
```

The VBClock Application

This application creates a visually appealing and fun analog clock that displays the system time. The VBClock application also demonstrates several graphics techniques and an alternative approach to adding a generic About box to your applications. As shown in Figure 31-8, the VBClock form's background is a colorful bitmap graphic. I created this bitmap using a public domain program that generates fractals based on the Mandelbrot set, but you can load any bitmap file into the *picBackGround* PictureBox control to customize the clock.

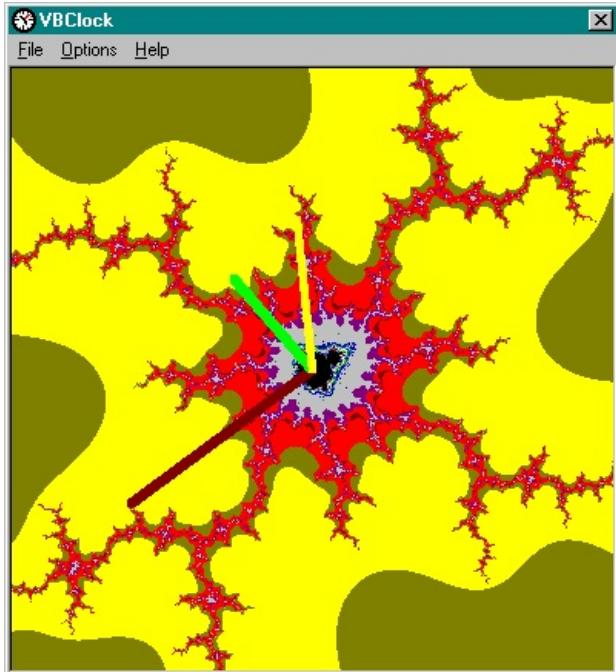


Figure 31-8. The VBClock application in action.

The Options menu provides selections to manually set the time and to change the colors of the clock's hands. To set the time, I added a simple InputBox function to the program. You might want instead to use a DTPicker control, which can easily be accomplished using a slight modification of the technique in the VBCal wizard demonstration earlier in this chapter. To change the hand colors, I used a dialog box form that demonstrates a useful hot spot graphics technique. The Help menu provides access to the Contents and Search entry points into the associated help file.

NOTE

A great way to update your system time is to use the NISTTime application, described later in this chapter. You can run NISTTime and VBClock simultaneously to see the clock adjustment in real time.

Figure 31-9 shows the project list for the VBClock application. Each of the three files is explained in more detail below.

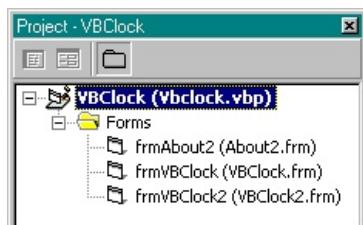


Figure 31-9. The VBClock project list.

VBCLOCK.FRM

The VBClock form displays the analog clock image and updates the clock's hands once per second. This form has only three controls: a PictureBox control to display the background and the clock hands, a Timer control to update the clock, and a CommonDialog control to provide the interface to the associated help file. You can load any bitmap image into the PictureBox control or change the startup hand colors to suit your taste.

NOTE

AutoRedraw is an important property of the PictureBox control. It should be set to *True* so that the clock hands are drawn smoothly and crisply. If AutoRedraw is set to *False*, you'll probably see some flickering of the image as the hands are erased and redrawn.

Figure 31-10 shows the VBClock form during development.

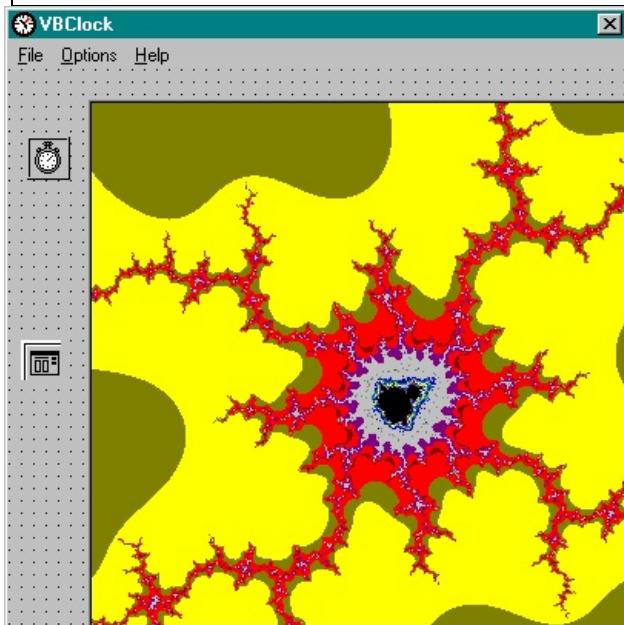


Figure 31-10. The VBClock form during development.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

VBCLOCK.FRM Menu Design Window Entries

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&OpenDear John, How Do I...	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &AsDear John, How Do I...	mnuSaveAs	1	<i>False</i>
-	mnuFileDash1	1	<i>True</i>
&Exit	mnuExit	1	<i>True</i>
&Options	mnuOption	0	<i>True</i>
&Set TimeDear John, How Do I...	mnuSetTime	1	<i>True</i>
&Hand ColorsDear John, How Do I...	mnuHandColors	1	<i>True</i>

<i>&Help</i>	<i>mnuHelp</i>	0	<i>True</i>
<i>&Contents</i>	<i>mnuContents</i>	1	<i>True</i>
<i>&Search For Help On</i> <i>Dear John, How Do I...</i>	<i>mnuSearch</i>	1	<i>True</i>
-	<i>mnuHelpDash1</i>	1	<i>True</i>
<i>&About</i> <i>Dear John, How Do I...</i>	<i>mnuAbout</i>	1	<i>True</i>

VBCLOCK.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmVBClock</i>
Caption	<i>VBClock</i>
BorderStyle	<i>3 - Fixed Dialog</i>
Timer	
Name	<i>tmrClock</i>
Interval	<i>100m</i>
CommonDialog	
Name	<i>cdlOne</i>
PictureBox	
Name	<i>picBackGround</i>
AutoRedraw	<i>True</i>
AutoSize	<i>True</i>
Picture	<i>MNDLBRT.BMP</i>

Source Code for VBCLOCK.FRM

```

Option Explicit

Public gintHourHandColor As Integer
Public gintMinuteHandColor As Integer
Public gintSecondHandColor As Integer
Private mintHnum As Integer
Private mintMnum As Integer
Private mintSnum As Integer
Private mlngHcolor As Long
Private mlMcolor As Long
Private mlngScolor As Long
Private msngHlen As Single
Private msngMlen As Single
Private msngSlen As Single
Private mstrAppname As String
Private mstrSection As String
Private mstrKey As String
Private mstrSetting As String

Private Const sngPi = 3.141593!
Private Const sngTwoPi = sngPi + sngPi
Private Const sngHalfPi = sngPi / 2

```

```

Private Sub Form_Load()
    'Fill form exactly with background image
    picBackGround.Move 0, 0
    Me.Width = picBackGround.Width + (Me.Width - ScaleWidth)
    Me.Height = picBackGround.Height + (Me.Height - ScaleHeight)
    'Change the scaling of the clock face
    picBackGround.Scale (-2, -2)-(2, 2)
    'Center form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
    'Set width of hands in pixels
    picBackGround.DrawWidth = 5
    'Set length of hands
    msngHlen = 0.8
    msngMlen = 1.5
    msngSlen = 1
    'Set colors of hands from Registry settings
    mstrAppname = "VBClock"
    mstrSection = "Hands"
    mstrKey = "mlngHcolor"
    mstrSetting = GetSetting(mstrAppname, mstrSection, mstrKey)
    gintHourHandColor = Val(mstrSetting)
    mstrKey = "mlMcolor"
    mstrSetting = GetSetting(mstrAppname, mstrSection, mstrKey)
    gintMinuteHandColor = Val(mstrSetting)
    mstrKey = "mlngScolor"
    mstrSetting = GetSetting(mstrAppname, mstrSection, mstrKey)
    gintSecondHandColor = Val(mstrSetting)
End Sub

Private Sub Form_Unload(Cancel As Integer)
    'Save current hand colors
    mstrKey = "mlngHcolor"
    mstrSetting = Str$(gintHourHandColor)
    SaveSetting mstrAppname, mstrSection, mstrKey, mstrSetting
    mstrKey = "mlMcolor"
    mstrSetting = Str$(gintMinuteHandColor)
    SaveSetting mstrAppname, mstrSection, mstrKey, mstrSetting
    mstrKey = "mlngScolor"
    mstrSetting = Str$(gintSecondHandColor)
    SaveSetting mstrAppname, mstrSection, mstrKey, mstrSetting
End Sub

Private Sub mnuAbout_Click()
    frmAbout2.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuHandColors_Click()
    'Show form for selecting hand colors
    frmVBClock2.Show vbModal
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"

```

```

cdlOne.HelpCommand = cdlHelpPartialKey
cdlOne.ShowHelp
End Sub

Private Sub tmrClock_Timer()
    Dim dblHang As Double
    Dim dblMang As Double
    Dim dblSang As Double
    Dim dblHx As Double
    Dim dblHy As Double
    Dim dblMx As Double
    Dim dblMy As Double

    Dim dblSx As Double
    Dim dblSy As Double
    'Keep track of current second
    Static intLastSecond As Integer
    'Check to see if new second
    If Second(Now) = intLastSecond Then
        Exit Sub
    Else
        intLastSecond = Second(Now)
    End If
    'Update time variables
    mintHnum = Hour(Now)
    mintMnum = Minute(Now)
    mintSnum = Second(Now)
    'Calculate hand angles
    dblHang = sngTwoPi * (mintHnum + mintMnum / 60) / 12 - sngHalfPi
    dblMang = sngTwoPi * (mintMnum + mintSnum / 60) / 60 - sngHalfPi
    dblSang = sngTwoPi * mintSnum / 60 - sngHalfPi
    'Calculate endpoints for each hand
    dblHx = msngHlen * Cos(dblHang)
    dblHy = msngHlen * Sin(dblHang)
    dblMx = msngMlen * Cos(dblMang)
    dblMy = msngMlen * Sin(dblMang)
    dblSx = msngSlen * Cos(dblSang)
    dblSy = msngSlen * Sin(dblSang)
    'Restore background image
    picBackGround.Cls
    'Draw new hands
    picBackGround.Line (0, 0)-(dblMx, dblMy), _
        QBColor(gintMinuteHandColor)
    picBackGround.Line (0, 0)-(dblHx, dblHy), _
        QBColor(gintHourHandColor)
    picBackGround.Line (0, 0)-(dblSx, dblSy), _
        QBColor(gintSecondHandColor)
End Sub

Private Sub mnuSetTime_Click()
    Dim strPrompt As String
    Dim strTitle As String
    Dim strDefault As String
    Dim strStartTime As String
    Dim strTim As String
    Dim strMsg As String
    'Ask user for new time
    strPrompt = "Enter the time, using the format 00:00:00"

    strTitle = "VBClock"
    strDefault = Time$
    strStartTime = strDefault
    strTim = InputBox$(strPrompt, strTitle, strDefault)
    'Check if user clicked Cancel

```

```

`or clicked OK with no change to time
If strTim = "" Or strTim = strStartTime Then
    Exit Sub
End If
`Set new time
On Error GoTo ErrorTrap
Time = strTim
Exit Sub
ErrorTrap:
strMsg = "The time you entered is invalid. " + strTim
MsgBox strMsg, 48, "VBClock"
Resume Next
End Sub

```

The Timer control's Interval property is set to *100 milliseconds* ($1/10$ of a second) instead of *1000 milliseconds* (1 second). The hands need to be updated at the rate of once per second, but setting the timer to a rate of once per second causes intermittent jerkiness in the movement of the hands. The jerkiness occurs because a Visual Basic timer isn't based on an exact timing between activations. Instead, the timer activates after the indicated interval, as soon as the system can get back to the timer after that interval has transpired. This causes a slight, unpredictable variation in the actual activations of timer events. (Activations are probably more accurate in faster computers and less accurate in slower ones.) Every so often, this unpredictable delay can cause the VBClock application's clock hands to jump to the next second erratically.

To fix this problem, set the timer's Interval property to something less than half a second. I chose *100 milliseconds*. The result is that the hands are updated much more accurately and are never off by more than about a tenth of a second. This updating action is smooth enough that the user won't notice any variation in the beat. The Timer event procedure checks the system time against the previously updated second shown by the hands of the clock. If a new second has not yet arrived, the procedure exits, causing very little delay to the system's overall speed.

Using the Registry

The user has the option of changing the color of the clock's hands. To maintain this color information between sessions, the application uses the Registry to retrieve the last known color settings during the form's Load event and saves the current color settings during the form's Unload event. This simple example demonstrates the use of the Registry to save the state of an application. Using these same techniques, you can use the Registry to store any details about the state of the application that you may want to add.

VBCLOCK2.FRM

The VBClock2 form provides a graphical way to select the colors for the three hands of the clock. My goal here was to allow the user to select one of the 16 main colors for each of the hands, but I didn't want to complicate the form with a lot of verbal descriptions of the colors. The solution was to draw all 16 colors in each of three picture boxes and let the user select the colors visually. Figure 31-11 shows the VBClock2 form in action. This approach is simple and uncluttered, and the user sees exactly what color he or she is selecting. Besides, like they always say, a picture is worth two thousand bytes!

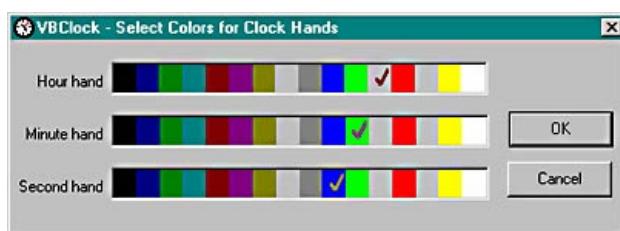


Figure 31-11. The VBClock2 form, which lets the user select clock hand colors.

During the form's Load event procedure, I scaled the three picture boxes to divide them into 16 regions.

The height of each picture box is scaled from 0 to 1, and the width from 0 to 16. Regardless of the actual size of the picture boxes, each one will display the 16 main colors in rectangular regions of equal size. Likewise, when the user clicks anywhere on a picture box, the x-coordinate converts to a number from 0 to 15, indicating the selected color.

Image Hot Spots

The Click event doesn't provide the information required to determine exactly where in a picture box the mouse click happened. The best way to determine the location of the click is to maintain the last known position of the mouse pointer using a global variable. The picture box's MouseMove event does provide x and y pointer location information, so it's easy to update the global variables (in this case, *mintHourMouseX*, *mintMinuteMouseX*, and *mintSecondMouseX*) to keep track of the mouse. The values stored in these variables at the time of a Click event determine what color the user is selecting. This technique can be used to define hot spots on your graphics. In fact, by mathematically scrutinizing the current xy-coordinates of the mouse pointer, you can define hot spots in your graphics that are circular, polygonal, or whatever shape and size you want.

Figure 31-12 shows the VBClock2 form during the development process.

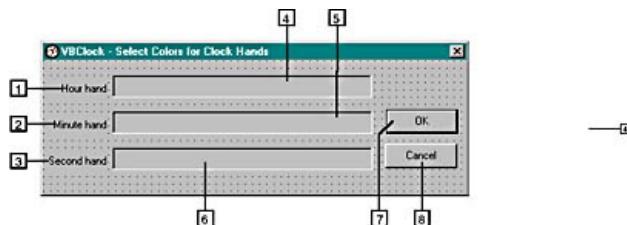


Figure 31-12. The VBClock2 form during development.

To create this form, use the table and source code in this chapter to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

VBCLOCK2.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmVBClock2</i>
	Caption	<i>VBClock - Select Colors for Clock Hands</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
	Icon	<i>CLOCK02 ICO</i>
Label		
1	Name	<i>IblHourHand</i>
	Caption	<i>Hour hand</i>
	Alignment	<i>1 - Right Justify</i>
Label		
2	Name	<i>IblMinuteHand</i>
	Caption	<i>Minute hand</i>
	Alignment	<i>1 - Right Justify</i>
Label		
3	Name	<i>IblSecondHand</i>
	Caption	<i>Second hand</i>
	Alignment	<i>1 - Right Justify</i>

PictureBox		
4	Name	<i>picHourColor</i>
PictureBox		
5	Name	<i>picMinuteColor</i>
PictureBox		
6	Name	<i>picSecondColor</i>
CommandButton		
7	Name	<i>cmdOK</i>
	Caption	<i>OK</i>
	Default	<i>True</i>
CommandButton		
8	Name	<i>cmdCancel</i>
	Caption	<i>Cancel</i>
	Cancel	<i>True</i>

* The number in the ID No. column corresponds to the number in Figure 31-12 that identifies the location of the object on the form.

Source Code for VBCLOCK2.FRM

```

Option Explicit

Private mintHourMouseX As Integer
Private mintMinuteMouseX As Integer
Private mintSecondMouseX As Integer
Private mintHourHand As Integer
Private mintMinuteHand As Integer
Private mintSecondHand As Integer

Private Sub cmdCancel_Click()
    'Cancel without changing hand colors
    Unload Me
End Sub

Private Sub cmdOK_Click()
    'Reset hand colors
    frmVBClock.gintHourHandColor = mintHourHand
    frmVBClock.gintMinuteHandColor = mintMinuteHand
    frmVBClock.gintSecondHandColor = mintSecondHand
    'Return to clock form
    Unload Me
End Sub

Private Sub Form_Activate()
    Form_Paint
End Sub

Private Sub Form_Load()
    'Get current hand colors
    mintHourHand = frmVBClock.gintHourHandColor
    mintMinuteHand = frmVBClock.gintMinuteHandColor
    mintSecondHand = frmVBClock.gintSecondHandColor
    'Scale picture boxes

```

```

picHourColor.Scale (0, 0)-(16, 1)
picMinuteColor.Scale (0, 0)-(16, 1)
picSecondColor.Scale (0, 0)-(16, 1)
End Sub

Private Sub Form_Paint()
    Dim i As Integer
    'Draw the 16 colors in each color "bar"
    For i = 0 To 15
        'Draw colored boxes
        picHourColor.Line (i, 0)-(i + 1, 1), QBColor(i), BF
        picMinuteColor.Line (i, 0)-(i + 1, 1), QBColor(i), BF
        picSecondColor.Line (i, 0)-(i + 1, 1), QBColor(i), BF
    Next i
    'Draw check marks for current colors

    picHourColor.DrawWidth = 2
    picHourColor.Line (mintHourHand + 0.3, 0.5) _
        -(mintHourHand + 0.5, 0.7), QBColor(mintHourHand Xor 15)
    picHourColor.Line (mintHourHand + 0.5, 0.7) _
        -(mintHourHand + 0.8, 0.2), QBColor(mintHourHand Xor 15)
    picMinuteColor.DrawWidth = 2
    picMinuteColor.Line (mintMinuteHand + 0.3, 0.5) _
        -
        (mintMinuteHand + 0.5, 0.7), QBColor(mintMinuteHand Xor 15)
    picMinuteColor.Line (mintMinuteHand + 0.5, 0.7) _
        -
        (mintMinuteHand + 0.8, 0.2), QBColor(mintMinuteHand Xor 15)
    picSecondColor.DrawWidth = 2
    picSecondColor.Line (mintSecondHand + 0.3, 0.5) _
        -
        (mintSecondHand + 0.5, 0.7), QBColor(mintSecondHand Xor 15)
    picSecondColor.Line (mintSecondHand + 0.5, 0.7) _
        -
        (mintSecondHand + 0.8, 0.2), QBColor(mintSecondHand Xor 15)
End Sub

Private Sub picHourColor_Click()
    'Determine selected hour hand color
    mintHourHand = mintHourMouseX
    Form_Paint
End Sub

Private Sub picHourColor_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    'Keep track of mouse location
    mintHourMouseX = Int(X)
End Sub

Private Sub picMinuteColor_Click()
    'Determine selected minute hand color
    mintMinuteHand = mintMinuteMouseX
    Form_Paint
End Sub

Private Sub picMinuteColor_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    'Keep track of mouse location
    mintMinuteMouseX = Int(X)
End Sub

Private Sub picSecondColor_Click()
    'Determine selected second hand color
    mintSecondHand = mintSecondMouseX

```

```

Form_Paint
End Sub

Private Sub picSecondColor_MouseMove(Button As Integer, _
Shift As Integer, X As Single, Y As Single)
    'Keep track of mouse location
    mintSecondMouseX = Int(X)
End Sub

```

ABOUT2.FRM

In Chapter 12, "[Dialog Boxes, Windows, and Other Forms](#)," I provided a generic, fill-in-the-blanks About dialog box that you could easily plug into your own applications. To further demonstrate the App object's properties and how they can automate the About dialog box even further, I created ABOUT2.FRM. This form is completely generic in that all displayed data is modified by the calling application, yet the only line of source code required to activate this form is a call to the form's Display method.

Figure 31-13 shows the About2 form as it is displayed by the VBClock application.

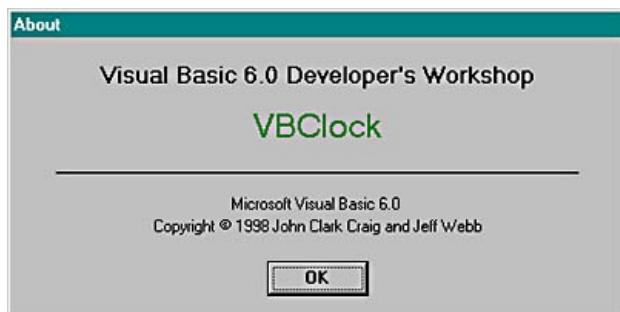


Figure 31-13. The About2 form as activated by VBClock.

So when, where, and how does the information displayed by the About2 form get set? The trick is to set the properties of the App object during the development process. From the Visual Basic Project menu, choose Project Properties to display the Project Properties dialog box, and then click the Make tab, as shown in Figure 31-14.

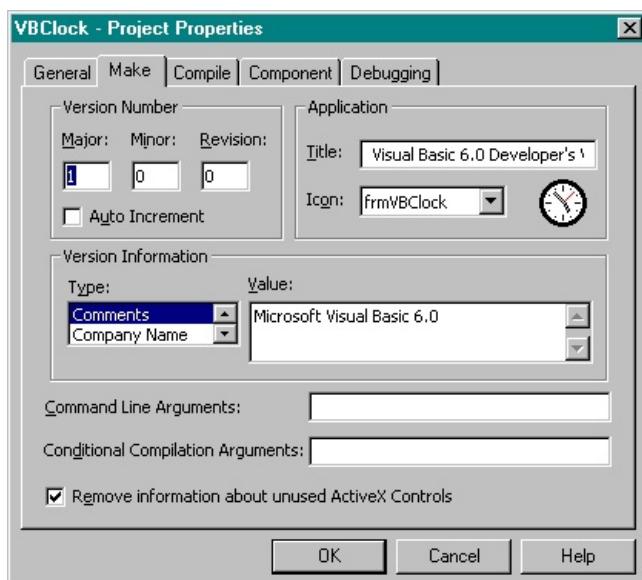


Figure 31-14. The Project Properties dialog box, in which many of the App properties are set.

Visual Basic's App object provides a long list of useful properties, many of which you can set using the Project Properties dialog box. The version numbers, title, and properties listed in the Version Information section of the Make tab can be set here, and they can be accessed programmatically by referring to properties of the App object.

The About2 form accesses four App properties to set the Caption properties of four Label controls. Here's the code that loads these labels, which you'll find in the form's Load event procedure:

```
' Set labels using App properties
lblHeading.Caption = App.Title
lblApplication.Caption = App.ProductName
lblVB.Caption = App.Comments
lblCopyright.Caption = App.LegalCopyright
```

Feel free to modify the About2 form to display any other App properties you want. For example, you might want to display the version number information or the trademark text. I chose only four of the properties in order to make this About dialog box similar to the original About dialog box so that you can easily compare the two techniques. Figure 31-15 shows the About2 form during development.

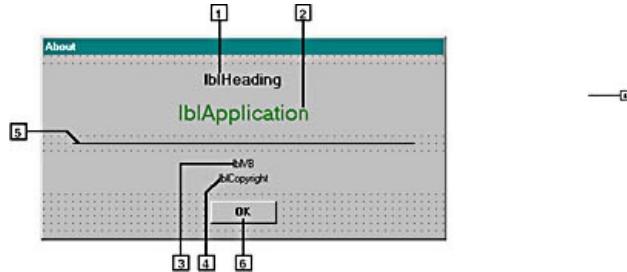


Figure 31-15. The About2 form during development.

To create this form, use the following table and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

ABOUT2.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmAbout2</i>
	Caption	<i>About</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
Label		
1	Name	<i>lblHeading</i>
	Alignment	<i>2 - Center</i>
	Font	<i>MS Sans Serif, Bold, 12</i>
Label		
2	Name	<i>lblApplication</i>
	Alignment	<i>2 - Center</i>
	Font	<i>MS Sans Serif, Regular, 18</i>
	ForeColor	<i>&H00008000&</i>
Label		
3	Name	<i>lblVB</i>
	Alignment	<i>2 - Center</i>
Label		
4	Name	<i>lblCopyright</i>
	Alignment	<i>2 - Center</i>

Line

5	Name	<i>linSeparator</i>
---	------	---------------------

CommandButton

6	Name	<i>cmdOK</i>
	Caption	<i>OK</i>

* The number in the ID No. column corresponds to the number in Figure 31-15 that identifies the location of the object on the form.

Source Code for ABOUT2.FRМ

```
Option Explicit

Private Sub cmdOK_Click()
    Unload Me
End Sub

Private Sub Form_Load()
    'Center form
    Left = (Screen.Width - Width) \ 2
    Top = (Screen.Height - Height) \ 2
    'Set labels using App properties
    lblHeading.Caption = App.Title
    lblApplication.Caption = App.ProductName
    lblVB.Caption = App.Comments
    lblCopyright.Caption = App.LegalCopyright
End Sub

Public Sub Display()
    'Display self as modal
    Me.Show vbModal
End Sub
```

The NISTTime Application

This little utility automatically dials up and connects with the National Institute of Standards and Technology (NIST) in Boulder, Colorado. You can use it to adjust your computer's clock to something better than 1-second accuracy. I say "something better" because various factors can limit the efficiency of the processing, such as modem throughput speed, the speed and efficiency of your computer at setting its clock, telephone line delays, and so on. The NIST time service does attempt to adjust for this delay by sending the "on mark" character 45 milliseconds before the actual time. As a result, the NISTTime application can set your clock's time far more accurately than you can set it manually.

How NISTTime Works

The NISTTime application uses the MSComm control to handle the modem connection over the telephone to the NIST facility. This control makes it simple to set up the correct baud rate, telephone number, and other modem parameters, as shown in the relevant lines of code in the form's Load event procedure. Originally, the baud rate for this service was limited to either 300 or 1200 baud, but NIST has installed modems that adjust automatically to the baud rate of the modem making the call. As shown in the source code listing, I've set my modem speed to 14,400 baud, which has worked fine for me. If you experience any line noise or inconsistent connections, try a lower baud rate.

NOTE

At 300 baud, the NIST service sends a condensed string of information incompatible with this application. Use 1200 baud or higher for best results.

Once connected, the service sends a header message, followed by a line of information repeated once per second. At the end of this line of data, an asterisk is transmitted, marking the exact time. The NISTTime application reads and stores this data as it arrives, watching for two occurrences of the asterisk time marks. When the second asterisk arrives, the NISTTime application extracts minute and second information from the string of data and sets the system clock to these values.

The string of incoming data provides other useful information, such as the modified Julian date; the year, month, day, and hour numbers for Coordinated Universal Time; and even a number indicating a possible leap second adjustment at the end of the current month. All of this extra data is ignored in the NISTTime application because your clock is probably already close to the correct local time and only an adjustment for minutes and seconds will be necessary.

NOTE

To get a full accounting and description of the information broadcast by this service, dial in to the same number using a terminal program such as HyperTerminal. Once you are connected, type a question mark to receive several pages of detailed information.

The NISTTime form uses two timers to control the communications interaction: one timer's interval is set to 1 millisecond to process incoming bytes; the other control is set to trigger after 1 minute. Normally, the NISTTime application makes the connection, sets your clock, and hangs up just a few seconds after dialing. However, if anything is amiss, the 1-minute timer disconnects the phone at the end of a minute. The service itself disconnects after roughly 1 minute of operation—all of which makes the chances of accidentally running up long-distance bills highly unlikely. I added this second watchdog timer as a redundant safety mechanism.

As NISTTime runs, several short messages are displayed to indicate its progress. No interaction with the user is required, and the application unloads shortly after the clock is set. Figures 31-16 through 31-18 show the normal sequence of these messages during the few seconds of operation.

By default, this application assumes your modem is on serial port 1, your baud rate is 14,400 baud, and the call to Boulder, Colorado, is a long-distance call to area code 303. However, I've set up a scheme that

lets you make easy adjustments to these parameters. The first time you run NISTTime, a special text file named NISTTIME.TXT is created in the application's folder. The default contents are as follows:

```
PORt 1  
TELEPHONE 1-303-494-4774  
SETTings 14400,N,8,1
```

This file can be edited with any text editor, such as NotePad, to change the port number, telephone number, or modem settings as required for your system. For example, on my system the modem is on port 2 and it's a local call to Boulder from my home. Here's how I edited the NISTTIME.TXT file for proper operation on my system:

```
PORt 2  
TELEPHONE 494-4774  
SETTings 14400,N,8,1
```

NOTE

To monitor your system's time, either run the VBClock program presented in the previous section, or double-click on the time displayed in your task bar to display the Date/Time Properties dialog box.



Figure 31-16. The NISTTime application as it dials in to NIST.



Figure 31-17. The application receiving the data used to set the system clock.



Figure 31-18. The NISTTime message displayed while disconnecting from NIST.

NISTTIME.FRM

The NISTTime form is small—just big enough to display progress messages in the upper-left corner of your display. The form contains two Timer controls, an MSComm control, and a Label control to display progress messages.

Figure 31-19 shows the form during development.

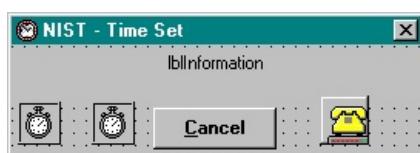


Figure 31-19. The NISTTime form during development.

To create this form, use the table and source code in this chapter to add the appropriate controls, set any

nondefault properties as indicated, and enter the source code lines as shown.

NISTTIME.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmNISTTime</i>
Caption	<i>NIST - Time Set</i>
BorderStyle	<i>3 - Fixed Dialog</i>
Icon	<i>CLOCK01 ICO</i>
Timer	
Name	<i>tmrGetBytes</i>
Interval	<i>1</i>
Enabled	<i>False</i>
Timer	
Name	<i>tmrWatchDog</i>
Interval	<i>60000</i>
Enabled	<i>False</i>
MSComm	
Name	<i>comControl</i>
Handshaking	<i>2 - comRTS</i>
CommandButton	
Name	<i>cmdCancel</i>
Caption	<i>&Cancel</i>
Label	
Name	<i>lblInformation</i>
Alignment	<i>2 - Center</i>

Source Code for NISTTIME.FRM

```

Option Explicit

Const PORT = 1
Const TELEPHONE = "1-303-494-4774"
Const SETTINGS = "14400,N,8,1"
Const BUFSIZ = 3000

Dim intNistNdx As Integer
Dim strNistBuf As String * BUFSIZ
Dim strTelephone As String
Dim strSettings As String
Dim intPort As Integer
Dim strA As String
Dim intP As Integer

Private Sub cmdCancel_Click()
    Unload Me

```

```

End Sub

Private Sub Form_Load()
    'Locate form near upper-left corner
    Me.Left = (Screen.Width - Me.Width) * 0.1
    Me.Top = (Screen.Height - Me.Height) * 0.1
    'Display first informational message

    lblInformation.Caption =
        "Dialing National Institute of Standards " & _
        "and Technology Telephone Time Service"
    'Show form and first message
    Show
    'Load port and telephone number from text file
    strTelephone = TELEPHONE
    intPort = PORT
    strSettings = SETTINGS
    Open App.Path & "\Nisttime.txt" For Binary As #1
    If LOF(1) = 0 Then
        Close #1
        'Create file the first time
        Open App.Path & "\Nisttime.txt" For Output As #1
        Print #1, "PORT" & Str(intPort)
        Print #1, "TELEPHONE " & strTelephone
        Print #1, "SETTINGS " & strSettings
    End If
    Close #1
    Open App.Path & "\Nisttime.txt" For Input As #1
    Do Until EOF(1)
        Line Input #1, strA
        strA = UCase$(strA)
        intP = InStr(strA, "PORT")
        If intP Then intPort = Val(Mid(strA, intP + 4))
        intP = InStr(strA, "TELEPHONE")
        If intP Then strTelephone = Mid(strA, intP + 9)
        intP = InStr(strA, "SETTINGS")
        If intP Then strSettings = Mid(strA, intP + 8)
    Loop
    Close #1
    'Set up MSComm control parameters
    comControl.CommPort = intPort
    comControl.SETTINGS = strSettings
    'Set to read entire buffer
    comControl.InputLen = 0
    comControl.PortOpen = True
    'Send command to dial NIST
    comControl.Output = "ATDT" + strTelephone + vbCr
    'Activate timers
    tmrGetBytes.Enabled = True
    tmrWatchDog.Enabled = True
    'Enable Cancel button

    cmdCancel.Enabled = True
End Sub

Private Sub Form_Unload(Cancel As Integer)
    'This usually hangs up phone
    comControl.DTREnable = False
    'The following also hangs up phone
    Pause 1500
    'Update message for user
    lblInformation.Caption = "Hanging up"
    Refresh
    'Send commands to control modem

```

```

comControl.Output = "+++"  

Pause 1500  

comControl.Output = "ATH0" + vbCrLf  

`Close down communications  

comControl.PortOpen = False  

End Sub

Private Sub Pause(dblMillisecond As Double)  

    Dim sngEndOfPause As Single  

    `Determine end time of delay  

    sngEndOfPause = Timer + dblMillisecond / 1000  

    `Loop away time  

    Do  

        Loop While Timer < sngEndOfPause  

End Sub

Private Sub SetTime(strA As String)  

    Dim intHo As Integer  

    Dim intMi As Integer  

    Dim intSe As Integer  

    Dim dblTimeNow As Double  

    `Extract current hour from system  

    intHo = Hour(Now)  

    `Extract minute and second from NIST string  

    intMi = Val(Mid(strA, 22, 2))  

    intSe = Val(Mid(strA, 25, 2))  

    `Construct new time  

    dblTimeNow = TimeSerial(intHo, intMi, intSe)  

    `Set system clock  

    Time = Format(dblTimeNow, "hh:mm:ss")  

End Sub

Private Sub tmrGetBytes_Timer()  

    Static blnConnect As Boolean  

    Dim strTmp As String  

    Dim intBytes As Integer  

    Dim intP1 As Integer  

    Dim intP2 As Integer  

    `Check for incoming bytes  

    If comControl.InBufferCount = 0 Then  

        Exit Sub  

    Else  

        strTmp = comControl.Input  

        intBytes = Len(strTmp)  

        If intBytes + intNistNdx >= BUFSIZ Then  

            lblInformation.Caption = "Hanging up"  

            tmrGetBytes.Enabled = False  

            tmrWatchDog.Enabled = False  

            Unload Me  

        Else  

            Mid(strNistBuf, intNistNdx + 1, intBytes) = strTmp  

            intNistNdx = intNistNdx + intBytes  

        End If  

    End If  

    `Check for sign that we've connected  

    If blnConnect = False Then  

        If InStr(strNistBuf, "*" & vbCrLf) Then  

            lblInformation.Caption = "Connected. Setting clock"  

            blnConnect = True  

        End If  

    Else  

        `Check for time marks  

        intP1 = InStr(strNistBuf, "*")  

        intP2 = InStr(intP1 + 1, strNistBuf, "*")
    End If
End Sub

```

```
'Time received if two time marks found
If intP2 > intP1 Then
    SetTime Mid(strNistBuf, intP1, intP2 - intP1 + 1)
    Unload Me
End If
End If
End Sub

Private Sub tmrWatchDog_Timer()
    'Activate safety timeout if no connection
    Beep
    Unload Me
End Sub
```

Chapter Thirty-Two

Databases

Visual Basic is playing an increasingly important role in the corporate environment, especially since its database capabilities have been enhanced to make it a powerful development and front-end tool. The three applications in this chapter—AreaCode, DataDump, and Jot—are examples of just how easy it is to work with Microsoft Access or other databases using Visual Basic.

The AreaCode Application

The AreaCode application consists of two forms, AREACODE.FRM and the generic ABOUT.FRM. AREACODE.FRM uses two Data controls to access the same database of telephone area codes in different ways. The first Data control, in the top half of the form, accesses a table of area codes in the database file AREACODE.MDB, which was created using Microsoft Access 97. The Data control in the bottom half of the form accesses a query in the same database file. The table tapped by the first Data control presents area code data in ascending (201 through 970) area code order. The query provides the same table of area code data in ascending state abbreviation order (AK through WY). This application makes it easy either to look up where a call came from when all you know is the area code or to look up an area code for a known state.

The single form for the AreaCode application uses few database-related lines of code because the Data control and several TextBox controls have properties that simplify the connection to the database. To create the form, I drew the controls and then set a handful of properties to make the connection to the database. I added a few lines of code to handle minor details of the search process—and then I was done!

Connecting controls to a database in this way is a two-step process. First a Data control is connected to the database file by setting the DatabaseName property, and then it is connected to a specific table or query within that database by setting the RecordSource property. In general, each Data control connects to a single table or query. Many other types of controls, such as Label and ListBox controls, now have properties that make the final connection to the database by connecting to a specific Data control. In the AreaCode application, I've used several TextBox controls. All I had to do was set each TextBox control's DataSource property to the name of one of the Data controls and then set each of their DataField properties to indicate the specific field within the table or query.

Although you can navigate the database table or query by clicking the Data control, I've added some code to demonstrate another way to manipulate the database records. In the top half of the form, the *txtAreaCode* text box lets the user type in a three-digit area code. After the user enters the three digits, a bookmark is set at the current record (so that the current record can be displayed again if the search fails), and several RecordSet properties of the Data control are set to cause the *txtAreaCode_Change* event procedure to begin searching for a matching area code in the database immediately.

Similarly, when the user types a two-character state abbreviation in the *txtStateAbbrev1* text box in the bottom half of the form, a search for the state abbreviation is performed and the first record for that state is displayed. The user can then click the Next Record button of the Data control to see the records with the additional area codes for that state.

Figure 32-1 shows the AreaCode form after a successful search for the area code 303 in the top half and after a successful search for the state of Washington in the bottom half. Note how the top and bottom halves of the form work independently of each other, even though they access the same database.

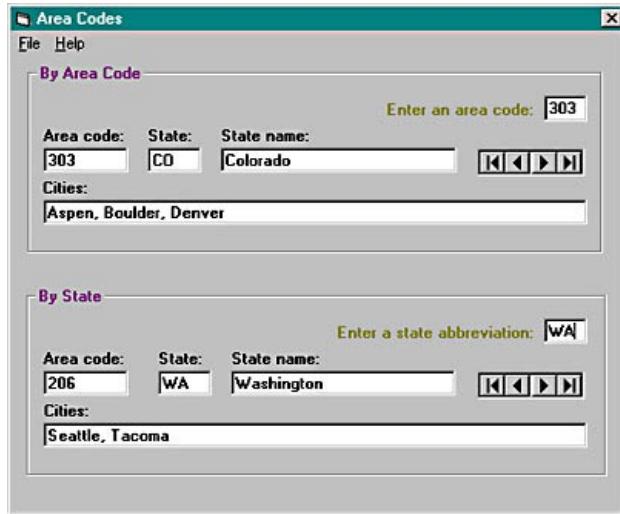


Figure 32-1. The AreaCode form after locating area code 303 in the top half and an area code for Washington state in the bottom half.

AREACODE.FRM

The AreaCode form contains two nearly identical sets of controls: one set for the top half of the form, and one set for the bottom half. Two Data controls connect to the same database, and several TextBox controls connect to fields through the associated Data control. To organize the TextBox controls, I've grouped them with some descriptive labels within Frame controls. A single CommonDialog control provides access to the associated help file by using the control's *ShowHelp* method. I assigned the name of the database file to the Data controls' *DatabaseName* property in the Form_Load event procedure, and I use the *App.Path* property to identify the location of the database. If your database is in a location other than the directory containing the application, assign the full path to the database.

Figure 32-2 shows the AreaCode form during development.

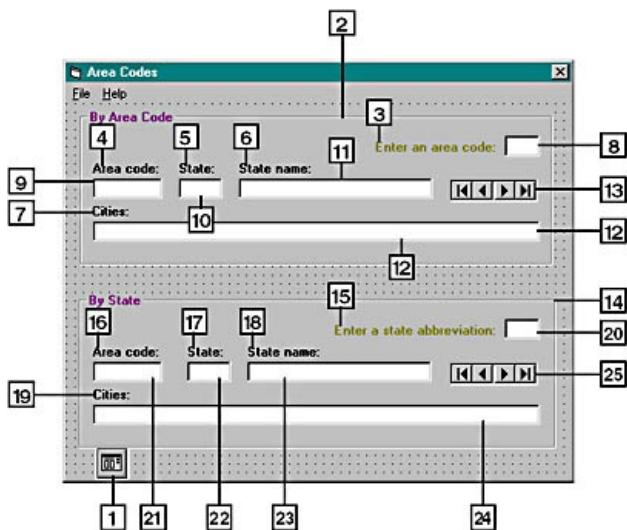


Figure 32-2. The AreaCode form during development.

To create this form, use the tables and source code below to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

AREACODE.FRM Menu Design Window Entries

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&OpenDear John, How Do I...	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &AsDear John, How Do I...	mnuSaveAs	1	<i>False</i>
-	mnuFileDash1	1	<i>True</i>
E&xit	mnuExit	1	<i>True</i>
&Help	mnuHelp	0	<i>True</i>
&Contents	mnuContents	1	<i>True</i>
&Search For Help OnDear John, How Do I...	mnuSearch	1	<i>True</i>
-	mnuHelpDash1	1	<i>True</i>
&AboutDear John, How Do I...	mnuAbout	1	<i>True</i>

AREACODE.FRM Objects and Property Settings

ID No.*	Property	Value
---------	----------	-------

Form		
	Name	<i>frmAreaCode</i>
	Caption	<i>Area Codes</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
CommonDialog		
1	Name	<i>cdlOne</i>
Frame		
2	Name	<i>fraByAreaCode</i>
	Caption	<i>By Area Code</i>
	ForeColor	<i>&H000000FF&</i>
Label		
3	Name	<i>lblPrompt2</i>
	Caption	<i>Enter an area code:</i>
	ForeColor	<i>&H00FF0000&</i>
Label		
4	Name	<i>lblAreaCode2</i>
	Caption	<i>Area code:</i>
Label		
5	Name	<i>lblState2</i>
	Caption	<i>State:</i>
Label		
6	Name	<i>lblStateName2</i>
	Caption	<i>State name:</i>
Label		
7	Name	<i>lblCities2</i>
	Caption	<i>Cities:</i>
TextBox		
8	Name	<i>txtAreaCode</i>
	Alignment	<i>2 - Center</i>
TextBox		
9	Name	<i>txtAreaCode2</i>
	DataSource	<i>datAreaCode2</i>
	DataField	<i>AreaCode</i>
TextBox		
10	Name	<i>txtState2</i>
	DataSource	<i>datAreaCode2</i>
	DataField	<i>State</i>

TextBox		
11	Name	<i>txtStateName2</i>
	DataSource	<i>datAreaCode2</i>
	DataField	<i>StateName</i>
TextBox		
12	Name	<i>txtCities2</i>
	DataSource	<i>datAreaCode2</i>
	DataField	<i>Cities</i>
Data		
13	Name	<i>datAreaCode2</i>
	RecordSource	<i>AreaCode</i>
Frame		
14	Name	<i>fraByState</i>
	Caption	<i>By State</i>
	ForeColor	<i>&H000000FF&</i>
Label		
15	Name	<i>lblPrompt1</i>
	Caption	<i>Enter a state abbreviation:</i>
	ForeColor	<i>&H00FF0000&</i>
Label		
16	Name	<i>lblAreaCode1</i>
	Caption	<i>Area code:</i>
Label		
17	Name	<i>lblState1</i>
	Caption	<i>State:</i>
Label		
18	Name	<i>lblStateName1</i>
	Caption	<i>State name:</i>
Label		
19	Name	<i>lblCities1</i>
	Caption	<i>Cities:</i>
TextBox		
20	Name	<i>txtStateAbbrev1</i>
	Alignment	<i>2 - Center</i>
TextBox		
21	Name	<i>txtAreaCode1</i>
	DataSource	<i>datAreaCode1</i>

	DataField	AreaCode
TextBox		
22	Name	<i>txtState1</i>
	DataSource	<i>datAreaCode1</i>
	DataField	<i>State</i>
TextBox		
23	Name	<i>txtStateName1</i>
	DataSource	<i>datAreaCode1</i>
	DataField	<i>StateName</i>
TextBox		
24	Name	<i>txtCities1</i>
	DataSource	<i>datAreaCode1</i>
	DataField	<i>Cities</i>
Data		
25	Name	<i>datAreaCode1</i>
	RecordSource	<i>ByState</i>

* The number in the ID No. column corresponds to the number in Figure 32-2 that identifies the location of the object on the form.

Source Code for AREACODE.FRM

```

Option Explicit

Private Sub Form_Load()
    'Center this form
    Me.Left = (Screen.Width - Width) \ 2
    Me.Top = (Screen.Height - Height) \ 2
    datAreaCode1.DatabaseName = App.Path & "\AreaCode.mdb"
    datAreaCode2.DatabaseName = App.Path & "\AreaCode.mdb"
End Sub

Private Sub txtAreaCode_Change()
    Dim strBookmark As String
    Dim strCriteria As String
    'Wait for user to enter all three digits
    If Len(txtAreaCode.Text) = 3 Then
        'Record current record
        strBookmark = datAreaCode2.Recordset.Bookmark
        'Search for first matching area code
        strCriteria = "AreaCode = " + txtAreaCode.Text
        datAreaCode2.Recordset.FindFirst strCriteria
        'Handle unmatched area code
        If datAreaCode2.Recordset.NoMatch Then
            Beep
            datAreaCode2.Recordset.Bookmark = strBookmark
        End If
    End If
End Sub

Private Sub txtAreaCode_KeyPress(KeyAscii As Integer)
    If Len(txtAreaCode.Text) = 3 Then

```

```

        txtAreaCode.Text = ""
    End If
End Sub

Private Sub txtStateAbbrev1_Change()
    Dim strBookmark
    Dim strCriteria
    'Wait for user to enter two-letter abbreviation
    If Len(txtStateAbbrev1.Text) = 2 Then
        'Record current record
        strBookmark = datAreaCode1.Recordset.Bookmark
        'Search for first matching state
        strCriteria = "State = '" + txtStateAbbrev1.Text + "'"
        datAreaCode1.Recordset.FindFirst strCriteria
        'Handle unmatched state abbreviation
        If datAreaCode1.Recordset.NoMatch Then
            Beep
            datAreaCode1.Recordset.Bookmark = strBookmark
        End If
    End If
End Sub

Private Sub txtStateAbbrev1_KeyPress(KeyAscii As Integer)
    KeyAscii = Asc(UCase$(Chr$(KeyAscii)))
    If Len(txtStateAbbrev1.Text) = 2 Then
        txtStateAbbrev1.Text = ""
    End If
End Sub

Private Sub mnuAbout_Click()
    'Set properties
    About.Application = "AreaCode"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

```

This code relies on a query within the database to sort the AreaCode records by state, but there is a simple alternative to creating the query. We can set the Data control's RecordSource property to a structured query language (SQL) string, which provides a flexible and powerful tool for manipulating data. For example, you can add the following line of code to the form's Load event procedure to produce the same result as setting the RecordSource property to the ByState query:

```
datAreaCode1.RecordSource = "Select * from AREACODE order by State"
```

If you add this line to the program listing, you can delete the ByState query from the database. Both Data controls now access the same database table: one using the default area code order of the records, and the other using the SQL command to reorder the records by state.

The DataDump Application

With Visual Basic, it's easy to manipulate database files programmatically. The data access object (DAO) model provides a structured hierarchy of objects that lets you access and modify any part of a database. (See Chapter 23, "[Database Access](#)," for more about the DAO model.) The DataDump application described here is a streamlined utility that demonstrates how you can loop through the various parts of this hierarchy to determine the layout of any field in any table of any database.

Unlike the AreaCode application, this program contains no Data controls or other data-bound controls. DataDump contains only a TextBox control for displaying the analyzed internal structure of a user-selected database and a CommonDialog control for selecting a database file to analyze. All analysis of the database file contents is performed by looping through various objects within the DAO model.

This application provides a list of all tables within the database and all fields within each table. Each field is described by name, type, and data size in bytes. I've found this utility useful for quickly reviewing the contents of a database's tables without having to run Access. If you want, you can easily extend the current code to include a listing of any queries stored in the database in the analysis. Visual Basic Books Online provides a good explanation of the complete DAO hierarchy for those interested in digging into the subject even deeper.

At load time, the DataDump form prompts for a database filename by using the CommonDialog control, as shown in Figure 32-3. It then proceeds to analyze the selected database, concatenating all the results into a single string variable. The database structure is traversed just once per run of the program, and the results string is displayed in the TextBox control whenever the form is resized.

To improve the readability of the output, I have formatted the string using `vbCrLf` constants and extra spaces for padding. The font of the text box is set to Courier New, which is a monospace font that provides consistent alignment of the indented text lines. The TextBox control has scrollbars that let you view the data list no matter how long it gets, and the control adjusts automatically to fill the form whenever the form is resized.

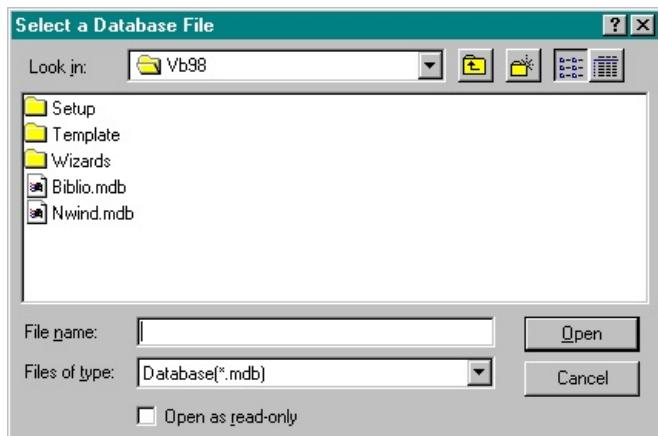


Figure 32-3. The CommonDialog control lets the user select a database file for analysis.

Figure 32-4 shows the structure of the BIBLIO.MDB database that ships with Visual Basic.

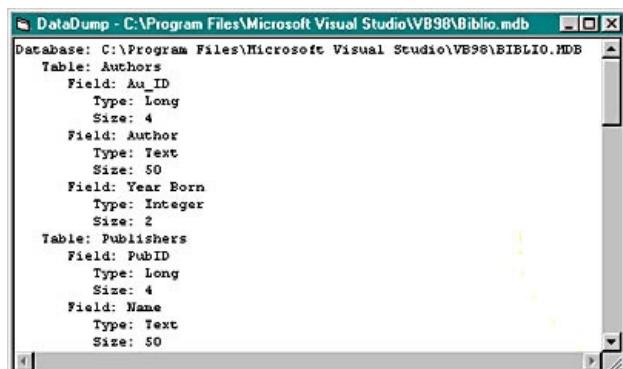


Figure 32-4. The DataDump application in action, showing the tables and fields in BIBLIO.MDB.

DATADUMP.FRM

Figure 32-5 below shows the DataDump form during development. The TextBox control is smaller than the form but is resized at runtime to the size of the form.

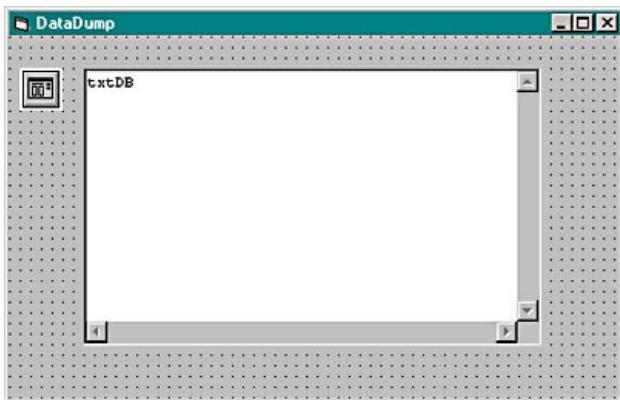


Figure 32-5. The DataDump form during development.

To create this form, use the following table and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

DATADUMP.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmDataDump</i>
Caption	<i>DataDump</i>
CommonDialog	
Name	<i>cdlOne</i>
TextBox	
Name	<i>txtDB</i>
Font	<i>Courier New</i>
MultiLine	<i>True</i>
ScrollBars	<i>3 - Both</i>

Source Code for DATADUMP.FRM

```

Option Explicit

Dim strDBFileName As String
Dim strDB As String

Private Sub Form_Load()
    'Center this form
    Me.Left = (Screen.Width - Width) \ 2
    Me.Top = (Screen.Height - Height) \ 2
    'Prompt user for database filename
    cdlOne.DialogTitle = "Select a Database File"
    cdlOne.Filter = "Database(*.mdb)|*.mdb"
    cdlOne.CancelError = True
    'Check for Cancel button click
    On Error Resume Next
    cdlOne.ShowOpen
    If Err = cdlCancel Then End
    'Prepare to analyze database

```

```

strDBFileName = cdlOne.FileName
Me.Caption = "DataDump - " & strDBFileName
GetStructure
End Sub

Private Sub Form_Resize()
  `Size text box to fit form
  txtDB.Move 0, 0, ScaleWidth, ScaleHeight
  `Display analysis string in text box
  txtDB.Text = strDB
End Sub

Private Sub GetStructure()
  `Looping variables
  Dim intI As Integer
  Dim intJ As Integer
  `Database objects
  Dim dbDump As Database
  Dim recDump As Recordset
  Dim tbdDump As TableDef
  Dim fldDump As Field
  `Open the database
  Set dbDump = Workspaces(0).OpenDatabase(strDBFileName)
  strDB = "Database: " & dbDump.Name & vbCrLf
  `Process each table
  For intI = 0 To dbDump.TableDefs.Count - 1
    Set tbdDump = dbDump.TableDefs(intI)
    If Left(tbdDump.Name, 4) <> "MSys" Then
      `Get table's name
      strDB = strDB & Space(3) & "Table: "
      strDB = strDB & tbdDump.Name & vbCrLf
      `Process each field in each table
      For intJ = 0 To tbdDump.Fields.Count - 1
        Set fldDump = tbdDump.Fields(intJ)
        `Get field's name
        strDB = strDB & Space(6) & "Field: "
        strDB = strDB & fldDump.Name & vbCrLf
        `Get field's data type
        strDB = strDB & Space(9) & "Type: "
        Select Case fldDump.Type
        Case dbBoolean
          strDB = strDB & "Boolean" & vbCrLf
        Case dbByte
          strDB = strDB & "Byte" & vbCrLf
        Case dbInteger
          strDB = strDB & "Integer" & vbCrLf
        Case dbLong
          strDB = strDB & "Long" & vbCrLf
        Case dbCurrency
          strDB = strDB & "Currency" & vbCrLf
        Case dbSingle
          strDB = strDB & "Single" & vbCrLf
        Case dbDouble
          strDB = strDB & "Double" & vbCrLf
        Case dbDate
          strDB = strDB & "Date" & vbCrLf
        Case dbText
          strDB = strDB & "Text" & vbCrLf
        Case dbLongBinary
          strDB = strDB & "LongBinary" & vbCrLf
        Case dbMemo
          strDB = strDB & "Memo" & vbCrLf
        Case Else
          strDB = strDB & "(unknown)" & vbCrLf"
        End Select
      Next intJ
    Next intI
  End Sub

```

```
End Select
`Get field's size in bytes
If fldDump.Type <> dbLongBinary And _
fldDump.Type <> dbMemo Then
    strDB = strDB & Space(9) & "Size: "
    strDB = strDB & fldDump.Size & vbCrLf
End If
Next intJ
End If
Next intI
`Close database
dbDump.Close
End Sub
```

Testing the DataDump Application

To test the DataDump application, you'll need to set a reference to the Microsoft DAO 3.51 Object Library in the References dialog box.

The Jot Application

The Jot application uses a Multiple Document Interface (MDI) form to display multiple note windows. To make this a commercial application, you'd probably want to add a lot of features and modify the design, but the application is fully functional as is and it demonstrates a number of useful Visual Basic programming features.

I've included features that demonstrate how to create multiple copies of MDI child forms, how to use hot keys on an MDI form, how to create a database from scratch programmatically, and one way to center an image on an MDI form.

Figure 32-6 shows the Jot application at runtime. Buttons on the toolbar let you create new note windows and delete them when you have finished. Other buttons arrange the note windows and their minimized windows. When you close Jot, all notes are saved in a database; when you run Jot the next time, these notes reappear.

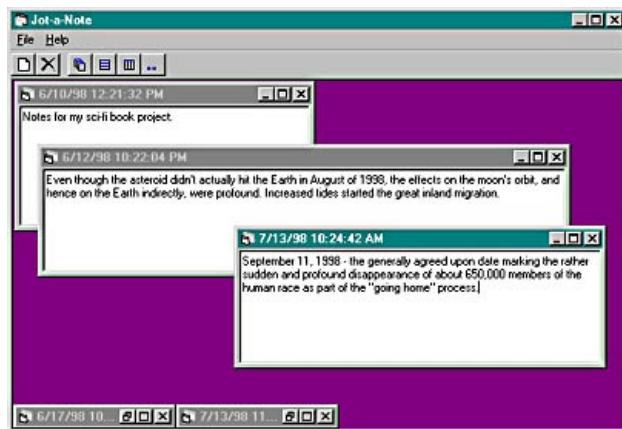


Figure 32-6. The Jot application in action.

Multiple Child Forms

The first button on the toolbar creates a new, blank note window: I created a single Note child form named *frmNote* at design time and placed a call in the toolbar button's *ButtonClick* event procedure to a procedure named *MakeNewNote*, which creates a new copy of the Note form. If you look in the *MakeNewNote* procedure, you'll see two lines of code that create and display each new copy of the Note form:

```
Dim newNote As New frmNote
newNote.Show
```

Each copy of this child form shares the same event procedures. To refer to the properties of the current copy of the child form (the Note form with the focus) in these procedures, I've used the *Me* prefix. For example, during the child form's *Unload* event procedure, its *Caption* property is saved in a field of the database. To record the caption of the child form currently unloading, the *Unload* event procedure refers to *Me.Caption*.

MDI Form Hot Key

The Note form's *KeyPreview* property is set to *True* so that all keypresses are intercepted before they are processed further by the form or its controls. The only keypress the Note form intercepts is *Ctrl-T*, which updates the date and time displayed in the current Note form's caption to the current date and time. Normally the date and time of each note are fixed at the time the note is created. This hot key combination lets you update the caption when you want. Even if you assign your own hot key combinations, you can still use the standard Windows shortcut key combinations *Ctrl-X*, *Ctrl-C*, and *Ctrl-V* to cut, copy, and paste as you edit these notes. The *Ctrl-T* command works much like these built-in commands. You can easily add code to process any other keys or combinations of keys in the same *KeyDown* event procedure.

Figures 32-7 and 32-8 show a Note form just before and just after *Ctrl-T* is pressed to update the date

and time in the caption.

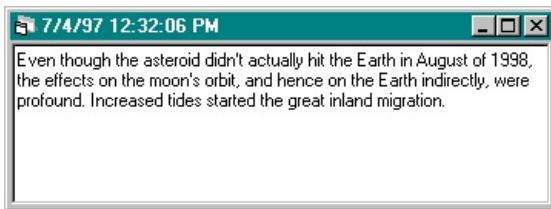


Figure 32-7. A note originally created on July 4, 1997.

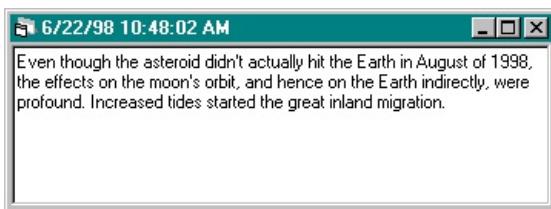


Figure 32-8. The same note updated to the current date and time.

Creating a Database

When the Jot form loads, it attempts to open the JOT.MDB database file to read any previously saved notes. If this database file doesn't exist, which is true the first time you run the Jot application, the Jot form ignores the request. The database is not created until the first Note child form is unloaded. When a Note child form unloads and there is no database, a special block of code creates the database automatically. Understanding how these lines of code work provides concepts you'll find useful for mastering a lot of other database programming techniques. Here is the code that creates the new database (if the database doesn't already exist) in the Note child form's Unload event procedure:

```
'Create empty database
Set dbJot = Workspaces(0).CreateDatabase( _
    "Jot.mdb", dbLangGeneral)
`Create table
Set tbdJot = dbJot.CreateTableDef("JotTable")
`Create field
Set fldJot = tbdJot.CreateField("JotDateTime", dbDate)
`Add new field to table's Fields collection
tbdJot.Fields.Append fldJot
`Create second field
Set fldJot = tbdJot.CreateField("JotNote", dbMemo)
`Add it to table
tbdJot.Fields.Append fldJot
`Add new table to TableDefs collection
dbJot.TableDefs.Append tbdJot
`New database is now open
```

The CreateDatabase, CreateTableDef, and CreateField methods, as their names imply, create a Database object, a TableDef object, and a Field object, respectively. However, these methods don't actually hook the created objects into their parent objects—they just allocate memory and create all the behind-the-scenes details of the database objects. It takes an explicit second step to connect each new object to the structure of the parent object that created it. The code above shows several Append methods used to hook the new objects into the structure of the containing collection. For example, the TableDef object contains a collection named Fields, which contains all Field objects in the defined table. The Append method of the Fields collection is called to hook the freshly created Field object into this collection. It's a two-step process: create an object, and then append it to a collection to tie everything together. As shown above, this same two-step process is used to create fields and tables, append fields to tables, and append tables to a database.

Centering an Image on an MDI Form

The types of controls that can be placed on an MDI form are limited. In general, only controls with Align

properties can be placed on an MDI form, and these controls will always be aligned at the top, bottom, left, or right of the form. An MDI form does have a Picture property, but any image loaded into this property is also aligned at the top-left corner of the form. So how did the Jot application end up with a bitmap image that always moves to the center of the MDI form as the form is resized? It's a useful trick to know.

You can move a child form to any relative position within the MDI parent form, including dead center, with a little bit of calculation. I've taken advantage of this by centering the Splash form in the Jot form's Resize event procedure.

The Splash form doesn't display anything except a picture because its BorderStyle property has been set to 0 - *None* and the form is sized to match the size of the Image control it contains. As a result, when the Splash form is placed in the center of the MDI Jot form, the user sees only a centered bitmap picture. Figures 32-9 and 32-10 show the Jot form with the centered image before and after the form is resized.

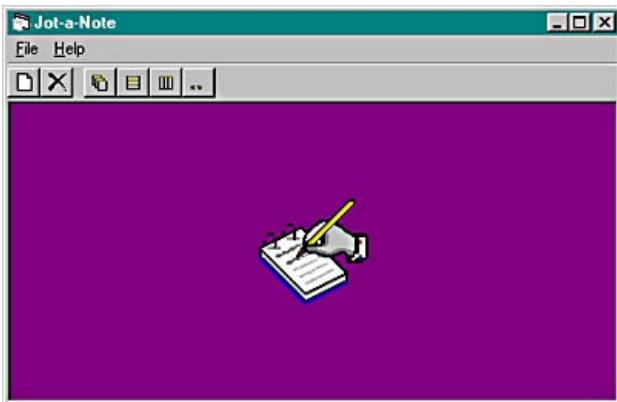


Figure 32-9. The Jot form with a centered image.

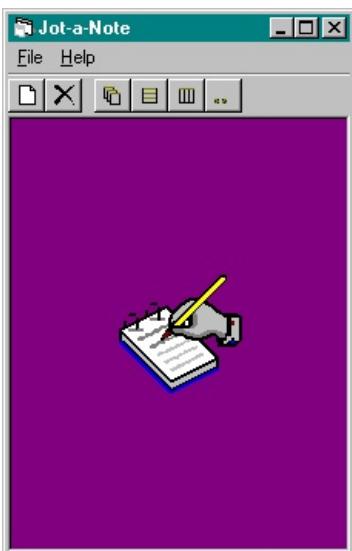


Figure 32-10. The image still centered after the Jot form is resized.

JOT.FRM

As shown in the project list in Figure 32-11, the Jot application contains four files. JOT.FRM is the main MDI startup form, so we'll take a look at it first.

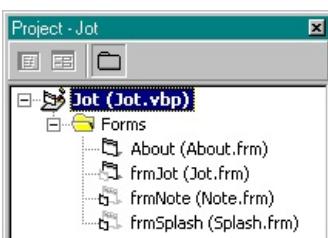
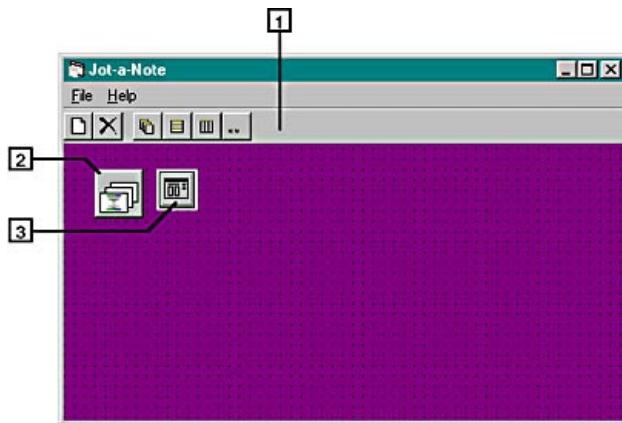


Figure 32-11. The Jot application's project list.

Figure 32-12 below shows the Jot form during development.

**Figure 32-12.** The MDI Jot form during development.

To create this form, add an MDI form to a new Standard EXE project and use the following tables to add the appropriate controls and set any nondefault properties.

JOT.FRM Menu Design Window Entries

Caption	Name	Indentation	Enabled
&File	mnuFile	0	True
&New	mnuNew	1	False
&OpenDear John, How Do I...	mnuOpen	1	False
&Save	mnuSave	1	False
Save &AsDear John, How Do I...	mnuSaveAs	1	False
-	mnuFileDash1	1	True
E&xit	mnuExit	1	True
&Help	mnuHelp	0	True
&Contents	mnuContents	1	True
&Search For Help OnDear John, How Do I...	mnuSearch	1	True
-	mnuHelpDash1	1	True
&AboutDear John, How Do I...	mnuAbout	1	True

JOT.FRM (MDI) Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	frmJot
	Caption	Jot-a-Note
Toolbar		
1	Name	tlbToolBar
ImageList		
2	Name	imlIcons

CommonDialog

3	Name	cdlOne
---	------	--------

* The number in the ID No. column corresponds to the number in Figure 32-12 that identifies the location of the object on the form.

Toolbars, Buttons, and ToolTips

In earlier versions of Visual Basic, you could create toolbars with buttons and ToolTips by using a variety of tricks and techniques or third-party custom controls, but Visual Basic now makes this easier than ever to accomplish. Here I'll describe how to create the toolbar with buttons that appears on the Jot form, shown in Figure 32-12 above.

First load the button images into the ImageList control. Right-click the ImageList control and select Properties to edit this control's properties. Change the image size option to 16 x 16. Click the Images tab, and click the Insert Picture button to add each button's image.

The following table lists the images to be inserted into the ImageList control.

Image Index	Image
1	NEW.BMP
2	DELETE.BMP
3	CASCADE.BMP
4	TILEH.BMP
5	TILEV.BMP
6	ARNGICON.BMP

I found the first two images in this table, NEW.BMP and DELETE.BMP, in the \Common\Graphics\Bitmaps\Tlbr_w95 folder of my Visual Studio CD-ROM. The CASCADE.BMP, TILEH.BMP, TILEV.BMP, and ARNGICON.BMP images were created using the Microsoft Paint utility and can be found on the companion CD-ROM.

Figure 32-13 below shows the ImageList control's Property Pages dialog box after all the button images have been inserted.

Once the ImageList control has its button images loaded, you can connect this control to the Toolbar control. Right-click the toolbar and select Properties. Set the ImageList entry to *imlIcons* to make the connection. To explicitly create buttons and assign images from the list to them, click the Buttons tab. Figure 32-14 shows the Toolbar control's Property Pages dialog box.

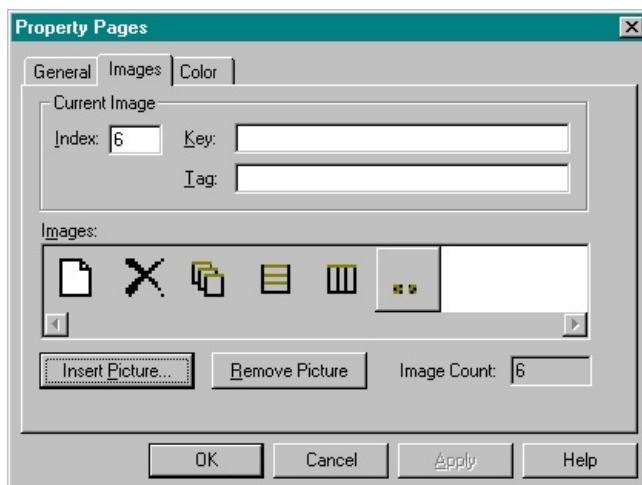


Figure 32-13. Button images inserted into the ImageList control.

The toolbar buttons are added one at a time by clicking the Insert Button button. For each button, fill in the fields with the appropriate data. The text you type in the ToolTipText field will appear as the button's ToolTip. Enter a descriptive label in the Key field—this text will be used in the program to determine which button the user clicks, and an appropriate entry here will make your program code easier to follow and more self-documenting. In the Image field, enter the number of the image to be associated from the ImageList control—this is the entry that makes the connection between each button and a specific image.

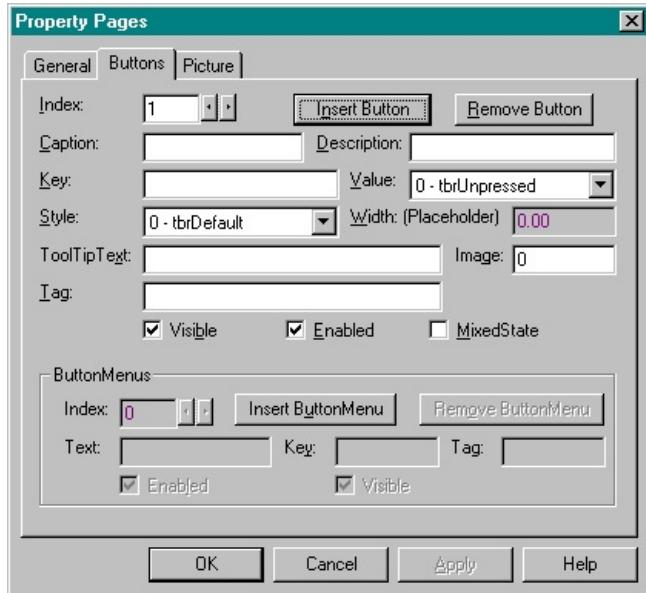


Figure 32-14. The Toolbar control's Property Pages dialog box.

The Style property determines the type of button; this property is normally left at the default setting of *0 - tbrDefault*. To add a space between buttons on the toolbar, simply add a button and set its Style property to *3 - tbrSeparator*. You won't need to assign an image or a key to these separator buttons. Other useful properties are available to control the appearance and behavior of your buttons, but we won't need them in this example application. (You should take the time to become familiar with the full set of properties later.)

The following table shows the button settings for the *tlbToolBar* control.

Button Index	Key	Style	ToolTipText	Image Index
1	New	<i>0 - tbrDefault</i>	New	1
2	Delete	<i>0 - tbrDefault</i>	Delete	2
3		<i>3 - tbrSeparator</i>		0
4	Cascade	<i>0 - tbrDefault</i>	Cascade	3
5	<i>TileHorizontally</i>	<i>0 - tbrDefault</i>	<i>Tile Horizontally</i>	4
6	<i>TileVertically</i>	<i>0 - tbrDefault</i>	<i>Tile Vertically</i>	5
7	<i>ArrangeIcons</i>	<i>0 - tbrDefault</i>	<i>Arrange Icons</i>	6

That's all there is to setting up a toolbar with buttons and ToolTips! If you've worked with Visual Basic ToolTip do-it-yourself techniques in the past, you'll quickly appreciate the ease with which ToolTips can now be implemented.

Enter the following source code for the Jot form, which includes code to make your toolbar buttons active.

Source Code for JOT.FRM

```
Option Explicit
Public blnSave As Boolean
```

```

Private Sub MDIForm_Load()
    'Center this form on screen
    Me.Move (Screen.Width - Width) \ 2, _
        (Screen.Height - Height) \ 2
    'Load all previous notes from database
    LoadNotes
    'Display splash image
    frmSplash.Show
End Sub

Private Sub MDIForm_QueryUnload(Cancel As Integer, _
UnloadMode As Integer)
    'Prepare to save all current notes
    blnSave = True
End Sub

Private Sub MDIForm_Resize()
    'Move splash image to center of MDI form
    frmSplash.Move (Me.ScaleWidth - frmSplash.Width) \ 2, _
        (Me.ScaleHeight - frmSplash.Height) \ 2
End Sub

Private Sub mnuAbout_Click()
    'Set properties and display About form
    About.Application = "Jot"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

Private Sub LoadNotes()
    Dim dbJot As Database
    Dim recJot As Recordset
    'Open database of previous notes
    On Error Resume Next
    Set dbJot = Workspaces(0).OpenDatabase("Jot.mdb")
    'If database doesn't exist, nothing to display
    If Err Then Exit Sub
    On Error GoTo 0
    'Open table of notes
    Set recJot = dbJot.OpenRecordset("JotTable")
    'RecordCount will be 1 if there are any records
    If recJot.RecordCount Then
        'Create note window for each note
        Do Until recJot.EOF
            MakeNewNote Format(recJot!JotDateTime, _

```

```

        "General Date"), recJot!JotNote
        recJot.MoveNext
    Loop
End If
`Empty database table for now
dbJot.Execute "Delete * from JotTable"
`Close recordset and database
recJot.Close
dbJot.Close
End Sub

Private Sub MakeNewNote(sdtDateTime As String, strNote As String)
    `Create new copy of note form
    Dim newNote As New frmNote
    `Set caption and note contents
    newNote.rtfNote.Text = strNote
    newNote.Caption = sdtDateTime
    `Display new note form
    newNote.Show
End Sub

Private Sub tlbToolBar_ButtonClick(ByVal Button As MSComctlLib.Button)
    Select Case Button.Key
    Case "New"
        MakeNewNote Format(Now, "General Date"), ""
    Case "Delete"
        Unload ActiveForm
    Case "Cascade"
        frmJot.Arrange vbCascade
    Case "TileVertically"
        frmJot.Arrange vbTileVertical
    Case "TileHorizontally"
        frmJot.Arrange vbTileHorizontal
    Case "ArrangeIcons"
        frmJot.Arrange vbArrangeIcons
    Case Else
        End Select
    End Sub

```

SPLASH.FRM

This borderless form displays the Jot logo in the center of the main Jot form, as explained previously. The form contains one Image control, which you can load at design time using the JOT.BMP bitmap image file (found on the companion CD-ROM). Figure 32-15 shows the Splash form during development. Notice that the form's border is not visible because the BorderStyle property is set to 0 - None.

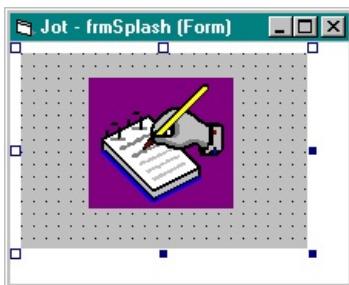


Figure 32-15. The Splash form during development.

To create this form, position an Image control on a blank form and set the nondefault settings for the form and control as shown below.

SPLASH.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmSplash</i>
Caption	<i>frmSplash</i>
BorderStyle	<i>0 - None</i>
MDIChild	<i>True</i>
ShowInTaskBar	<i>False</i>
Image	
Name	<i>imgSplash</i>
Picture	<i>JOT.BMP</i>

The code for this form is contained in the Form_Load event procedure. When the form loads, the Image control is moved to the upper-left corner of the form and the form is sized to the dimensions of the Image control.

Source Code for SPLASH.FRM

```
Option Explicit

Private Sub Form_Load()
    'Move image to upper left corner
    imgSplash.Move 0, 0
    'Size this form to size of splash image
    Me.Width = imgSplash.Width
    Me.Height = imgSplash.Height
End Sub
```

NOTE.FRM

The Note form is created once during development and duplicated as needed at runtime. Each copy of this form holds a single user-edited note. I chose to use a RichTextBox control for the actual note-editing control on the Note form, but a TextBox control would work well too. If you want to enhance the Jot application, the RichTextBox control offers more formatting capabilities.

Figure 32-16 below shows the Note form during development with a single RichTextBox control.

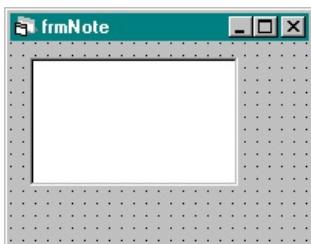


Figure 32-16. The Note form during development.

To create this form, position a RichTextBox control on a blank form, set the nondefault settings for the form and control, and enter the source code lines as shown.

NOTE.FRM Objects and Property Settings

Property	Value
Form	
Name	<i>frmNote</i>

Caption	<i>frmNote</i>
KeyPreview	<i>True</i>
MDIChild	<i>True</i>
RichTextBox	
Name	<i>rtfNote</i>
ScrollBars	<i>3 - rtfBoth</i>

Source Code for NOTE.FRM

```

Option Explicit

Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)
    'Update date and time in caption with hot key Ctrl-T
    If KeyCode = 84 And Shift = 2 Then
        Me.Caption = Format(Now, "General Date")
    End If
End Sub

Private Sub Form_Resize()
    rtfNote.Move 0, 0, ScaleWidth, ScaleHeight
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Dim dbJot As Database
    Dim recJot As Recordset
    Dim tbdJot As TableDef
    Dim fldJot As Field
    Dim vntErrorNumber As Variant
    Dim dtmDateTime As Date
    Dim strNote As String
    'blnSave means save this note
    If frmJot.blnSave = False Then Exit Sub
    'Open database to save this note
    On Error Resume Next
    Set dbJot = Workspaces(0).OpenDatabase("Jot.mdb")
    vntErrorNumber = Err
    On Error GoTo 0
    'Create database if it does not already exist
    If vntErrorNumber Then
        'Create empty database
        Set dbJot = Workspaces(0).CreateDatabase( _
            "Jot.mdb", dbLangGeneral)
        'Create table
        Set tbdJot = dbJot.CreateTableDef("JotTable")
        'Create field
        Set fldJot = tbdJot.CreateField("JotDateTime", dbDate)
        'Add new field to table's Fields collection
        tbdJot.Fields.Append fldJot
        'Create second field
        Set fldJot = tbdJot.CreateField("JotNote", dbMemo)
        'Add it to table
        tbdJot.Fields.Append fldJot
        'Add new table to TableDefs collection
        dbJot.TableDefs.Append tbdJot
        'New database is now open
    End If
    'Get working recordset
    Set recJot = dbJot.OpenRecordset("JotTable", dbOpenTable)
    'Add new record

```

```
recJot.AddNew
`Prepare data for placing in record
dtmDateTime = Me.Caption
strNote = Me.rtfNote.Text
If strNote = "" Then strNote = " "
`Load fields in new record
recJot!JotDateTime = dtmDateTime
recJot!JotNote = strNote
`Be sure database is updated
recJot.Update
`Close recordset and database
recJot.Close
dbJot.Close
End Sub
```

Testing the Jot Application

To test the Jot application, you'll need to set a reference to the Microsoft DAO 3.51 Object Library in the References dialog box. Also, in the Project Properties dialog box, set the Startup Object property to *frmJot*. Although Jot is a simple program, it provides a solid foundation for creating your own MDI applications.

Chapter Thirty-Three

Utilities

It can be extremely easy to create a useful little utility in Visual Basic. Sometimes such utilities are tools that can help you during the development of other Visual Basic applications, and sometimes they're just good ways to learn more about Visual Basic programming in general. This chapter presents three utilities: one that lets you experiment with the mouse pointer, one that lets you quickly view or listen to multimedia files anywhere on your system, and even one that tells you today's windchill index.

The MousePtr Application

The MousePtr application is a handy utility you can use to quickly review any of the 16 standard mouse pointers and to load and view any icon file as a mouse pointer.

Figure 33-1 shows the MousePtr form at runtime. When you select a mouse pointer option, the mouse pointer changes to reflect your choice. To use an icon as a mouse pointer, select the last setting, 99 - *Custom Icon*. The first time you select this option, the Select An Icon File dialog box appears. This dialog box allows you to choose the icon to be displayed. Subsequent selection of the 99 - *Custom Icon* setting displays the same custom icon. To change the icon that is displayed, click the Select Icon button and choose a different icon in the Select An Icon File dialog box.

I've provided a TextBox control as a learning tool and a memory aid. Selecting the 0 - *Default* or 1 - *Arrow* setting appears to display the same standard arrow. You'll see the difference in the way these mouse pointers behave when you move them across the face of the TextBox control. The default mouse pointer changes to an I-beam when it's located over editable text in a text box, as shown in Figure 33-1, whereas the arrow mouse pointer remains an arrow no matter which controls it passes over.

Figure 33-2 shows the EARTH.ICO icon as a mouse pointer. You'll find this icon and many others installed in your Visual Basic folders, or on the Visual Studio CD-ROM.

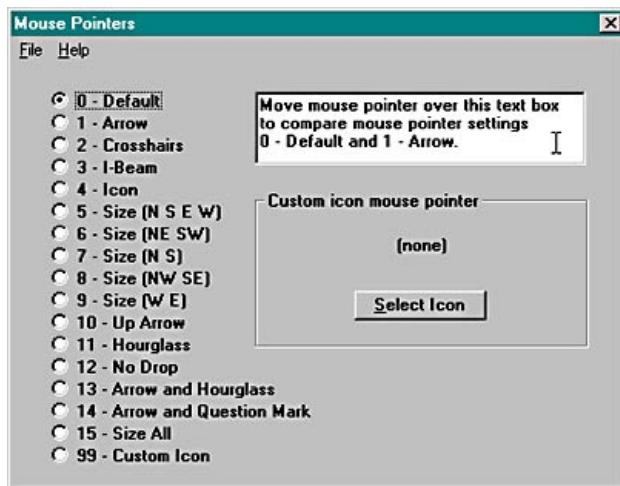


Figure 33-1. The MousePtr form in action.

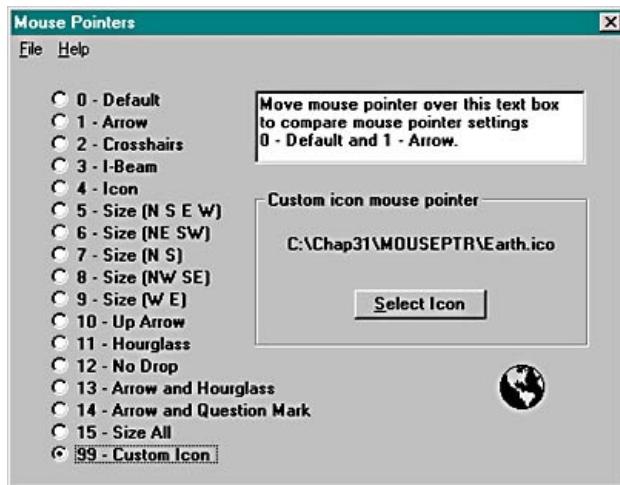


Figure 33-2. Using the EARTH.ICO icon as an interesting mouse pointer.

If you plan to use several icon files as custom mouse pointers in an application, you might want to store them in a resource file, as described in Chapter 26, "[Project Development](#)."

As shown in the project list in Figure 33-3, this project contains two files. The About form displays the standard About dialog box for this application.

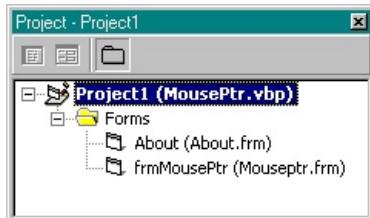


Figure 33-3. The MousePtr project list.

MOUSEPTR.FRM

The MousePtr form displays an array of option buttons you can click to select a mouse pointer. The Index property of each option button corresponds to the value assigned to the form's MousePointer property, which displays the type of mouse pointer. These numbers range from 0 through 15, with a big jump to 99 for the special case in which an icon file is loaded as a user-defined (custom) mouse pointer. If you build this form manually, be sure to set the last option button's Index property to 99.

Figure 33-4 shows the MousePtr form during development.

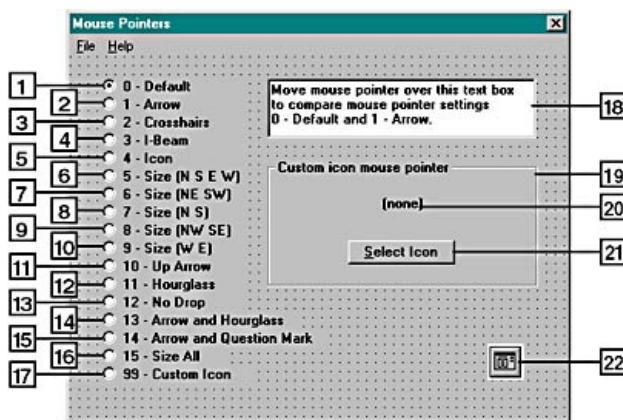


Figure 33-4. The MousePtr form during development.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

MOUSEPTR.FRM Menu Design Window Entries

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&Open	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &As	mnuSaveAs	1	<i>False</i>
-	mnuFileDash1	1	<i>True</i>
E&xit	mnuExit	1	<i>True</i>
&Help	mnuHelp	0	<i>True</i>
&Contents	mnuContents	1	<i>True</i>
&Search For Help	mnuSearch	1	<i>True</i>
-	mnuHelpDash1	1	<i>True</i>
&About	mnuAbout	1	<i>True</i>

MOUSEPTR.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmMousePtr</i>
	Caption	<i>Mouse Pointers</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
OptionButton		
1	Name	<i>optMousePtr</i>
	Caption	<i>0 - Default</i>
	Index	<i>0</i>
2	Name	<i>optMousePtr</i>
	Caption	<i>1 - Arrow</i>
	Index	<i>1</i>
3	Name	<i>optMousePtr</i>
	Caption	<i>2 - Crosshairs</i>
	Index	<i>2</i>
4	Name	<i>optMousePtr</i>
	Caption	<i>3 - I-Beam</i>
	Index	<i>3</i>
5	Name	<i>optMousePtr</i>
	Caption	<i>4 - Icon</i>
	Index	<i>4</i>
6	Name	<i>optMousePtr</i>
	Caption	<i>5 - Size (N S E W)</i>
	Index	<i>5</i>
7	Name	<i>optMousePtr</i>
	Caption	<i>6 - Size (NE SW)</i>
	Index	<i>6</i>
8	Name	<i>optMousePtr</i>
	Caption	<i>7 - Size (N S)</i>
	Index	<i>7</i>
9	Name	<i>optMousePtr</i>
	Caption	<i>8 - Size (NW SE)</i>
	Index	<i>8</i>
10	Name	<i>optMousePtr</i>
	Caption	<i>9 - Size (WE)</i>

	Index	9
11	Name	<i>optMousePtr</i>
	Caption	<i>10 - Up Arrow</i>
	Index	10
12	Name	<i>optMousePtr</i>
	Caption	<i>11 - Hourglass</i>
	Index	11
13	Name	<i>optMousePtr</i>
	Caption	<i>12 - No Drop</i>
	Index	12
14	Name	<i>optMousePtr</i>
	Caption	<i>13 - Arrow and Hourglass</i>
	Index	13
15	Name	<i>optMousePtr</i>
	Caption	<i>14 - Arrow and Question Mark</i>
	Index	14
16	Name	<i>optMousePtr</i>
	Caption	<i>15 - Size All</i>
	Index	15
17	Name	<i>optMousePtr</i>
	Caption	<i>99 - Custom Icon</i>
	Index	99

TextBox

18	Name	<i>txtTestArea</i>
	Text	<i>Move mouse pointer over this text box to compare mouse pointer settings 0 - Default and 1 - Arrow.</i>
	MultiLine	<i>True</i>

Frame

19	Name	<i>fraSelect</i>
	Caption	<i>Custom icon mouse pointer</i>

Label

20	Name	<i>lblIcon</i>
	Caption	<i>(none)</i>

CommandButton

21	Name	<i>cmdSelect</i>
	Caption	<i>&Select Icon</i>

CommonDialog

22	Name	<i>dlgOne</i>
----	------	---------------

* The number in the ID No. column corresponds to the number in Figure 33-4 that identifies the location of the object on the form.

Source Code for MOUSEPTR.FRM

```

Option Explicit

Private Sub cmdSelect_Click()
    'Prompt user for icon filename
    dlgOne.DialogTitle = "Select an Icon File"
    dlgOne.Filter = "Icon(*.ico) | *.ico"
    dlgOne.CancelError = True
    'Check for Cancel button click
    On Error Resume Next
    dlgOne.ShowOpen
    If Err <> cdlCancel Then
        'Load icon as mouse pointer
        frmMousePtr.MouseIcon = LoadPicture(dlgOne.FileName)
        'Display current icon path
        lblIcon.Caption = dlgOne.FileName
        'Select mouse pointer type 99
        optMousePtr(99).SetFocus
    End If
End Sub

Private Sub Form_Load()
    'Center this form
    Left = (Screen.Width - Width) \ 2
    Top = (Screen.Height - Height) \ 2
    'Move focus off text box and onto option buttons
    Show
    optMousePtr(0).SetFocus
End Sub

Private Sub optMousePtr_Click(Index As Integer)
    'Set selected mouse pointer
    frmMousePtr.MousePointer = Index
    'Open dialog box if no icon file previously specified
    If Index = 99 And lblIcon = "(none)" Then
        cmdSelect_Click
    End If
End Sub

Private Sub mnuAbout_Click()
    'Set properties
    About.Application = "MousePtr"
    About.Heading = "Microsoft Visual Basic 6.0 " & _
        "Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    dlgOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    dlgOne.HelpCommand = cdlHelpContents
    dlgOne.ShowHelp
End Sub

```

```
Private Sub mnuSearch_Click()
    dlgOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    dlgOne.HelpCommand = cdlHelpPartialKey
    dlgOne.ShowHelp
End Sub
```

The ShowTell Application

The ShowTell application is simple in design, yet it provides a handy utility for quickly reviewing many types of image files (bitmap, icon, JPG, GIF, and Windows metafile), listening to sound files (WAV), or viewing video clips (AVI and MPG).

The startup form, *frmShowTell*, provides a button for each type of image or multimedia file. The various types of files fall into two broad categories: those that provide a static picture or image that can be loaded into an Image control and those that are more dynamic and require activation using the mciExecute multimedia API function. I've set up the application to handle the differences in the types of files automatically.

Figure 33-5 shows the startup form at runtime.

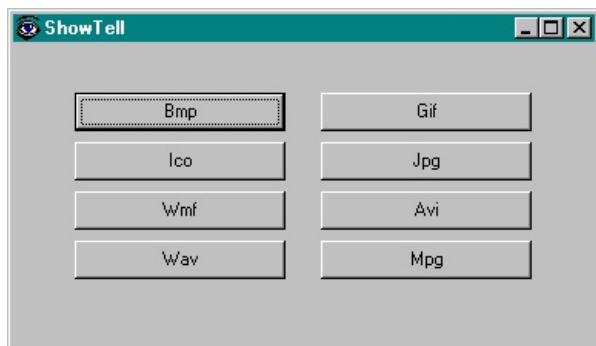


Figure 33-5. The *ShowTell* application startup form.

When one of the buttons is clicked, a CommonDialog control is used to display an open dialog box that lets you choose a file of the indicated type located anywhere on your system. Static images are displayed on a second form, *frmImage*, and dynamic multimedia files are played using an API function call.

Immediately after the user closes the displayed image, or when a sound or video clip automatically finishes playing, the open dialog box reappears, allowing the user to select another file of the same type. To close the dialog box and return to the main form, click the Cancel button instead of selecting a file.

In Figure 33-6, a sample JPG file is displayed, and in Figure 33-7 on the following page, a sample AVI file is displayed.



Figure 33-6. The *ShowTell* application displaying a JPG file.

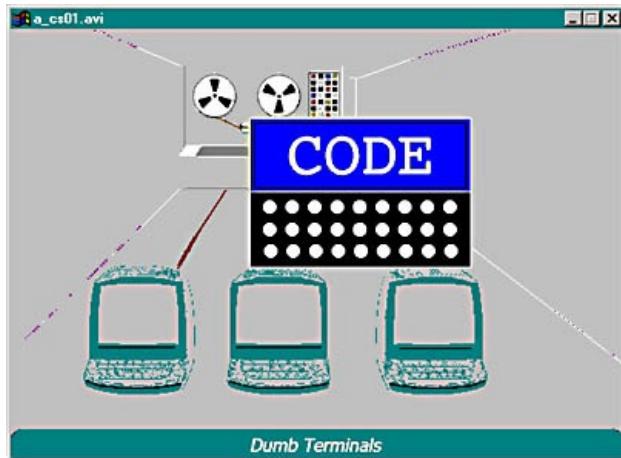


Figure 33-7. The ShowTell application displaying a video clip (AVI).

As shown in Figure 33-8, the ShowTell project consists of two forms.



Figure 33-8. The ShowTell project list.

SHOWTELL.FRM

As mentioned, the ShowTell form uses a CommonDialog control to display an open dialog box that allows the user to select a file to be displayed or played. While the open dialog box is visible and the selected image or multimedia file is displayed, the ShowTell form is hidden. When static images are displayed, the ShowTell form stays hidden until the image is closed by the user, but when multimedia files are played, the ShowTell program displays the open dialog box as soon as the multimedia file has started.

Figure 33-9 shows the ShowTell form during development. This form contains eight command buttons and one CommonDialog control.

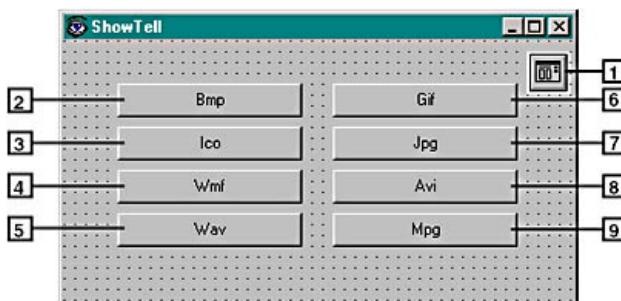


Figure 33-9. The ShowTell form during development.

To create the ShowTell form, use the table and source code on the following pages to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

SHOWTELL.FRM Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmShowTell</i>
	Caption	<i>ShowTell</i>
	Icon	<i>Eye.ico</i>

CommonDialog

1	Name	<i>dlgOne</i>
---	------	---------------

CommandButton

2	Name	<i>cmdShow</i>
	Caption	<i>Bmp</i>
	Index	<i>0</i>
3	Name	<i>cmdShow</i>
	Caption	<i>Ico</i>
	Index	<i>1</i>
4	Name	<i>cmdShow</i>
	Caption	<i>Wmf</i>
	Index	<i>2</i>
5	Name	<i>cmdShow</i>
	Caption	<i>Wav</i>
	Index	<i>3</i>
6	Name	<i>cmdShow</i>
	Caption	<i>Gif</i>
	Index	<i>4</i>
6	Name	<i>cmdShow</i>
	Caption	<i>Gif</i>
	Index	<i>4</i>

CommandButton

7	Name	<i>cmdShow</i>
	Caption	<i>Jpg</i>
	Index	<i>5</i>
8	Name	<i>cmdShow</i>
	Caption	<i>Avi</i>
	Index	<i>6</i>
9	Name	<i>cmdShow</i>
	Caption	<i>Mpg</i>
	Index	<i>7</i>

* The number in the ID No. column corresponds to the number in Figure 33-9 that identifies the location of the object on the form.

Source Code for SHOWTELL.FRM

```
Option Explicit
```

```
Private Declare Function mciExecute _
```

```

Lib "winmm.dll" (
    ByVal lpstrCommand As String _
) As Long

Private Sub Form_Load()
    'Center this form on screen
    Me.Move (Screen.Width - Width) \ 2, -
        (Screen.Height - Height) \ 2
End Sub

Private Sub cmdShow_Click(intIndex As Integer)
    Me.Hide
    Select Case intIndex
    Case 0
        ShowMedia "bmp"
    Case 1
        ShowMedia "ico"
    Case 2
        ShowMedia "wmf"
    Case 3
        PlayMedia "wav"
    Case 4
        ShowMedia "gif"
    Case 5
        ShowMedia "jpg"
    Case 6
        PlayMedia "avi"
    Case 7
        PlayMedia "mpg"
    End Select
    Me.Show
End Sub

Private Sub PlayMedia(strFileExt As String)
    'Process multimedia files
    Dim strFilter As String
    Dim strFile As String
    Dim lngX As Long
    strFilter = "*." & strFileExt & "|*." & strFileExt
    Do
        strFile = GetFileName(strFilter)
        If strFile = "" Then Exit Do
        lngX = mciExecute("Play " & strFile)
    Loop
End Sub

Private Sub ShowMedia(strFileExt As String)
    'Process static images
    Dim strFilter As String
    Dim strFile As String
    Dim lngX As Long
    strFilter = "*." & strFileExt & "|*." & strFileExt
    Do
        strFile = GetFileName(strFilter)
        If strFile = "" Then Exit Do
        frmImage.Display strFile
    Loop
End Sub

Private Function GetFileName(strFilter As String) As String
    'Use CommonDialog control to display a dialog box that allows
    'user to select media file
    With dlgOne
        .DialogTitle = "ShowTell"

```

```
.Flags = cdlOFNHideReadOnly
.Filter = strFilter
.FileName = ""
.CancelError = True
On Error Resume Next
>ShowOpen
On Error GoTo 0
.GetFileName = .FileName
End With
End Function
```

FRMIMAGE.FRM

The *frmImage* form loads and displays an image from a file. The form's code comprises just one public method, *Display*, which is responsible for handling all aspects of the display of the graphics image. *Display*'s one parameter is the path and filename of the file to be loaded and displayed. This filename is automatically placed in the form's *Caption* property for easy reference. The image is then loaded from the file, and the form is sized and moved to display the full image at the center of the screen.

To create this form, add a second form to your *ShowTell* project and set its *Name* property to *frmImage*. Add an *Image* control named *imgOne*, and enter the following code on the form:

Source Code for FRMIMAGE.FRM

```
Option Explicit

Public Sub Display(strFileName As String)
    Dim lngWedge As Long
    Dim lngHedge As Long
    Me.Caption = strFileName
    With imgOne
        .Picture = LoadPicture(strFileName)
        .Move 0, 0
        lngWedge = Width - ScaleWidth
        lngHedge = Height - ScaleHeith

        Me.Move (Screen.Width - (.Width + lngWedge)) \ 2, _
            (Screen.Height - (.Height + lngHedge)) \ 2, _
            .Width + lngWedge, .Height + lngHedge
    End With
    Me.Show vbModal
End Sub
```

The WindChil Application

The WindChil application uses the same equation as that employed by the National Weather Service to calculate effective windchill given actual air temperature and wind speed. (If the scientific and technical aspects defining the windchill factor interest you, search for the keywords *wind chill* on the World Wide Web. I've found several sites with excellent explanations.)

This application demonstrates the Slider control, which I've set up to let the user select the current air temperature and wind speed. Figure 33-10 shows the WindChil form in action: the two sliders are set to a wind speed of 17 miles per hour and a temperature of 32 degrees Fahrenheit.

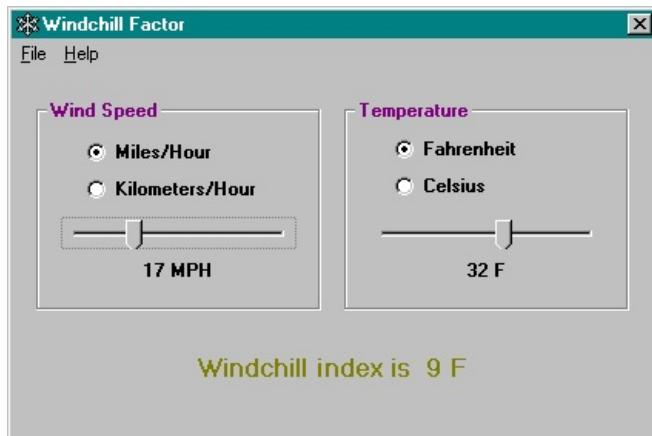


Figure 33-10. The WindChil form in action.

This application has some advantages over the standard charts you'll find posted on the World Wide Web and elsewhere. In my application, the temperature and wind speed can be set to any values in the ranges provided by the Slider controls, whereas the charts round off values to the nearest 5 or 10 degrees or miles per hour. The biggest advantage, however, is that with the addition of a few option buttons, I've made it easy to use metric values if you want. You have your choice of temperature scales (Fahrenheit or Celsius) and your choice of wind speed units (miles per hour or kilometers per hour). When you click an option button, the current settings of the sliders are retained, and the numbers are converted to the alternative units. Figure 33-11 shows the same settings as in Figure 33-10, here converted to metric units.

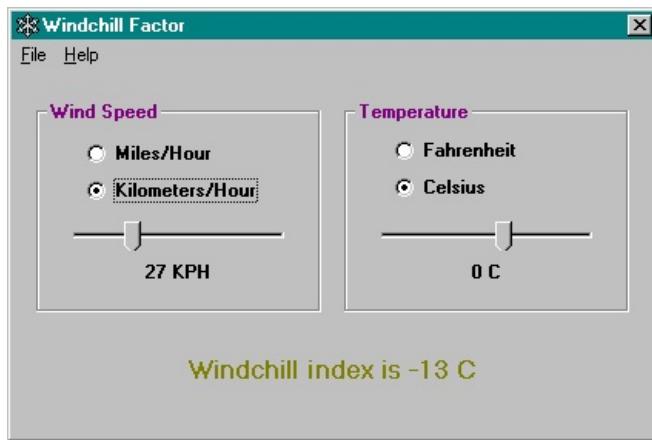


Figure 33-11. The WindChil form, showing values in metric units.

WINDCHIL.FRM

The Min and Max properties of the two sliders are set to the full range of accuracy for the windchill equation: 5 through 50 miles per hour for wind speed, and -50 through +90 degrees Fahrenheit for temperature. Values outside these ranges are invalid. The use of sliders rather than text box data entry fields simplifies the process of validating the numbers a user enters. If you use a Slider control, there's no way a user can enter an invalid wind speed or temperature value and no code is needed to check and enforce the entered values. In the "WINDCHIL.FRM Objects and Property Settings" table beginning on page 712, notice that the LargeChange property of the Slider controls is set to 1. This allows the user to click the slider at either side of the slider's knob and adjust the setting by 1 degree or 1 unit of wind.

speed. This technique works well for the range of values selected.

Figure 33-12 shows the WindChil form during development. I've set the ForeColor property of some of the text labels to red or blue to brighten the appearance of the form. You might prefer the default black ForeColor setting for your labels. You can experiment with the appearance of this form without affecting its calculations.

NOTE

I don't use Variant data types routinely, mostly because explicit data typing generally results in slightly faster code and more predictable intermediate calculation results, but in this program I decided to dimension most of the numeric variables as Variants as an experiment. It worked well. The program runs plenty fast enough, and the displayed results exactly match the various windchill charts I found on the Internet. Some people argue that Variants are actually easier to program than explicit variable types. I think this depends on the particular program and its design. Automatic conversions between strings and numbers can be very tricky, for instance, and I've seen some mighty strange and hard-to-track-down results from unexpected automatic type conversions. On the other hand, in some applications, such as WindChil, Variants work just great.

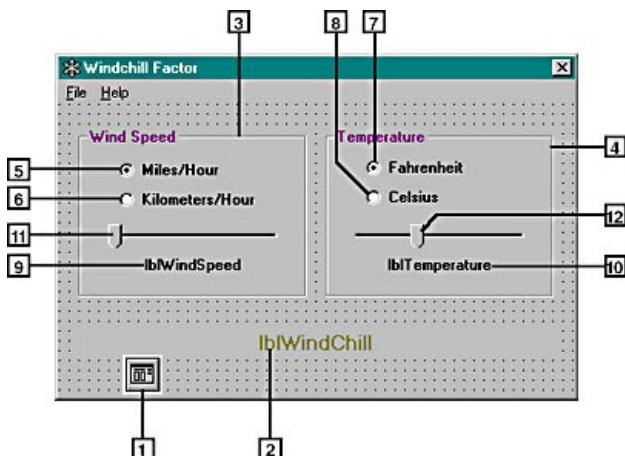


Figure 33-12. The WindChil form during development.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

WINDCHIL.FRM Menu Design Window Entries

Caption	Name	Indentation	Enabled
&File	mnuFile	0	<i>True</i>
&New	mnuNew	1	<i>False</i>
&Open	mnuOpen	1	<i>False</i>
&Save	mnuSave	1	<i>False</i>
Save &As	mnuSaveAs	1	<i>False</i>
-	mnuFileDash1	1	<i>True</i>
E&xit	mnuExit	1	<i>True</i>
&Help	mnuHelp	0	<i>True</i>
&Contents	mnuContents	1	<i>True</i>
&Search For Help	mnuSearch	1	<i>True</i>

-		<i>mnuHelpDash1</i>	1	<i>True</i>
<i>&About</i>	<i>Dear John, How Do I...</i>	<i>mnuAbout</i>	1	<i>True</i>

WINDCHIL.FRМ Objects and Property Settings

ID No.*	Property	Value
Form		
	Name	<i>frmWindChill</i>
	Caption	<i>Windchill Factor</i>
	BorderStyle	<i>3 - Fixed Dialog</i>
	Icon	<i>Snow.ico</i>
CommonDialog		
1	Name	<i>dlgOne</i>
Label		
2	Name	<i>lblWindChill</i>
	Alignment	<i>2 - Center</i>
	Font	<i>MS Sans Serif Bold 12</i>
	ForeColor	<i>&H00FF0000&</i>
Frame		
3	Name	<i>fraWindSpeed</i>
	Caption	<i>Wind Speed</i>
	ForeColor	<i>&H000000FF&</i>
Frame		
4	Name	<i>fraTemperature</i>
	Caption	<i>Temperature</i>
	ForeColor	<i>&H000000FF&</i>
OptionButton		
5	Name	<i>optMPH</i>
	Caption	<i>Miles/Hour</i>
	Value	<i>True</i>
6	Name	<i>optKPH</i>
	Caption	<i>Kilometers/Hour</i>
7	Name	<i>optFahrenheit</i>
	Caption	<i>Fahrenheit</i>
	Value	<i>True</i>
8	Name	<i>optCelsius</i>
	Caption	<i>Celsius</i>
Label		

9	Name	<i>lblWindSpeed</i>
	Alignment	<i>2 - Center</i>
10	Name	<i>lblTemperature</i>
	Alignment	<i>2 - Center</i>
Slider		
11	Name	<i>sldWindSpeed</i>
	LargeChange	<i>1</i>
	Max	<i>50</i>
	Min	<i>5</i>
12	Name	<i>sldTemperature</i>
	LargeChange	<i>1</i>
	Max	<i>90</i>
	Min	<i>-50</i>

* The number in the ID No. column corresponds to the number in Figure 33-12 that identifies the location of the object on the form.

Source Code for WINDCHIL.FRM

```
Option Explicit

Private Sub mnuAbout_Click()
    'Set properties
    With About
        .Application = "WindChill"
        .Heading = "Microsoft Visual Basic 6.0 " & _
            "Developer's Workshop"
        .Copyright = "1998 John Clark Craig"
        .Display
    End With
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    With dlgOne
        .HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
        .HelpCommand = cdlHelpContents
        .ShowHelp
    End With
End Sub

Private Sub mnuSearch_Click()
    With dlgOne

        .HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
        .HelpCommand = cdlHelpPartialKey
        .ShowHelp
    End With
End Sub
```

```

Private Function Celsius(vntF)
    'Convert Fahrenheit to Celsius
    Celsius = (vntF + 40) * 5 / 9 - 40
End Function

Private Sub ChillOut()
    Dim vntWind
    Dim vntTemp
    Dim vntChill
    Dim strDisplay As String
    'Get working values from scrollbars
    vntWind = sldWindSpeed.Value
    vntTemp = sldTemperature.Value
    'Convert to MPH if KPH selected
    If optKPH.Value = True Then
        vntWind = Mph(vntWind)
    End If
    'Convert to Fahrenheit if Celsius selected
    If optCelsius.Value = True Then
        vntTemp = Fahrenheit(vntTemp)
    End If
    'Calculate windchill index
    vntChill = Int(0.0817 * (Sqr(vntWind) * 3.71 +
        5.81 - 0.25 * vntWind) * (vntTemp - 91.4) + 91.4)
    'Convert back to Celsius if selected
    If optCelsius.Value = True Then
        vntChill = Celsius(vntChill)
    End If
    'Display windchill index
    strDisplay = "Windchill index is " & Str$(CInt(vntChill))
    If optFahrenheit.Value = True Then
        lblWindChill.Caption = strDisplay & " F"
    Else
        lblWindChill.Caption = strDisplay & " C"
    End If
End Sub

Private Sub cmdCancel_Click()
    'End if Cancel button clicked
    Unload frmWindChill
End Sub

Private Function Fahrenheit(vntC)
    'Convert Celsius to Fahrenheit
    Fahrenheit = (vntC + 40) * 9 / 5 - 40
End Function

Private Sub Form_Load()
    'Force scrollbars to update
    sldWindSpeed_Change
    sldTemperature_Change
End Sub

Private Sub sldTemperature_Change()
    Dim vntTmp
    'Get temperature
    vntTmp = sldTemperature.Value
    'Display using selected units
    If optCelsius.Value = True Then
        lblTemperature.Caption = vntTmp & " C"
    Else
        lblTemperature.Caption = vntTmp & " F"
    End If
    'Calculate windchill index

```

```
    ChillOut
End Sub

Private Sub sldTemperature_Scroll()
    'Update when slider knob moves
    sldTemperature_Change
End Sub

Private Sub sldWindSpeed_Change()
    Dim vntTmp
    'Get wind speed
    vntTmp = sldWindSpeed.Value
    'Display using selected units
    If optKPH.Value = True Then
        lblWindSpeed.Caption = vntTmp & " KPH"
    Else
        lblWindSpeed.Caption = vntTmp & " MPH"
    End If
    'Calculate windchill index
    ChillOut
End Sub

Private Sub sldWindSpeed_Scroll()
    'Update when slider knob moves
    sldWindSpeed_Change
End Sub

Private Function Kph(vntMph)
    'Convert MPH to KPH
    Kph = vntMph * 1.609344
End Function

Private Function Mph(vntKph)
    'Convert KPH to MPH
    Mph = vntKph / 1.609344
End Function

Private Sub optCelsius_Click()
    Dim vntC
    'Convert current temperature to Celsius
    vntC = Celsius(sldTemperature.Value)
    If vntC < -45 Then vntC = -45
    'Reset scrollbar for Celsius
    sldTemperature.Min = -45
    sldTemperature.Max = 32
    sldTemperature.Value = CInt(vntC)
    sldTemperature_Change
End Sub

Private Sub optFahrenheit_Click()
    Dim vntF
    'Convert current temperature to Fahrenheit
    vntF = Fahrenheit(sldTemperature.Value)
    If vntF < -50 Then vntF = -50
    'Reset scrollbar for Fahrenheit
    sldTemperature.Min = -50
    sldTemperature.Max = 90
    sldTemperature.Value = CInt(vntF)
    sldTemperature_Change
End Sub

Private Sub optKPH_Click()
    Dim vntK
    'Convert current wind speed to KPH
```

```
vntK = Kph(sldWindSpeed.Value)
`Reset scrollbar for KPH
sldWindSpeed.Min = 8
sldWindSpeed.Max = 80
sldWindSpeed.Value = vntK
sldWindSpeed_Change
End Sub

Private Sub optMPH_Click()
    Dim vntM

    `Convert current wind speed to MPH
    vntM = Mph(sldWindSpeed.Value)
    `Reset scrollbar for MPH
    sldWindSpeed.Min = 5
    sldWindSpeed.Max = 50
    sldWindSpeed.Value = vntM
    sldWindSpeed_Change
End Sub
```

Chapter Thirty-Four

Advanced Applications

This chapter covers a few advanced programming techniques that can enhance your Visual Basic development productivity. The *Messages* application demonstrates one way you can use commands embedded in an externally edited text file to control an application's behavior. The *Secret* application pieces together several forms and techniques presented earlier in this book to create a file encryption program. *BitPack* provides a working demonstration of a C-language dynamic link library (DLL) that enhances speed and utility, and the *Dialogs* application demonstrates several creative techniques that can enhance your Visual Basic forms.

The Messages Application

You can create your own programming, or scripting, language to perform specific tasks. I've created the Messages application as a simple example. This application displays a series of text boxes containing messages on the screen. It accomplishes this by declaring and using Message objects. The Message objects are defined by the class module MSG.CLS and its associated MSG.FRM file.

A Message object has a FileName property, which is set to the name of a specially formatted text file with the extension MES (a message file). (I describe the syntax of these message files below.) The MSG.CLS and MSG.FRM files work together to display messages from the message file. I've set up two special commands that can be typed into the message files to control the appearance and behavior of the displayed messages. I've kept these commands simple, but it would be easy to expand on the concept presented here if you wanted more creative control over the messages displayed.

Message File Syntax

Each message file contains blocks of text to be displayed. Three tilde characters (~~~) mark the start of each text block, and a message file can contain as many text blocks as you want. All messages in the selected message file are displayed sequentially. You can control the display of each message somewhat by including commands on the same line as each block separator.

An example message file helps clarify how this works. Here's the sample MESSAGE.MES file I've provided with this sample application on the companion CD-ROM:

MESSAGE.MES

This message file provides a sampling of the features demonstrated in the Messages application.

Note that all of these lines appearing before the first text block header will be ignored.

~~~

This is the first text block in the MESSAGE.MES file. Notice that the display window sizes automatically for the dimensions of the message.

Close this display window to proceed to the next text block in this file. Click the Close button in the upper-right corner of this message.

~~~ P 10

This message should automatically disappear in 10 seconds. You may close it manually before then if you want.

~~~ F 2

This message should be in a flashing window, with the flash rate set to 2 times per second.

~~~ F 5

This message should be in a flashing window, with the flash rate set to 5 times per second.

~~~ P 20 F 1

This is the last message in this file. The flash rate is 1 time per second, and the message will disappear automatically in 20 seconds if you don't close it manually before then.

This file defines five messages to be displayed sequentially by the Message object when it is activated in the Messages application. The first message is displayed in a nonflashing window and stays displayed until the user closes the window. The second text block's header contains the P command, which indicates that the message is to disappear after being displayed for the specified number of seconds. In this case, the second message disappears (and the next message appears) after a pause of 10 seconds. The third message is controlled by the F command, which sets a flash rate for the display window. The number after the F command is the toggle rate for the flashing—in this case, twice per second. Notice that in the final message of the file, both commands are given; this message will be displayed for 20 seconds,

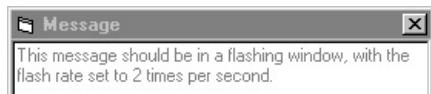
with a flash rate of once per second.

I've provided only the P and F commands, but it would be easy to add others. For example, you might want to add a C command to control the color of the message text. By studying the way these two commands work, you could easily add other commands on your own.

Figure 34-1 shows the first message as it's displayed, and Figure 34-2 shows the third message.



**Figure 34-1.** A message displayed by the Messages application.

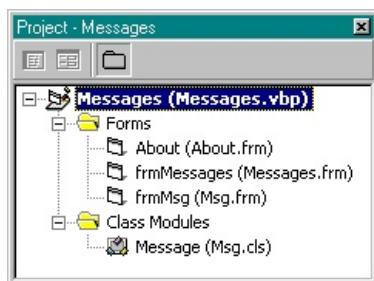


**Figure 34-2.** A message window that flashes to catch the user's attention.

### Why Use Message Files?

It would be fairly straightforward to embed the messages directly in an application's source code. Using external ASCII text files has some advantages, however. For example, to change the displayed messages you simply edit the message file using Notepad, WordPad, or any other text editor. Also, creating multiline text messages in Visual Basic code requires you to concatenate the lines using the `vbCrLf` constant, and making changes to these somewhat messy lines in your application can be tedious. An external message file has the same advantage as a resource file if you're creating applications to distribute internationally. Instead of editing and recompiling the application's source code for each foreign language, you need only change the message file's contents.

Figure 34-3 shows the project list for the Messages application. To add the messaging capabilities to your own applications, add the MSG.CLS file, the MSG.FRM file, and an activation code similar to that found in the MESSAGES.FRM demonstration form.

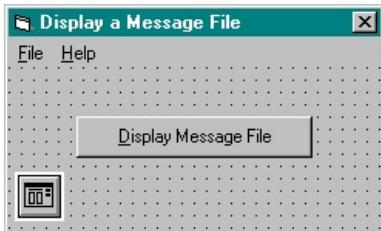


**Figure 34-3.** The Messages project list.

### MESSAGES.FRM

MESSAGES.FRM is the main startup form for the Messages application. In the source code for this application, you'll notice that a single instance of a Message object, `msgOne`, is declared and manipulated by this form. To sequentially display all messages defined in the MESSAGE.MES file, this form sets the declared Message object's `FileName` property to the selected file's path and filename and then calls the object's `Display` method. The rest of the displayed message's appearance is controlled by commands embedded within the message file itself.

I've added a `CommonDialog` control and a `CommandButton` control to this form so that you can select any other message file or files you might want to create for experimentation purposes. Figure 34-4 shows the Messages form during development.



**Figure 34-4.** The Messages form during development.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

### MESSAGES.FRM Menu Design Window Entries

| Caption                                    | Name         | Indentation | Enabled      |
|--------------------------------------------|--------------|-------------|--------------|
| &File                                      | mnuFile      | 0           | <i>True</i>  |
| &New                                       | mnuNew       | 1           | <i>False</i> |
| &Open Dear John, How Do I...               | mnuOpen      | 1           | <i>False</i> |
| &Save                                      | mnuSave      | 1           | <i>False</i> |
| Save &As Dear John, How Do I...            | mnuSaveAs    | 1           | <i>False</i> |
| -                                          | mnuFileDash1 | 1           | <i>True</i>  |
| E&xit                                      | mnuExit      | 1           | <i>True</i>  |
| &Help                                      | mnuHelp      | 0           | <i>True</i>  |
| &Contents                                  | mnuContents  | 1           | <i>True</i>  |
| &Search For Help On Dear John, How Do I... | mnuSearch    | 1           | <i>True</i>  |
| -                                          | mnuHelpDash1 | 1           | <i>True</i>  |
| &About Dear John, How Do I...              | mnuAbout     | 1           | <i>True</i>  |

### MESSAGES.FRM Objects and Property Settings

| Property             | Value                  |
|----------------------|------------------------|
| <b>Form</b>          |                        |
| Name                 | frmMessages            |
| BorderStyle          | 3 - Fixed Dialog       |
| Caption              | Display a Message File |
| <b>CommandButton</b> |                        |
| Name                 | cmdMessages            |
| Caption              | &Display Message File  |
| <b>CommonDialog</b>  |                        |
| Name                 | cdlOne                 |

### Source Code for MESSAGES.FRM

```
Option Explicit
```

```
Private Sub cmdMessages_Click()
```

```

`Declare new Message object
Dim msgOne As New Message
`Prompt user for message file (*.MES)
cdlOne.DialogTitle = "Message Files"
cdlOne.Flags = cdlOFNHideReadOnly
cdlOne.Filter = "Messages(*.mes)|*.mes"
cdlOne.CancelError = True
On Error Resume Next
cdlOne.ShowOpen
`Quit if user canceled or closed dialog box
If Err Then Exit Sub
On Error GoTo 0
`Display message file
With msgOne
    .FileName = cdlOne.FileName
    .Display
End With
End Sub

Private Sub Form_Load()
    `Center this form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
End Sub

Private Sub mnuAbout_Click()
    `Set properties
    About.Application = "Messages"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

```

## MSG.CLS

The MSG.CLS class module is the blueprint used to create Message objects. This particular class module requires an associated MSG.FRM file that provides the visual elements of the Message object. If you want to add Message objects to your applications, be sure to add both of these files.

Each Message object provides one property, FileName, which the calling application must set in order to display messages. Set the FileName property to the full path and filename of a selected message file. The only method provided by each Message object is Display, which starts the sequential display of all messages contained in the indicated message file.

There are quite a few lines of code in the Display method. This code interacts with, and controls, the

properties and methods of the associated *frmMsg* form. Commands embedded within the given message file are interpreted in the *Display* method, and the *frmMsg* form is controlled from here to provide the indicated operation.

### Source Code for MSG.CLS

```
Option Explicit

`Property that defines message file to be displayed
Public FileName As String

`Method to display message file
Public Sub Display()
    Dim strH As String
    Dim strJ As String
    Dim strA As String
    Dim strB As String
    Dim strC As String
    Dim intFilNum As Integer
    Dim lngNdx As Long
    Dim lngFlashRate As Long
    Dim lngPauseTime As Long
    Dim lngHeight As Long
    Dim lngWidth As Long
    Dim lngMaxTextWidth As Long
    `Get next available file I/O number
    intFilNum = FreeFile
    `Trap error if filename is invalid
    On Error Resume Next
    Open FileName For Input As #intFilNum
    If Err Then
        MsgBox "File not found: " & FileName
        Exit Sub
    End If
    On Error GoTo 0
    `Find start of first text block
    Do Until EOF(intFilNum)
        Line Input #intFilNum, strH
        `Skip lines until three tilde characters are found
        If InStr(strH, "~~~") = 1 Then
            strJ = UCASE$(strH)
            Exit Do
        End If
    Loop
    `Loop through all text blocks
    Do Until EOF(intFilNum)
        strB = ""
        strH = strJ
        lngWidth = 0
        lngHeight = 0
        `Load all of current text block
        Do Until EOF(intFilNum)
            Line Input #intFilNum, strA
            `End of this block is at start of next block
            If InStr(strA, "~~~") = 1 Then
                strJ = UCASE$(strA)
                Exit Do
            End If
            `Keep track of widest line of text
            lngMaxTextWidth = frmMsg.TextWidth(strA & "XX")
            If lngMaxTextWidth > lngWidth Then
                lngWidth = lngMaxTextWidth
            End If
            `Keep track of total height of all lines
```

```

        lngHeight = lngHeight + 1
        `Accumulate block of text lines
        If lngHeight > 1 Then
            strB = strB & vbCrLf & strA

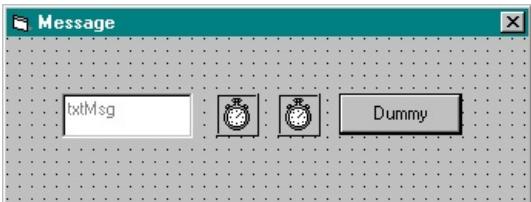
        Else
            strB = strA
        End If
    Loop
    `Check for flash rate in block header
    lngNdx = InStr(strH, "F")
    If lngNdx Then
        lngFlashRate = Val(Mid$(strH, lngNdx + 1))
    Else
        lngFlashRate = 0
    End If
    `Check for pause time in block header
    lngNdx = InStr(strH, "P")
    If lngNdx Then
        lngPauseTime = Val(Mid$(strH, lngNdx + 1))
    Else
        lngPauseTime = 0
    End If
    `Prepare message form's text box
    With frmMsg.txtMsg
        .Text = strB
        .Left = 0
        .Top = 0
        .Width = lngWidth
        .Height = (lngHeight + 1) * frmMsg.TextHeight("X")
    End With
    `Prepare message form
    With frmMsg
        .Width = .txtMsg.Width + (.Width - .ScaleWidth)
        .Height = .txtMsg.Height + (.Height - .ScaleHeight)
        .Left = (Screen.Width - .Width) \ 2
        .Top = (Screen.Height - .Height) \ 2
        `Set flash and pause properties if given
        If lngPauseTime > 0 Then .Pause = lngPauseTime
        If lngFlashRate > 0 Then .Flash = lngFlashRate
    End With
    `Show message and wait until it closes
    frmMsg.Show vbModal
Loop
End Sub

```

## MSG.FRМ

The *frmMsg* form is the working partner of the MSG.CLS class module. Together they form the basis for a Message object. Notice that MSG.FRМ interacts only with the MSG.CLS module. The main Messages form does not directly set any of the *frmMsg* form's properties, call any of its methods, or in any way directly interact with it. In this way, the *frmMsg* form becomes an integral part of the Message objects defined by the MSG.CLS class module.

MSG.FRМ has four controls: two Timer controls, a TextBox control to display the messages, and a dummy CommandButton control, which will be explained shortly. Figure 34-5 shows MSG.FRМ during the development process.



**Figure 34-5.** The MSG.FRM form during development.

To create this form, use the following table and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

### MSG.FRM Objects and Property Settings

| Property             | Value                      |
|----------------------|----------------------------|
| <b>Form</b>          |                            |
| Name                 | <i>frmMsg</i>              |
| Caption              | <i>Message</i>             |
| BorderStyle          | <i>3 - Fixed Dialog</i>    |
| <b>TextBox</b>       |                            |
| Name                 | <i>txtMsg</i>              |
| ForeColor            | <i>&amp;H00FF0000&amp;</i> |
| MultiLine            | <i>True</i>                |
| Locked               | <i>True</i>                |
| <b>Timer</b>         |                            |
| Name                 | <i>tmrTerminate</i>        |
| <b>Timer</b>         |                            |
| Name                 | <i>tmrFlash</i>            |
| <b>CommandButton</b> |                            |
| Name                 | <i>cmdDummy</i>            |
| Default              | <i>True</i>                |
| Caption              | <i>Dummy</i>               |

### Source Code for MSG.FRM

```

Option Explicit

Private Declare Function FlashWindow _ 
Lib "user32" (
    ByVal hwnd As Long,
    ByVal bInvert As Long
) As Long

Private Sub Form_Paint()
    'Remove focus from text box
    cmdDummy.Left = Screen.Width * 2
    cmdDummy.SetFocus
End Sub

Private Sub tmrTerminate_Timer()
    Unload Me

```

```
End Sub

Private Sub tmrFlash_Timer()
    'Toggle form flashing
    FlashWindow hwnd, CLng(True)
End Sub

Property Let Flash(PerSecond As Integer)
    'Set and activate form flashing rate
    tmrFlash.Interval = 1000 / PerSecond
    tmrFlash.Enabled = True
End Property

Property Let Pause(Seconds As Double)
    'Set and activate auto-unload timing
    tmrTerminate.Interval = 1000 * Seconds
    tmrTerminate.Enabled = True
End Property
```

The FlashWindow API function is called from the tmrFlash\_Timer event procedure to toggle the flashing of the MSG.FRM form. The Interval property of this timer determines the flash rate.

The Dummy command button's only purpose is to get the focus, and the flashing cursor, out of the text box while the message is displayed. As this form is painted, the Dummy command button's Left property is set to twice the width of the screen, guaranteeing that the button will be out of sight. By setting the button's Default property to *True*, you ensure that the focus goes with it.

The Flash and Pause properties are not set directly by a calling application. Instead, these properties are set by the MSG.CLS module to control the form's behavior.

## The Secret Application

There's a lot of talk about privacy and security nowadays, particularly in reference to the transfer of financial or other proprietary information over the Internet. The Secret application shouldn't be used for critical security situations, but it does provide a modicum of privacy for your e-mail or for any file that you'd rather not have others view indiscriminately.

### NOTE

The level of security provided by this application is not foolproof, and determined attackers, as the experts call them, could crack messages encrypted with this program. Realistically, though, your messages and files will be secure from the prying eyes of more than 99 percent of the population.

To keep this application very simple, and very legal for exporting overseas, I've used a private key technique rather than the sophisticated but slightly messy public key technology. If you use this application to encrypt e-mail, both you and the party at the other end must agree on a password phrase in advance. Any password phrase of reasonable length can be used as the private key, but the password string is hashed by the program to 24 bits of unique key data—well within the limit allowed by the authorities. Visual Basic's random number generator is used as the source of the pseudorandom bytes; it's generally accepted in the cryptography world that this is not a very secure technique. Even so, someone would have to be highly motivated to go to the trouble of cracking your Secret messages and files. If you need an extremely secure cipher, go with one of the commercial products on the market. If you just want a tool that will provide reasonable privacy and that's very easy to use, the Secret application will do the trick.

### NOTE

A hash of a string is a one-way calculation, kind of like a check sum, that's repeatable but not easily reversible. The same password will always hash to the same result, but given the result of the hash, there's no easy way to determine the original password.

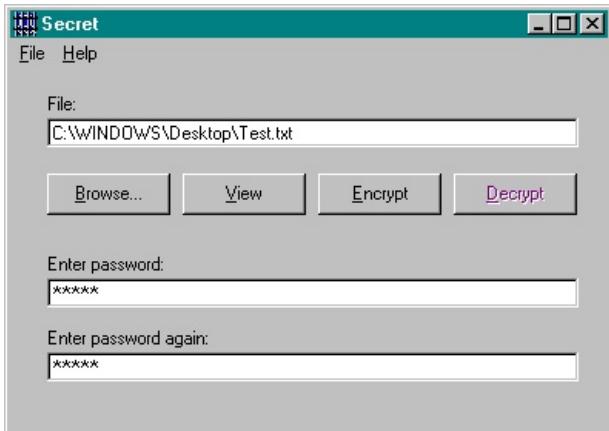
## How Does the Secret Application Work?

Here's how: You can select any file to encrypt or decrypt. A small header line is inserted at the beginning of encrypted files to allow Secret to detect whether a selected file is currently encrypted. Before you click either the Encrypt or the Decrypt button, enter a password in the appropriate field or fields. You must enter the same password twice for encryption, but it needs to be entered only once for decryption. This is a commonly used method that requires the user to type the password correctly in each box, thus preventing typographical errors from creeping in. Any mistyping of the password will prevent encryption of the file. The encrypted file is saved in a displayable, printable, e-mailable ASCII format, even if the original file contained binary data not suitable for these forms of output.

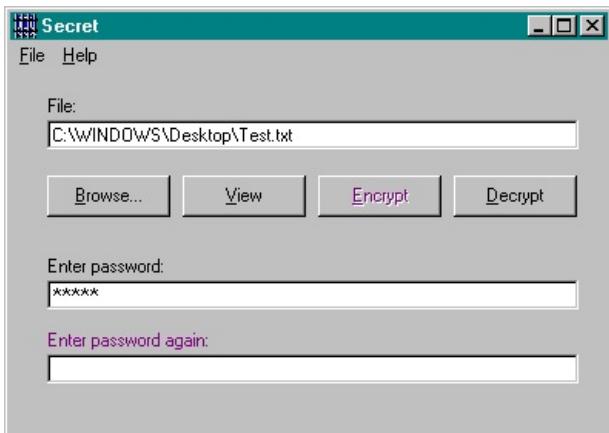
I've used an interesting technique that causes the message to appear different each time it's encrypted, even if the same password is used. This is accomplished by adding eight pseudorandom "salt" characters to the header line and hashing a combination of these salt characters with the password to form the rest of the header data. These 16 header characters are then used to encrypt the file, resulting in a unique result each time the file is encrypted. The header characters also provide a way to quickly check the accuracy of a password before the file is decrypted. The password and salt characters are hashed and the result is compared with the header line. An incorrect password results in the wrong hash.

Figure 34-6 shows the Secret application just after a text file named TEST.TXT has been selected and the short password *HAARP* has been entered twice. Because this file is not yet encrypted, the Encrypt button is enabled and the Decrypt button is not. Figure 34-7 shows the interface after the Encrypt button has been clicked; the file is now encrypted. Notice that the Decrypt button is now enabled, the Encrypt

button is not enabled, and the second password field is not enabled.



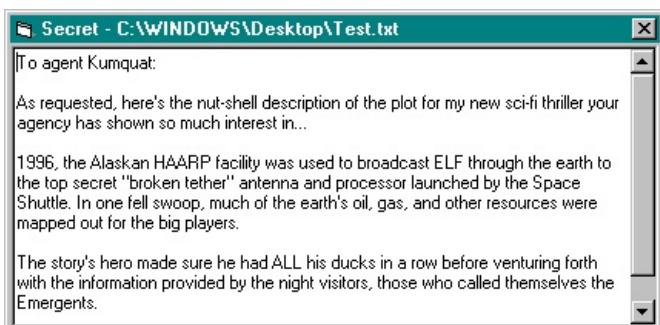
**Figure 34-6.** The TEST.TXT file selected to be encrypted by the Secret application.



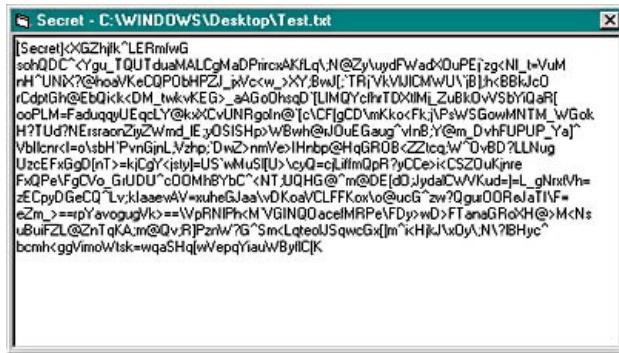
**Figure 34-7.** The TEST.TXT file, now in its encrypted state.

Figure 34-8 shows the sample TEST.TXT file before it's encrypted, and Figure 34-9 shows its contents in the encrypted state. Both of these views of the file contents were obtained by clicking the View button.

Figure 34-10 shows the same file after it has been decrypted and then encrypted a second time using the same password. Notice that the results of the two encryptions are entirely different, even though both versions decrypt to the same original file.



**Figure 34-8.** The original TEST.TXT file displayed using the View button.

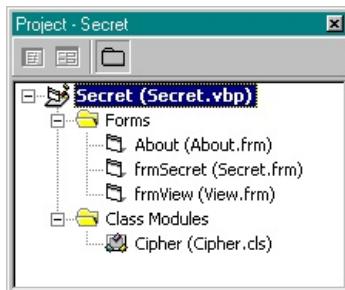


**Figure 34-9.** The encrypted TEST.TXT file displayed using the View button.



**Figure 34-10.** TEST.TXT encrypted a second time using the same password.

The Secret project comprises four files, as shown in the project list window in Figure 34-11. The CIPHER.CLS class module contains the same cipher class module that was presented in Chapter 18, "[Security](#)." The Secret form includes a Hash function that converts any string to a repeatable but unpredictable sequence of eight characters. This hash value is used to verify a user's password before a file is decrypted. The View form is a simple file-contents viewer that displays the contents of a selected file, ciphered or not, to let the user review the file in read-only mode.

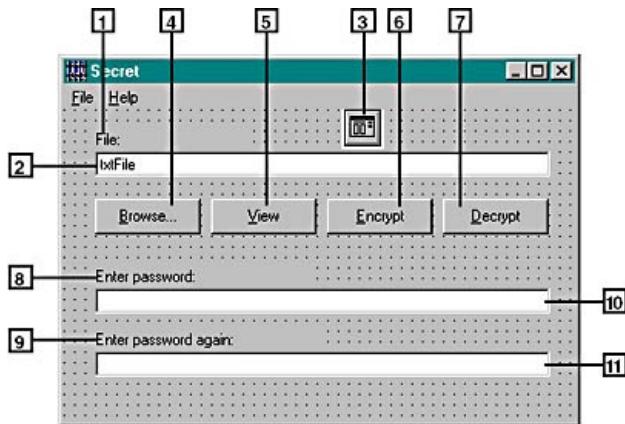


**Figure 34-11.** The project list for the Secret application.

## SECRET.FRM

SECRET.FRM is the main startup form for the Secret application. As shown in Figure 34-12, the form contains a text box for selecting a file to be processed, two text boxes for entering the password, and four command buttons that control all operations on the file. The CommonDialog control is used during the file selection process.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.



**Figure 34-12.** The Secret form during development.

#### SECRET.FRM Menu Design Window Entries

| Caption                                   | Name         | Indentation | Enabled      |
|-------------------------------------------|--------------|-------------|--------------|
| &File                                     | mnuFile      | 0           | <i>True</i>  |
| &New                                      | mnuNew       | 1           | <i>False</i> |
| &OpenDear John, How Do I...               | mnuOpen      | 1           | <i>False</i> |
| &Save                                     | mnuSave      | 1           | <i>False</i> |
| Save &AsDear John, How Do I...            | mnuSaveAs    | 1           | <i>False</i> |
| -                                         | mnuFileDash1 | 1           | <i>True</i>  |
| E&xit                                     | mnuExit      | 1           | <i>True</i>  |
| &Help                                     | mnuHelp      | 0           | <i>True</i>  |
| &Contents                                 | mnuContents  | 1           | <i>True</i>  |
| &Search For Help OnDear John, How Do I... | mnuSearch    | 1           | <i>True</i>  |
| -                                         | mnuHelpDash1 | 1           | <i>True</i>  |
| &AboutDear John, How Do I...              | mnuAbout     | 1           | <i>True</i>  |

#### SECRET.FRM Objects and Property Settings

| ID No.*             | Property | Value       |
|---------------------|----------|-------------|
| <b>Form</b>         |          |             |
|                     | Name     | frmSecret   |
|                     | Caption  | Secret      |
|                     | Icon     | Secur03.ico |
| <b>Label</b>        |          |             |
| 1                   | Name     | lblFile     |
|                     | Caption  | File:       |
| <b>TextBox</b>      |          |             |
| 2                   | Name     | txtFile     |
| <b>CommonDialog</b> |          |             |
| 3                   | Name     | cdlOne      |

| <b>CommandButton</b> |              |                                          |
|----------------------|--------------|------------------------------------------|
| 4                    | Name         | <i>cmdBrowse</i>                         |
|                      | Caption      | <i>&amp;BrowseDear John, How Do I...</i> |
| <b>CommandButton</b> |              |                                          |
| 5                    | Name         | <i>cmdView</i>                           |
|                      | Caption      | <i>&amp;View</i>                         |
| <b>CommandButton</b> |              |                                          |
| 6                    | Name         | <i>cmdEncrypt</i>                        |
|                      | Caption      | <i>&amp;Encrypt</i>                      |
| <b>CommandButton</b> |              |                                          |
| 7                    | Name         | <i>cmdDecrypt</i>                        |
|                      | Caption      | <i>&amp;Decrypt</i>                      |
| <b>Label</b>         |              |                                          |
| 8                    | Name         | <i>lblPassword1</i>                      |
|                      | Caption      | <i>Enter password:</i>                   |
| <b>Label</b>         |              |                                          |
| 9                    | Name         | <i>lblPassword2</i>                      |
|                      | Caption      | <i>Enter password again:</i>             |
| <b>TextBox</b>       |              |                                          |
| 10                   | Name         | <i>txtPassword1</i>                      |
|                      | PasswordChar | *                                        |
|                      | Text         | (blank)                                  |
| <b>TextBox</b>       |              |                                          |
| 11                   | Name         | <i>txtPassword2</i>                      |
|                      | PasswordChar | *                                        |
|                      | Text         | (blank)                                  |

\* The number in the ID No. column corresponds to the number in Figure 34-12 that identifies the location of the object on the form.

### Source Code for SECRET.FRM

```

Option Explicit

Private Sub cmdBrowse_Click()
    'Prompt user for filename
    cdlOne.DialogTitle = "Secret"
    cdlOne.Flags = cdloFNHideReadOnly
    cdlOne.Filter = "All files (*.*)|*.*"
    cdlOne.CancelError = True
    On Error Resume Next
    cdlOne.ShowOpen
    'Grab filename
    If Err = 0 Then
        txtFile.Text = cdlOne.FileName
    End If
End Sub

```

```
End If
On Error GoTo 0
End Sub

Private Sub cmdEncrypt_Click()
    'Make sure both passwords match exactly
    If txtPassword1.Text <> txtPassword2.Text Then
        MsgBox "The two passwords are not the same!", _
            vbExclamation, "Secret"
        Exit Sub
    End If
    'Encrypt file
    MousePointer = vbHourglass
    cmdEncrypt.Enabled = False
    cmdDecrypt.Enabled = False
    cmdView.Enabled = False
    cmdBrowse.Enabled = False
    Refresh
    Encrypt
    txtFile_Change
    MousePointer = vbDefault
End Sub

Private Sub cmdDecrypt_Click()
    MousePointer = vbHourglass
    cmdEncrypt.Enabled = False
    cmdDecrypt.Enabled = False
    cmdView.Enabled = False
    cmdBrowse.Enabled = False
    Refresh
    Decrypt
    txtFile_Change
    MousePointer = vbDefault
End Sub

Private Sub cmdView_Click()
    Dim strA As String
    Dim lngZndx As Long
    MousePointer = vbHourglass
    'Get file contents
    Open txtFile.Text For Binary As #1
    strA = Space$(LOF(1))
    Get #1, , strA
    Close #1
    Do
        lngZndx = InStr(strA, Chr$(0))
        If lngZndx = 0 Or lngZndx > 5000 Then Exit Do
        Mid$(strA, lngZndx, 1) = Chr$(1)
    Loop
    'Display file contents
    MousePointer = vbDefault
    frmView.rtfView.Text = strA
    frmView.Caption = "Secret - " & txtFile.Text
    frmView.Show vbModal
End Sub

Private Sub Form_Load()
    'Center this form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
    'Disable most command buttons
    cmdEncrypt.Enabled = False
    cmdDecrypt.Enabled = False
    cmdView.Enabled = False
```

```
'Initialize filename field
txtFile.Text = ""
End Sub

Private Sub mnuAbout_Click()
    'Set properties
    About.Application = "Secret"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

Private Sub txtFile_Change()
    Dim lngFileLen As Long
    Dim strHead As String
    'Check to see whether file exists
    On Error Resume Next
    lngFileLen = Len(Dir(txtFile.Text))
    'Disable buttons if filename isn't valid
    If Err <> 0 Or lngFileLen = 0 Or Len(txtFile.Text) = 0 Then
        cmdEncrypt.Enabled = False
        cmdDecrypt.Enabled = False
        cmdView.Enabled = False
        lblPassword1.Enabled = False
        txtPassword1.Enabled = False
        lblPassword2.Enabled = False
        txtPassword2.Enabled = False
        txtPassword2.Text = ""
        Exit Sub
    End If
    'Get first 8 bytes of selected file
    Open txtFile.Text For Binary As #1
    strHead = Space(8)
    Get #1, , strHead
    Close #1
    'Check to see whether file is already encrypted
    If strHead = "[Secret]" Then
        cmdEncrypt.Enabled = False
        cmdDecrypt.Enabled = True
        lblPassword2.Enabled = False
        txtPassword2.Enabled = False
        txtPassword2.Text = ""
    Else
        cmdEncrypt.Enabled = True
        cmdDecrypt.Enabled = False
        lblPassword2.Enabled = True
        txtPassword2.Enabled = True
    End If
End Sub
```

```
End If
lblPassword1.Enabled = True
txtPassword1.Enabled = True
cmdBrowse.Enabled = True
cmdView.Enabled = True
End Sub

Sub Encrypt()
    Dim strHead As String
    Dim strT As String
    Dim strA As String
    Dim cphX As New Cipher
    Dim lngN As Long
    Open txtFile.Text For Binary As #1
    'Load entire file into strA
    strA = Space$(LOF(1))
    Get #1, , strA
    Close #1
    'Prepare header string with salt characters
    strT = Hash(Date & Str(Timer))
    strHead = "[Secret]" & strT & Hash(strT & txtPassword1.Text)
    'Do the encryption
    cphX.KeyString = strHead
    cphX.Text = strA
    cphX.DoXor
    cphX.Stretch
    strA = cphX.Text
    'Write header
    Open txtFile.Text For Output As #1
    Print #1, strHead
    'Write encrypted data
    lngN = 1
    Do
        Print #1, Mid(strA, lngN, 70)
        lngN = lngN + 70
    Loop Until lngN > Len(strA)
    Close #1
End Sub

Sub Decrypt()
    Dim strHead As String
    Dim strA As String
    Dim strT As String
    Dim cphX As New Cipher
    Dim lngN As Long
    'Get header (first 18 bytes of encrypted file)
    Open txtFile.Text For Input As #1
    Line Input #1, strHead
    Close #1
    'Check for correct password
    strT = Mid(strHead, 9, 8)
    If InStr(strHead, Hash(strT & txtPassword1.Text)) <> 17 Then
        MsgBox "Sorry, this is not the correct password!", _
            vbExclamation, "Secret"
        Exit Sub
    End If
    'Get file contents
    Open txtFile.Text For Input As #1
    'Read past the header
    Line Input #1, strHead
    'Read and build the contents string
    Do Until EOF(1)
        Line Input #1, strT
        strA = strA & strT
    Loop
End Sub
```

```

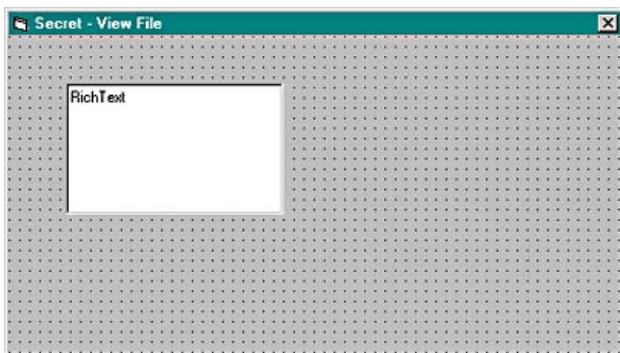
Loop
Close #1
`Decrypted file contents
cphX.KeyString = strHead
cphX.Text = strA
cphX.Shrink
cphX.DoXor
strA = cphX.Text
`Replace file with decrypted version
Kill txtFile.Text
Open txtFile.Text For Binary As #1
Put #1, , strA
Close #1
End Sub

Function Hash(strA As String) As String
Dim cphHash As New Cipher
cphHash.KeyString = strA & "123456"
cphHash.Text = strA & "123456"
cphHash.DoXor
cphHash.Stretch
cphHash.KeyString = cphHash.Text
cphHash.Text = "123456"
cphHash.DoXor
cphHash.Stretch
Hash = cphHash.Text
End Function

```

## **VIEW.FRM**

This relatively simple form provides a read-only file viewer window for the Secret application. Figure 34-13 shows the View form during development. The only control this form contains is a RichTextBox control to display the selected file. To create this form, use the following table and source code to add the appropriate control, set any nondefault properties as indicated, and enter the source code lines as shown.



**Figure 34-13.** The View form during development.

### **VIEW.FRM Objects and Property Settings**

| Property    | Value                     |
|-------------|---------------------------|
| <b>Form</b> |                           |
| Name        | <i>frmView</i>            |
| Caption     | <i>Secret - View File</i> |
| MaxButton   | <i>False</i>              |
| MinButton   | <i>False</i>              |

**RichTextBox**

|            |                    |
|------------|--------------------|
| Name       | <i>rtfView</i>     |
| ScrollBars | <i>3 - rtfBoth</i> |
| Locked     | <i>True</i>        |

**Source Code for VIEW.FRM**

```
Option Explicit

Dim mblnBeenHereDoneThis As Boolean

Private Sub Form_Load()
    mblnBeenHereDoneThis = False
End Sub

Private Sub Form_Resize()
    'Center this form, but only the first time
    If mblnBeenHereDoneThis = False Then
        Me.Left = (Screen.Width - Me.Width) \ 2
        Me.Top = (Screen.Height - Me.Height) \ 2
        mblnBeenHereDoneThis = True
    End If
    'Size RichTextBox to fill form
    rtfView.Move 0, 0, Me.ScaleWidth, Me.ScaleHeight
End Sub
```

**CIPHER.CLS**

This class module provides the cipher engine at the heart of the Secret application. The Cipher object was described in Chapter 18, "[Security](#)," but I'll present it again here for easy reference. To create the Cipher class, add a new class module to your project, name it Cipher, and enter the source code lines as shown.

**Source Code for CIPHER.CLS**

```
`CIPHER.CLS
Option Explicit

Private mstrKey As String
Private mstrText As String

`~~~.KeyString
`A string (key) used in encryption and decryption
Public Property Let KeyString(strKey As String)
    mstrKey = strKey
    Initialize
End Property

`~~~.Text
`Write text to be encrypted or decrypted
Public Property Let Text(strText As String)
    mstrText = strText
End Property

`Read text that was encrypted or decrypted
Public Property Get Text() As String
    Text = mstrText
End Property

`~~~.DoXor
```

```

`Exclusive-or method to encrypt or decrypt
Public Sub DoXor()
    Dim lngC As Long
    Dim intB As Long
    Dim lngN As Long
    For lngN = 1 To Len(mstrText)
        lngC = Asc(Mid(mstrText, lngN, 1))
        intB = Int(Rnd * 256)
        Mid(mstrText, lngN, 1) = Chr(lngC Xor intB)
    Next lngN
End Sub

`~~~.Stretch
`Convert any string to a printable, displayable string
Public Sub Stretch()
    Dim lngC As Long
    Dim lngN As Long
    Dim lngJ As Long
    Dim lngK As Long
    Dim lngA As Long
    Dim strB As String
    lngA = Len(mstrText)
    strB = Space(lngA + (lngA + 2) \ 3)
    For lngN = 1 To lngA
        lngC = Asc(Mid(mstrText, lngN, 1))
        lngJ = lngJ + 1
        Mid(strB, lngJ, 1) = Chr((lngC And 63) + 59)
        Select Case lngN Mod 3
            Case 1
                lngK = lngK Or ((lngC \ 64) * 16)
            Case 2
                lngK = lngK Or ((lngC \ 64) * 4)
            Case 0
                lngK = lngK Or (lngC \ 64)
                lngJ = lngJ + 1
                Mid(strB, lngJ, 1) = Chr(lngK + 59)
                lngK = 0
        End Select
    Next lngN
    If lngA Mod 3 Then
        lngJ = lngJ + 1
        Mid(strB, lngJ, 1) = Chr(lngK + 59)
    End If
    mstrText = strB
End Sub

`~~~.Shrink
`Inverse of the Stretch method;
`result can contain any of the 256-byte values
Public Sub Shrink()
    Dim lngC As Long
    Dim lngD As Long
    Dim lngE As Long
    Dim lngA As Long
    Dim lngB As Long
    Dim lngN As Long
    Dim lngJ As Long
    Dim lngK As Long
    Dim strB As String
    lngA = Len(mstrText)
    lngB = lngA - 1 - (lngA - 1) \ 4
    strB = Space(lngB)
    For lngN = 1 To lngB
        lngJ = lngJ + 1

```

```
    lngC = Asc(Mid(mstrText, lngJ, 1)) - 59
    Select Case lngN Mod 3
        Case 1
            lngK = lngK + 4
            If lngK > lngA Then lngK = lngA
            lngE = Asc(Mid(mstrText, lngK, 1)) - 59
            lngD = ((lngE \ 16) And 3) * 64
        Case 2
            lngD = ((lngE \ 4) And 3) * 64
        Case 0
            lngD = (lngE And 3) * 64
            lngJ = lngJ + 1
    End Select
    Mid(strB, lngN, 1) = Chr(lngC Or lngD)
Next lngN
mstrText = strB
End Sub

`Initializes random numbers using the key string
Private Sub Initialize()
    Dim lngN As Long
    Randomize Rnd(-1)
    For lngN = 1 To Len(mstrKey)
        Randomize Rnd(-Rnd * Asc(Mid(mstrKey, lngN, 1)))
    Next lngN
End Sub
```

## The BitPack Application

I developed the BitPack application while experimenting with C-language DLLs to see how much extra speed I could squeeze out of Visual Basic. Even with the native code compiler available in Visual Basic, a C DLL can make a real difference in the speed-critical sections of your Visual Basic applications. The BitPack application demonstrates this well.

I designed BITPACK.DLL to manipulate individual bits in a byte array, a task that Visual Basic is not particularly well suited for. This DLL provides three functions: BitGet, to return the current state of a bit; BitSet, to set the bit at the given location to 1; and BitClr, to set the bit to 0. You pass a byte array to these functions, along with a bit number, and the C code does the rest. For example, to retrieve bit number 542 from a byte array, the C code in the BitGet routine efficiently locates bit number 6 from byte number 67 in the array, extracts the bit, and returns 1 if that bit is set or 0 if it isn't. Byte arrays can be huge in 32-bit Visual Basic, so a practically unlimited store of bits can be accessed by these routines, all stored in a single byte array.

## Generating a Table of Prime Numbers (Sieve of Eratosthenes)

A practical use for these functions is in the field of data acquisition and process control. The state of thousands of switches, contact closures, and the like can be maintained in a byte array using only the three functions in the BitPack application (BitGet, BitSet, and BitClr). For this book, however, I created a small program to generate a table of prime numbers using the well-known sieve of Eratosthenes. Each bit in the byte array represents an odd integer. Using a couple of nested loops, it's easy to toggle all bits representing nonprime numbers to 1s, leaving all primes as 0s. Because of the DLL's speed, we can generate a table of primes in the range 1 through 1,000,000, for example, in just a few seconds. When you realize how many times the BitSet and BitGet functions are called to generate this table, you begin to get a sense of how much the C-language DLL functions can speed up some types of code!

## Creating the BitPack DLL Project Files

Before creating the Visual Basic application to generate the prime numbers table, you must create the DLL that is at the core of its operation.

### NOTE

With the latest Visual C++ compilers, creating a DLL is easier than ever. Because this book focuses on using Microsoft tools to create applications for the 32-bit Windows 95 environment, I've streamlined the following example DLL code. This should make it easier for you to focus on the essential points of the DLL creation task. If you're using a version of C other than Microsoft Visual C++ 6.0, you might need to make some adjustments to the following listings and development steps. For the best in-depth explanation of every aspect of DLL creation, refer to the documentation that comes with your compiler.

The following three listings are for the only three files you'll need to modify in your Visual C++ project. Start a new project in the 32-bit version of Visual C++, select Win32 Dynamic-Link Library as the type of project to be built, enter *BITPACK* as the project name, and click OK. When prompted for the type of DLL to create, select the ADLL That Exports Some Symbols option and click Finish. Visual C++ will create a new DLL project, complete with source files that you can modify. Create a new file named *BITPACK.DEF* and add it to your project. Add the following few lines to *BITPACK.DEF*:

```
;BitPack.def
LIBRARY BitPack
```

```
EXPORTS
    BitGet
    BitSet
    BitClr
```

The DEF file tells the outside world the names of exported functions—in other words, this file provides the list of functions you can call from your Visual Basic applications.

Two of the files Visual C++ automatically created for your project will need to be modified. The BITPACK.CPP source code file is a short file containing the efficient single-line functions that perform all the addressing, masking, and other bit manipulations required to access or process a single bit anywhere in a huge byte array. Edit the contents of BITPACK.CPP as follows:

```
// BitPack.cpp : Defines the entry point for the DLL application
//

#include "stdafx.h"
#include "BitPack.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

BITPACK_API int __stdcall BitGet(LPBYTE bytes, int bitpos)
{
    return( bytes[bitpos >> 3] & (1 << (bitpos % 8)) ? 1: 0 );
}

BITPACK_API int __stdcall BitSet(LPBYTE bytes, int bitpos)
{
    return( bytes[bitpos >> 3] |= (1 << (bitpos % 8)));
}

BITPACK_API int __stdcall BitClr(LPBYTE bytes, int bitpos)
{
    return( bytes[bitpos >> 3] &= ~(1 << (bitpos % 8)));
}
```

The only other file to edit is BITPACK.H, which provides declarations for the new BitPack functions. Edit the contents of BITPACK.H as shown below. Notice that most of this file does not require any alteration to the code that Visual C++ automatically creates for you:

```
// The following ifdef block is the standard way of creating
// macros that make exporting from a DLL simpler. All files
// within this DLL are compiled with the BITPACK_EXPORTS
// symbol defined on the command line. This symbol should not
// be defined on any project that uses this DLL. In this way, any
// other project whose source files include this file see
// BITPACK_API functions as being imported from a DLL, whereas
// this DLL sees symbols defined with this macro as being exported.
#ifndef BITPACK_EXPORTS
#define BITPACK_API __declspec(dllexport)
#else
#define BITPACK_API __declspec(dllimport)
#endif

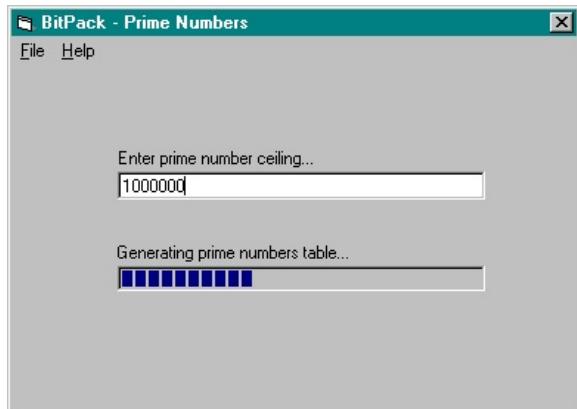
BITPACK_API int __stdcall BitGet(LPBYTE bytes, int bitpos);
BITPACK_API int __stdcall BitSet(LPBYTE bytes, int bitpos);
BITPACK_API int __stdcall BitClr(LPBYTE bytes, int bitpos);
```

Click the Build button in the Visual C++ environment to compile and link the two files in your project and create a small DLL module named BITPACK.DLL. Move or copy BITPACK.DLL to your Windows SYSTEM directory so that your Visual Basic Declare statements will be able to find the DLL file

automatically. BitPack's three functions can then be declared and called from a Visual Basic program located anywhere in your system.

## BITPACK.FRM

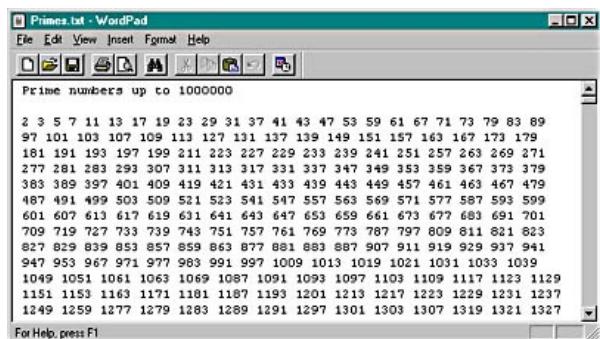
This form prompts the user for the largest desired prime number. It then calls the functions within the BitPack DLL to generate a table of prime numbers represented by bits in a byte array and creates an output file of the results. I've added a progress indicator bar to the form so that you can monitor the speed of the prime number calculations. On my computer, the generation of the prime numbers table is faster than the creation of the output file! Figure 34-14 shows the BitPack form in action, as it calculates all prime numbers up to 1,000,000.



**Figure 34-14.** The BitPack form in action.

The output file of prime numbers is written to the file C:\WINDOWS\DESKTOP\PRIMES.TXT, but you can change the location or the filename. The path and filename string is isolated for easy maintenance in a constant named *FileName*, near the top of the BITPACK.FRM source code. If you elect to generate a large table of prime numbers, this file can become fairly large. To roughly predict the size of the output file, cut the largest prime number value in half. For example, the generation of prime numbers up to 200,000 creates a PRIMES.TXT file of just a little over 100,000 bytes in size.

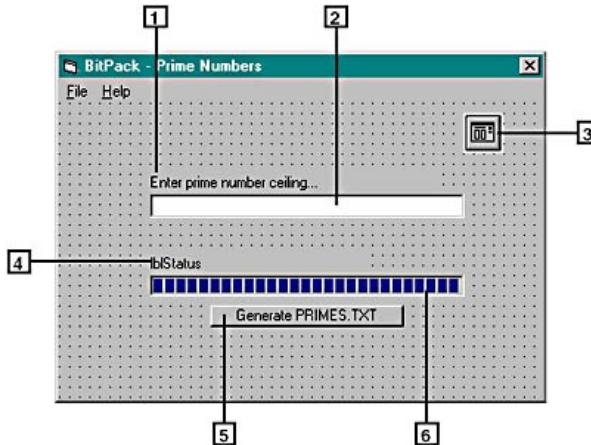
Figure 34-15 shows the first lines of the contents of PRIMES.TXT after the BitPack application has generated prime numbers up to 1,000,000.



**Figure 34-15.** The PRIMES.TXT file listing the generated prime numbers.

The BitPack form provides a working example of the ProgressBar control. It monitors the progress of the application as it creates the prime numbers table and also while it creates the PRIMES.TXT output file. I've toggled the Visible properties of the ProgressBar and CommandButton controls so that you'll always see one or the other, but not both at the same time.

Figure 34-16 shows the BitPack form during development.



**Figure 34-16.** The BitPack form during development.

To create this form, use the following tables and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

#### BITPACK.FRM Menu Design Window Entries

| Caption                                       | Name         | Indentation | Enabled      |
|-----------------------------------------------|--------------|-------------|--------------|
| &File                                         | mnuFile      | 0           | <i>True</i>  |
| &New                                          | mnuNew       | 1           | <i>False</i> |
| &Open<br>Dear John, How Do I...               | mnuOpen      | 1           | <i>False</i> |
| &Save                                         | mnuSave      | 1           | <i>False</i> |
| Save &As<br>Dear John, How Do I...            | mnuSaveAs    | 1           | <i>False</i> |
| -                                             | mnuFileDash1 | 1           | <i>True</i>  |
| E&xit                                         | mnuExit      | 1           | <i>True</i>  |
| &Help                                         | mnuHelp      | 0           | <i>True</i>  |
| &Contents                                     | mnuContents  | 1           | <i>True</i>  |
| &Search For Help On<br>Dear John, How Do I... | mnuSearch    | 1           | <i>True</i>  |
| -                                             | mnuHelpDash1 | 1           | <i>True</i>  |
| &About<br>Dear John, How Do I...              | mnuAbout     | 1           | <i>True</i>  |

#### BITPACK.FRM Objects and Property Settings

| ID No.*             | Property    | Value                                                        |
|---------------------|-------------|--------------------------------------------------------------|
| <b>Fmorm</b>        |             |                                                              |
|                     | Name        | <i>frmBitPack</i>                                            |
|                     | BorderStyle | <i>3 - Fixed Dialog</i>                                      |
|                     | Caption     | <i>BitPack - Prime Numbers</i>                               |
| <b>Label</b>        |             |                                                              |
| 1                   | Name        | <i>lblPrompt</i>                                             |
|                     | Caption     | <i>Enter prime number ceiling<br/>Dear John, How Do I...</i> |
| <b>TextBox</b>      |             |                                                              |
| 2                   | Name        | <i>txtMaxPrime</i>                                           |
| <b>CommonDialog</b> |             |                                                              |

|                      |         |                            |
|----------------------|---------|----------------------------|
| 3                    | Name    | <i>cdlOne</i>              |
| <b>Label</b>         |         |                            |
| 4                    | Name    | <i>lblStatus</i>           |
| <b>CommandButton</b> |         |                            |
| 5                    | Name    | <i>cmdPrimes</i>           |
|                      | Caption | <i>Generate PRIMES.TXT</i> |
| <b>ProgressBar</b>   |         |                            |
| 6                    | Name    | <i>prgOne</i>              |

\* The number in the ID No. column corresponds to the number in Figure 34-16 that identifies the location of the object on the form.

### Source Code for BITPACK.FRM

Option Explicit

```

Private Declare Function BitGet _
Lib "BitPack.dll" (
    ByRef bytB As Byte,
    ByVal lngN As Long
) As Long

Private Declare Function BitSet _
Lib "BitPack.dll" (
    ByRef bytB As Byte,
    ByVal lngN As Long
) As Long

Private Declare Function BitClr _
Lib "BitPack.dll" (
    ByRef bytB As Byte,
    ByVal lngN As Long
) As Long

`Change output path or filename here
Const FileName = "C:\Windows\Desktop\Primes.txt"

Private Sub cmdPrimes_Click()
    Dim lngN As Long
    Dim lngI As Long
    Dim lngJ As Long
    Dim lngK As Long
    Dim lngNext As Long
    Dim lngLast As Long
    Dim bytAry() As Byte
    Dim strP As String
    `Show hourglass while busy
    MousePointer = vbHourglass
    cmdPrimes.Visible = False
    prgOne.Visible = True
    prgOne.Value = 0
    `Get largest prime number specified
    lngN = Abs(Val(txtMaxPrime.Text))
    `Match only odd numbers to bits in byte array
    ReDim bytAry(lngN \ 16)
    `Keep user informed of progress
    lblStatus.Caption = "Generating prime numbers tableDear John, How Do I... "
    Refresh
    `Process byte array; 0 bits represent prime numbers
    lngK = (lngN - 3) \ 2
    For lngI = 0 To lngK
        If lngI Mod 2 = 1 Then
            bytAry(lngI \ 2) = BitGet(bytAry(lngI \ 2), lngI Mod 2)
        End If
    Next
    prgOne.Value = 100
    cmdPrimes.Visible = True
    prgOne.Visible = False
End Sub

```

```

`If next number is primeDear John, How Do I...
If BitGet(bytAry(0), lngI) = 0 Then
    `Dear John, How Do I... set bits that are multiples
    For lngJ = 3 * lngI + 3 To lngK Step 2 * lngI + 3
        BitSet bytAry(0), lngJ
    Next lngJ
    `Update progress bar, but not too often
    lngNext = Int(100 * lngI / lngK)
    If lngNext <> lngLast Then
        lngLast = lngNext
        prgOne.Value = lngNext
    End If
End If
Next lngI
`Keep user informed
lblStatus.Caption = "Writing prime numbers fileDear John, How Do I... "
lngLast = 0
prgOne.Value = 0
Refresh
`Write primes to file on desktop
Open FileName For Output As #1
`Bit table starts at 3, so output 2 as prime
Print #1, "Prime numbers up to" & Str$(lngN) & vbCrLf
strP = "2"
For lngI = 0 To lngK
    `If prime numberDear John, How Do I...
    If BitGet(bytAry(0), lngI) = 0 Then
        `Concatenate number to string for output
        strP = strP & Str$(lngI + lngI + 3)
        `If string is long enoughDear John, How Do I...
        If Len(strP) > 65 Then
            `Output string to file
            Print #1, LTrim$(strP)
            `Prepare for next line of output
            strP = ""
        `Update progress bar, but not too often
        lngNext = Int(100 * lngI / lngK)
        If lngNext > lngLast Then
            lngLast = lngNext
            prgOne.Value = lngNext
        End If
    End If
End If
Next lngI
`Print any last-line primes
Print #1, LTrim$(strP)
Close #1
`Set form to original visible state
lblStatus.Caption = ""
cmdPrimes.Visible = True
prgOne.Visible = False
MousePointer = vbDefault
End Sub

Private Sub Form_Load()
    txtMaxPrime.Text = ""
    lblStatus.Caption = ""
    prgOne.Visible = False
End Sub

Private Sub mnuAbout_Click()
    `Set properties
    About.Application = "BitPack"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"

```

```
About.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub
```

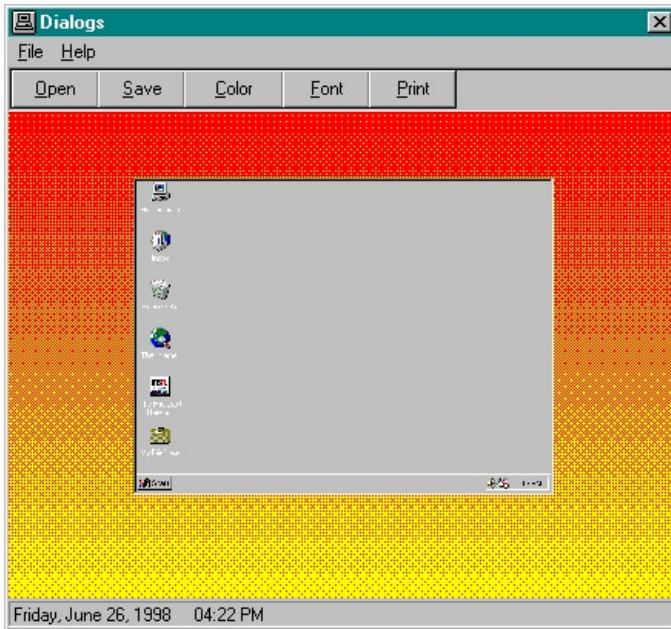
On the BitPack form, all calls to the bit manipulation functions in BITPACK.DLL pass the first member of the byte array *bytAry()*. You can also pass a single nonarray byte variable to these functions, in which case the *BitPos* parameter should stay in the range 0 through 7. For maximum speed, I elected not to include range-checking code within the DLL, so it's up to you to develop the code to prevent your application from passing *BitPos* values outside the range of a byte value or a byte array. Because there are 8 bits per byte-array element, an array dimensioned with the value 100, for example, has a range of legal *BitPos* values from 0 through 807.

To compute prime numbers using the sieve of Eratosthenes, I mapped the odd integers 3, 5, 7, Dear John, How Do I... to the bits 0, 1, 2, Dear John, How Do I... in the byte array. This allows a range of 16 integers to be covered in each 8-bit byte element. Because Visual Basic supports huge byte arrays, you can theoretically compute primes up to a very high value using this program.

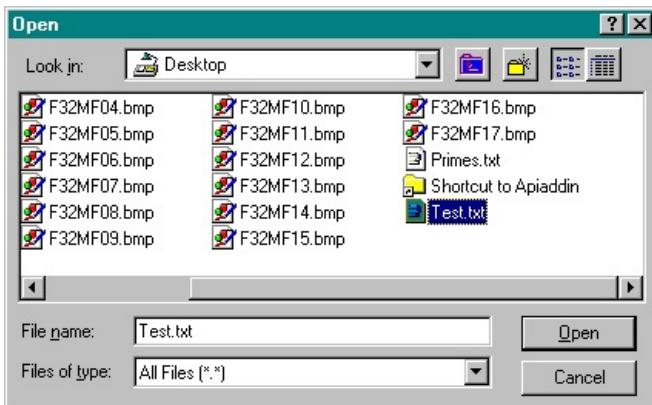
## The Dialogs Application

The CommonDialog control provides many powerful options for interacting with users in a standard way. The Dialogs application illustrates the five common dialog boxes provided by this one control. I've set up five buttons in a toolbar to activate the Open, Save As, Color, Font, and Print dialog boxes. The user's options in each dialog box are displayed for verification, but no files or settings are affected. You can select any file on your system while the Save As dialog box is displayed, for instance, but the file is left unaffected.

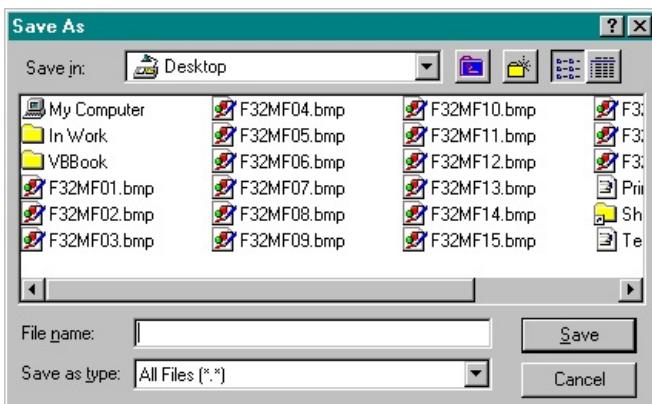
Figure 34-17 shows the Dialogs application at runtime, and Figures 34-18 through 34-22 show the dialog boxes that appear when you click the associated buttons on the toolbar.

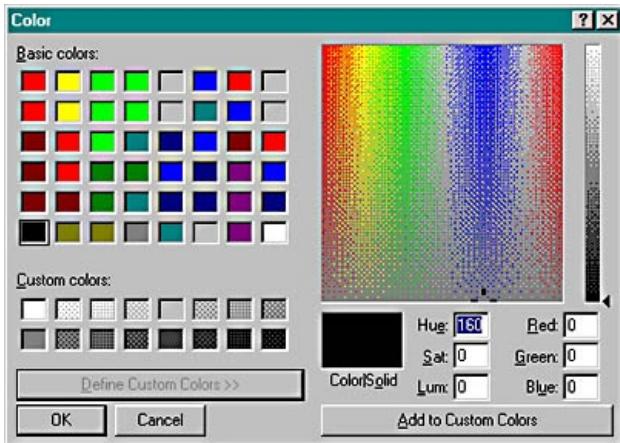
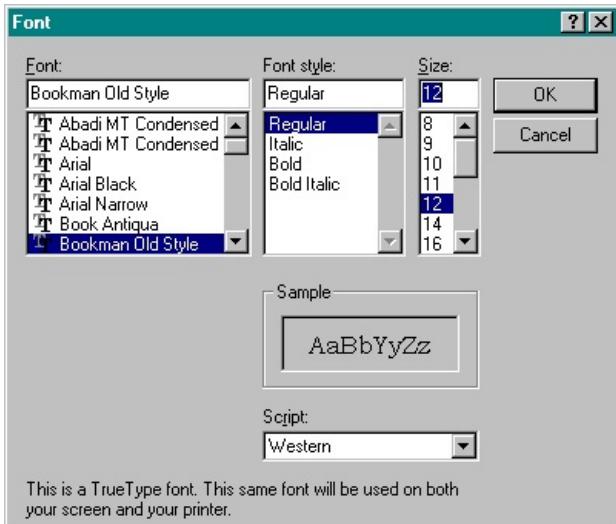
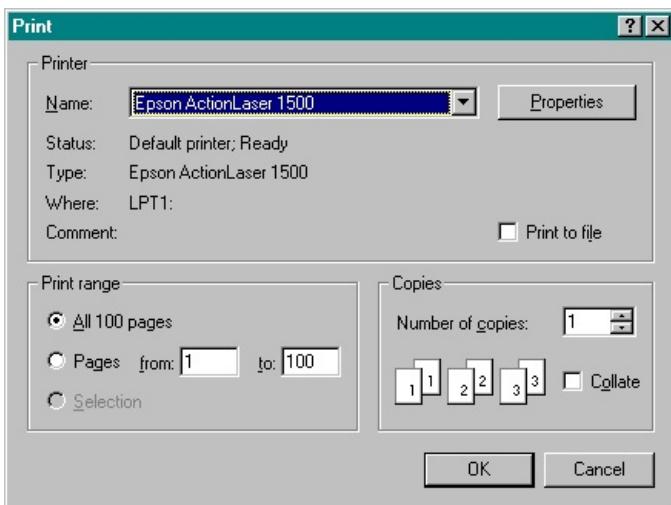


**Figure 34-17.** The Dialogs application in action.



**Figure 34-18.** The CommonDialog control's Open dialog box.



**Figure 34-19.** The CommonDialog control's Save As dialog box.**Figure 34-20.** The CommonDialog control's Color dialog box.**Figure 34-21.** The CommonDialog control's Font dialog box.**Figure 34-22.** The CommonDialog control's Print dialog box.

I like to keep this application handy when I'm working on new applications. When I need to set up a color selection feature in my new application, for example, I simply copy the relevant code from the Dialogs application, add a CommonDialog control, and modify the code for any unique requirements of the new application. Often this method saves me time compared with searching the online help to remember how to set up the CommonDialog control.

## Some Special Features

The Dialogs application contains several unique features that demonstrate some handy techniques.

## About and About2

Two types of About dialog boxes are shown by this program. The Help menu has both an About item and an About2 item. The About dialog box is the one I've used in many of the applications in this book, and the About2 dialog box provides the alternative About dialog box I presented in the VBClock application in Chapter 31, "[Date and Time](#)." Although both forms create similar About dialog boxes in the application, they are created using different techniques. I've included both of them here so that you can make a direct comparison of the two techniques.

## The Sunset Background

There are a few other interesting twists to this application. By a simple modification of the blue-to-black fade algorithm presented in Chapter 14, "[Graphics Techniques](#)," this form's background fades from red to yellow, like a colorful sunset. It's easy to tweak this code to fade from any color to another color that you specify. A more subtle color gradation would probably be more appropriate for many forms, but I liked the bright colors for this demonstration application.

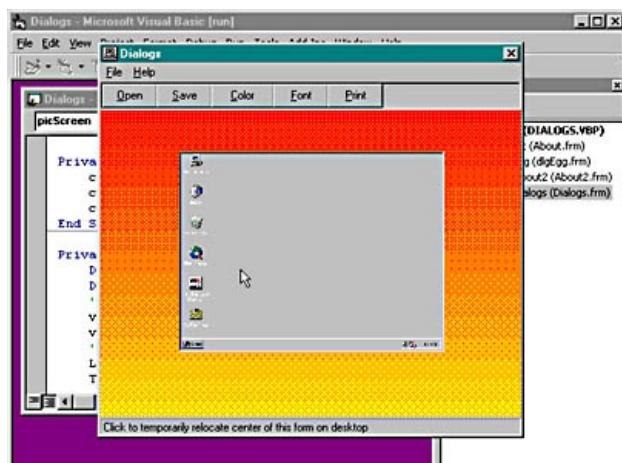
## The Hidden Message

I've also added a hidden message to this application, along the lines of the Easter egg presented in Chapter 18, "[Security](#)." Hidden messages, such as author credits, can be activated in a nearly infinite number of ways. In this application, I keep track of the locations of the last four mouse clicks on the main form. When the correct sequence of clicks occurs near each of the corners of the picture box in the middle of the form, a hidden message window pops up for 5 seconds. Try clicking just outside the picture box, near the upper-left corner, the upper-right corner, the lower-right corner, and finally the lower-left corner. If the order and locations of these four clicks are correct, you'll see a small form with a bright yellow message.

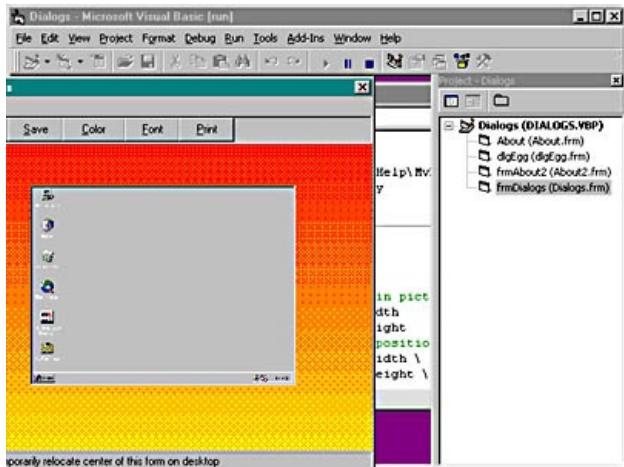
## Form Position

Throughout this book, I've centered most forms on the screen during each form's Load event procedure. A slight modification of this technique allows you to position a form at any location on the screen. To see how this works, click anywhere on the screen graphic in the center of the Dialogs application's main form. If you click one-quarter of the way across the image of the screen and three-quarters of the way down from its top, the entire form will jump to the same relative position on the real screen. After a 2-second delay, the form shifts to the center of the screen again, so you can experiment further. Take a look at the picScreen\_Click event procedure in the source code to see how the form's center is moved temporarily to the relative position indicated by the mouse click in the picture box.

Figure 34-23 shows the mouse at roughly the position described above, and Figure 34-24 shows the form's new, temporary location on the screen.



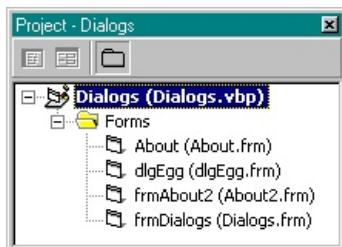
**Figure 34-23.** Clicking in the picture box to cause the application to temporarily relocate.



**Figure 34-24.** The entire application temporarily located in the indicated position on the screen.

## The Application Files

The Dialogs project contains four files. In addition to the main Dialogs form, two types of About dialog forms and a special hidden messages form are part of the project. Figure 34-25 shows the project list.

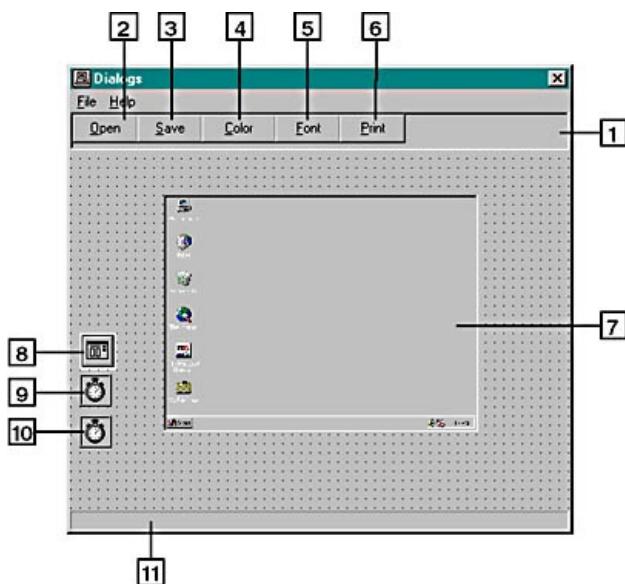


**Figure 34-25.** The Dialogs application project list.

## DIALOGS.FRM

The Dialogs form displays what looks like the Windows 95 desktop within a form. Actually, a picture box in the center of the form displays a reduced image of the Windows 95 desktop. (As described previously, a click anywhere on this image causes the form to temporarily jump to the same relative position on the real desktop.)

Figure 34-26 shows the Dialogs form during development.



**Figure 34-26.** The Dialogs form during development.

To create this form, use the tables and source code in this chapter to add the appropriate controls, set

any nondefault properties as indicated, and enter the source code lines as shown.

### DIALOGS.FRM Menu Design Window Entries

| Caption                                   | Name         | Indentation | Enabled |
|-------------------------------------------|--------------|-------------|---------|
| &File                                     | mnuFile      | 0           | True    |
| &New                                      | mnuNew       | 1           | False   |
| &OpenDear John, How Do I...               | mnuOpen      | 1           | False   |
| &Save                                     | mnuSave      | 1           | False   |
| Save &AsDear John, How Do I...            | mnuSaveAs    | 1           | False   |
| -                                         | mnuFileDash1 | 1           | True    |
| E&xit                                     | mnuExit      | 1           | True    |
| &Help                                     | mnuHelp      | 0           | True    |
| &Contents                                 | mnuContents  | 1           | True    |
| &Search For Help OnDear John, How Do I... | mnuSearch    | 1           | True    |
| -                                         | mnuHelpDash1 | 1           | True    |
| &AboutDear John, How Do I...              | mnuAbout     | 1           | True    |
| About&2Dear John, How Do I...             | mnuAbout2    | 1           | True    |

### DIALOGS.FRM Objects and Property Settings

| ID No.*              | Property    | Value            |
|----------------------|-------------|------------------|
| <b>Form</b>          |             |                  |
|                      | Name        | frmDialogs       |
|                      | Caption     | Dialogs          |
|                      | BorderStyle | 3 - Fixed Dialog |
|                      | Icon        | Pc01.ico         |
| <b>PictureBox</b>    |             |                  |
| 1                    | Name        | picTop           |
|                      | Align       | 1 - Align Top    |
| <b>CommandButton</b> |             |                  |
| 2                    | Name        | cmdOpen          |
|                      | Caption     | &Open            |
| <b>CommandButton</b> |             |                  |
| 3                    | Name        | cmdSave          |
|                      | Caption     | &Save            |
| <b>CommandButton</b> |             |                  |
| 4                    | Name        | cmdColor         |
|                      | Caption     | &Color           |
| <b>CommandButton</b> |             |                  |

|                      |          |                          |
|----------------------|----------|--------------------------|
| 5                    | Name     | <i>cmdFont</i>           |
|                      | Caption  | <i>&amp;Font</i>         |
| <b>CommandButton</b> |          |                          |
| 6                    | Name     | <i>cmdPrint</i>          |
|                      | Caption  | <i>&amp;Print</i>        |
| <b>PictureBox</b>    |          |                          |
| 7                    | Name     | <i>picScreen</i>         |
|                      | AutoSize | <i>True</i>              |
|                      | Picture  | <i>DESKTOP.BMP</i>       |
| <b>CommonDialog</b>  |          |                          |
| 8                    | Name     | <i>cdlOne</i>            |
| <b>Timer</b>         |          |                          |
| 9                    | Name     | <i>tmrClock</i>          |
|                      | Interval | <i>100</i>               |
| <b>Timer</b>         |          |                          |
| 10                   | Name     | <i>tmrRelocate</i>       |
|                      | Enabled  | <i>False</i>             |
|                      | Interval | <i>2000</i>              |
| <b>StatusBar</b>     |          |                          |
| 11                   | Name     | <i>stbBottom</i>         |
|                      | Align    | <i>2 - vbAlignBottom</i> |
|                      | Style    | <i>1 _ sbrSimple</i>     |

### Source Code for DIALOGS.FRM

```

Option Explicit

Dim mvntX, mvntY
Dim mvntLastSec
Dim mvntEggX(1 To 4)
Dim mvntEggY(1 To 4)

Private Sub Form_Click()
    Dim intI As Integer
    'Keep track of last four clicks on form
    For intI = 1 To 3
        mvntEggX(intI) = mvntEggX(intI + 1)
        mvntEggY(intI) = mvntEggY(intI + 1)
    Next intI
    mvntEggX(4) = mvntX
    mvntEggY(4) = mvntY
    'Check for correct sequence and position
    If Abs(mvntEggX(1) - 70) < 30 And _
        Abs(mvntEggY(1) - 60) < 30 And _
        Abs(mvntEggX(2) - 360) < 30 And _
        Abs(mvntEggY(2) - 60) < 30 And _
        Abs(mvntEggX(3) - 360) < 30 And _

```

## Microsoft® Visual Basic® 6.0 Developer's Workshop

```
Abs(mvntEggY(3) - 290) < 30 And _
Abs(mvntEggX(4) - 70) < 30 And _
Abs(mvntEggY(4) - 290) < 30 Then
    'Display hidden message
    dlgEgg.Show vbModal
End If
End Sub

Private Sub Form_Load()
    'Center this form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
    'Adjust button bar height
    picTop.Height = cmdOpen.Height +
        (picTop.Height - picTop.ScaleHeight)
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    'Signal timer to update status bar
    mvntLastSec = -1
    'Keep track of mouse location
    mvntX = X

    mvntY = Y
End Sub

Private Sub Form_Paint()
    Dim lngN As Long
    With Me
        .ScaleMode = vbPixels
        .DrawStyle = 5 'Transparent
        .DrawWidth = 1
    End With
    'Draw sunset background (fade from red to yellow)
    For lngN = 0 To ScaleHeight Step ScaleHeight \ 16
        Line (-1, lngN - 1) -
        (ScaleWidth, lngN + ScaleHeight \ 16), _
        RGB(255, lngN * 255 \ ScaleHeight, 0), BF
    Next lngN
End Sub

Private Sub mnuAbout_Click()
    'Set properties for the About dialog
    About.Application = "Dialogs"
    About.Heading =
        "Microsoft Visual Basic 6.0 Developer's Workshop"
    About.Copyright = "1998 John Clark Craig and Jeff Webb"
    About.Display
End Sub

Private Sub mnuAbout2_Click()
    'Display the About2 dialog
    frmAbout2.Display
End Sub

Private Sub mnuExit_Click()
    Unload Me
End Sub

Private Sub mnuContents_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpContents
    cdlOne.ShowHelp
```

```

End Sub

Private Sub mnuSearch_Click()
    cdlOne.HelpFile = App.Path & "\..\..\Help\Mvbdw.hlp"
    cdlOne.HelpCommand = cdlHelpPartialKey
    cdlOne.ShowHelp
End Sub

Private Sub picScreen_Click()
    Dim vntXpct
    Dim vntYpct
    'Determine mouse's relative position in picture
    vntXpct = 100 * mvntX \ picScreen.ScaleWidth
    vntYpct = 100 * mvntY \ picScreen.ScaleHeight
    'Move form's center to same relative position on screen
    Me.Left = Screen.Width * vntXpct \ 100 - Me.Width \ 2
    Me.Top = Screen.Height * vntYpct \ 100 - Me.Height \ 2
    'Set timer to move form back later
    tmrRelocate.Enabled = True
End Sub

Private Sub picScreen_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    'Keep track of mouse location
    mvntX = X
    mvntY = Y
    'Update status message at bottom of form
    stbBottom.SimpleText = "Click to temporarily relocate " & _
        "center of this form on desktop"
    'Signal timer not to display date and time in status bar
    mvntLastSec = -2
End Sub

Private Sub cmdColor_Click()
    'Set flags for Color dialog box
    cdlOne.Flags = cdlCCRGBInit
    'Show Color dialog box
    cdlOne.ShowColor
    'Display selected color value
    MsgBox "&H" & Hex$(cdlOne.Color), , _
        "Selected colorDear John, How Do I... "
End Sub

Private Sub cmdFont_Click()
    Dim strTab2 As String
    strTab2 = vbTab & vbTab
    'Set flags for Font dialog box
    cdlOne.Flags = cdlCFWYSIWYG + cdlCFBoth + cdlCFScalableOnly
    'Show Font dialog box
    cdlOne.ShowFont
    'Display selected font values
    MsgBox _
        "Font Name:" & vbTab & cdlOne.FontName & vbCrLf & _
        "Font Size:" & strTab2 & cdlOne.FontSize & vbCrLf & _
        "Bold:" & strTab2 & cdlOne.FontBold & vbCrLf & _
        "Italic:" & strTab2 & cdlOne.FontItalic, , _
        "Selected fontDear John, How Do I... "
End Sub

Private Sub cmdOpen_Click()
    'Set up sample filter for Open dialog box
    Dim strBat As String
    Dim strTxt As String

```

```

Dim strAll As String
strBat = "Batch Files (*.bat)|*.bat"
strTxt = "Text Files (*.txt)|*.txt"
strAll = "All Files (*.*)|*.*"
cdlOne.Filter = strBat & "|" & strTxt & "|" & strAll
`Set default filter to third one listed
cdlOne.FilterIndex = 3
`Hide "ReadOnly" check box
cdlOne.Flags = cdlOFNHideReadOnly
`Deselect previously selected file, if any
cdlOne.FileName = ""
`Show Open dialog box
cdlOne.ShowOpen
`Display selected filename
If cdlOne.FileName = "" Then Exit Sub
MsgBox cdlOne.FileName, , "Selected fileDear John, How Do I... "
End Sub

Private Sub cmdPrint_Click()
    Dim strPrintToFile As String
    `Set flags for Print dialog box
    cdlOne.Flags = cdlPDAllPages + cdlPDNoSelection
    `Set imaginary page range
    cdlOne.Min = 1
    cdlOne.Max = 100
    cdlOne.FromPage = 1
    cdlOne.ToPage = 100
    `Show Print dialog box
    cdlOne.ShowPrinter
    `Extract some printer data
    If cdlOne.Flags And cdlPDPrintToFile Then
        strPrintToFile = "Yes"
    Else
        strPrintToFile = "No"
    End If
    `Display selected print values
    MsgBox _
        "Begin Page:" & vbTab & cdlOne.FromPage & vbCrLf & _
        "End Page:" & vbTab & cdlOne.ToPage & vbCrLf & _
        "No. Copies:" & vbTab & cdlOne.Copies & vbCrLf & _
        "Print to File:" & vbTab & strPrintToFile
        , , "Selected print informationDear John, How Do I... "
End Sub

Private Sub cmdSave_Click()
    `Set up filter for Save As dialog box
    Dim strBat As String
    Dim strTxt As String
    Dim strAll As String
    strBat = "Batch Files (*.bat)|*.bat"
    strTxt = "Text Files (*.txt)|*.txt"
    strAll = "All Files (*.*)|*.*"
    cdlOne.Filter = strBat & "|" & strTxt & "|" & strAll
    `Set default filter to third one listed
    cdlOne.FilterIndex = 3
    `Hide ReadOnly check box
    cdlOne.Flags = cdlOFNHideReadOnly
    `Deselect previously selected file, if any
    cdlOne.FileName = ""
    `Show the Save As dialog box
    cdlOne.ShowSave
    `Display the selected file
    If cdlOne.FileName = "" Then Exit Sub

```

```

    MsgBox cdlOne.FileName, , "Save As' fileDear John, How Do I... "
End Sub

Private Sub tmrRelocate_Timer()
    'Relocate form once per move
    tmrRelocate.Enabled = False
    'Center this form
    Me.Left = (Screen.Width - Me.Width) \ 2
    Me.Top = (Screen.Height - Me.Height) \ 2
End Sub

Private Sub tmrClock_Timer()
    Dim vntSec
    vntSec = Second(Now)
    If vntSec = mvntLastSec Then Exit Sub
    If mvntLastSec = -2 Then Exit Sub
    mvntLastSec = vntSec
    'Update date and time in status line
    stbBottom.SimpleText = Format(Date, "Long Date") & _
        Space$(5) & Format(Time, "hh:mm AMPM")
End Sub

```

### DLGEGG.FRM

DLGEGG.FRM is a simple form that displays a secret message when the user clicks on the specified locations in the correct sequence. A timer causes the form to unload itself after a 5-second delay, although you could easily modify this form to unload when the user clicks anywhere on the form. Feel free to change the message or enhance the form as you want. Figure 34-27 shows the *dlgEgg* form during development.



**Figure 34-27.** The *dlgEgg* form during development.

To create this form, use the following table and source code to add the appropriate controls, set any nondefault properties as indicated, and enter the source code lines as shown.

### DLGEGG.FRM Objects and Property Settings

| Property    | Value                      |
|-------------|----------------------------|
| <b>Form</b> |                            |
| Name        | <i>dlgEgg</i>              |
| Caption     | <i>dlgEgg</i>              |
| BackColor   | <i>&amp;H0000FFFF&amp;</i> |
| BorderStyle | <i>1 - Fixed Single</i>    |
| ControlBox  | <i>False</i>               |
| MaxButton   | <i>False</i>               |
| MinButton   | <i>False</i>               |
| WindowState | <i>0 - Normal</i>          |

**Label**

|           |                                                                        |
|-----------|------------------------------------------------------------------------|
| Name      | <i>IblEgg</i>                                                          |
| Alignment | <i>2 - Center</i>                                                      |
| Caption   | <i>This "Easter egg" (hidden message) will disappear in 5 seconds.</i> |
| Font      | <i>Arial - Italic - 14</i>                                             |
| BackColor | <i>&amp;H0000FFFF&amp;</i>                                             |

**Timer**

|          |                |
|----------|----------------|
| Name     | <i>tmrQuit</i> |
| Enabled  | <i>True</i>    |
| Interval | <i>5000</i>    |

**Source Code for DLGEGG.FRM**

```
Option Explicit

Private Sub tmrQuit_Timer()
    Unload Me
End Sub
```

## About This Electronic Book

This electronic book was originally created—and still may be purchased—as a print book. For simplicity, the electronic version of this book has been modified as little as possible from its original form. For instance, there may be occasional references to sample files that come with the book. These files are available with the print version, but are not provided in this electronic edition.

## “Expanding” graphics

Many of the graphics shown in this book are quite large. To improve the readability of the book, reduced versions of these graphics are shown in the text. To see a full-size version, click on the reduced graphic.



### Your Information Source

We're the independent publishing division of Microsoft Corporation—your source of inside information and unique perspectives about Microsoft products and related technologies. We've been around since 1984, and we offer a complete line of computer books—from self-paced tutorials for first-time computer users to advanced technical references for professional programmers.

[mspress.microsoft.com](http://mspress.microsoft.com)