Programmer's Guide

Microsoft® Visual Basic™

Programmer's Guide

Microsoft® VISUAL BASIC™

Programming System for Windows™

Version 3.0

Programmer's Guide

Microsoft® Visual Basic™

Programming System for Windows™
Version 3.0

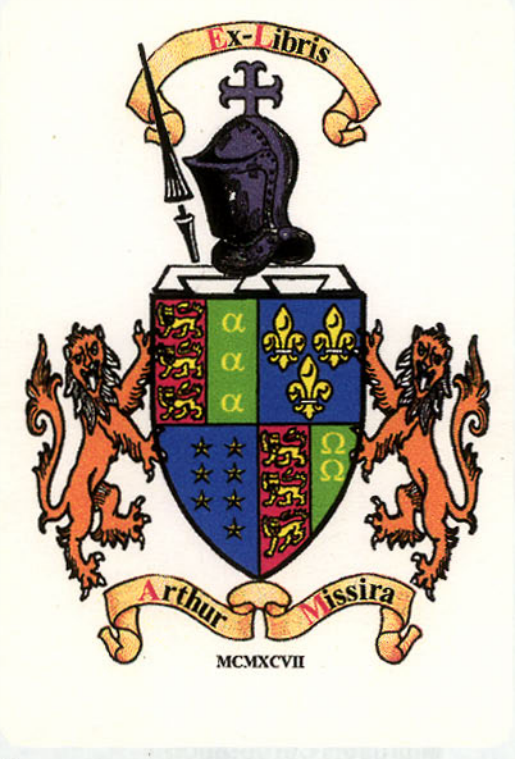
Microsoft Corporation

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

© 1993 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, and FoxPro are registered trademarks and Microsoft Access, QuickBasic, Visual Basic, and Windows are trademarks of Microsoft Corporation in the USA and other countries.
Btrieve is a registered trademark of Novell, Inc.
CompuServe is a registered trademark of CompuServe, Inc.
dBASE, dBASE III, and dBASE IV are registered trademarks of Borland International, Inc.
GEnie is a trademark of General Electric Corporation.
Hercules is a registered trademark of Hercules Computer Technology, Inc.
IBM is a registered trademark of International Business Machines Corporation.
Oracle is a registered trademark of Oracle Corporation.
Paradox is a registered trademark of Ansa Software, a Borland company.
PC Paintbrush is a registered trademark of ZSoft Corporation.
TrueType is a registered trademark of Apple Computer, Inc.

Portions of Chapter 22, "Object Linking and Embedding (OLE)," are reprinted with permission from the article "Front-Ends with Win 3.1's OLE," which was first published in *BASICPro* magazine Vol. 2 No. 3 May/June 1992.



Document Number DB51410-0993
Printed in Ireland: 03

Contents

Document Conventions ix

Chapter 1 Introduction 1
Setting Up 2
Getting Started 4
Using Online Documentation 8
New Features in Visual Basic 3.0 13
Microsoft Product Support Services 14

Chapter 2 Your First Visual Basic Application 15
Starting Visual Basic 16
Steps to Creating a Visual Basic Application 18
Working with Sample Applications 23
Saving, Debugging, and Distributing Your Applications 30

Chapter 3 Creating and Using Controls 31
Visual Basic Controls 32
Clicking Buttons to Perform Actions 36
Displaying and Entering Text 38
Controls That Present Choices to the User 43
Getting and Setting the Focus of an Object 64
Setting the Tab Order 65
Enabling, Disabling, and Controlling Visibility at Run Time 67
Working with Control Arrays 68

Chapter 4 Menus and Dialogs 75
Creating Menus at Design Time 76
Controlling Menus at Run Time 86
Working with Dialog Boxes 93

Chapter 5 Managing Projects 111
Understanding Projects 111
Creating, Opening, and Saving Projects 115
Creating New Forms and Code Modules 115
Adding, Removing, and Saving Files 116
Making and Running an Executable File 119
Setting Environment Options 119

Editing the AUTOLOAD.MAK File 121

Chapter 6 Programming Fundamentals 123
Structure of a Visual Basic Application 123
Procedures 130
Code in Procedures 134
Controlling Execution 140

Chapter 7 Variables, Constants, and Data Types 149
Declaring Variables 150
Scope and Lifetime of Variables 152
Fundamental Variable Data Types 157
Arrays 170
User-Defined Types (Structures) 176
Symbolic Constants 178

Chapter 8 Objects and Instances 181
An Introduction to Object Variables 181
Declaring an Object Variable 183
Using Object Variables 192
Multiple Instances and Me 201
The Forms and Controls Collections 203
System Objects 206

Chapter 9 Debugging 207
Approaches to Debugging 208
Design Time, Run Time, and Break Mode 211
Using Break Mode 213
Running Selected Portions of Your Application 218
Using the Calls Dialog 220
Using the Debug Window 222
Monitoring Data with Watch Expressions 223
Testing Data and Procedures with the Immediate Pane 227
Special Considerations in Debugging Events 231
Testing and Using Command-Line Arguments 232
Avoiding Bugs 233

Chapter 10 Handling Run-Time Errors 235
Why Use Error Handling? 236
How to Handle Errors 238
Unanticipated Errors 243

Generating Your Own Errors 246
Centralized Error Handling 247
Delayed Error Handling 251
Turning Off Error Handling 253

Chapter 11 Optimizing Your Application for Size and Speed 255
Speeding Calculations and Code 256
Speeding Display 257
Speeding the Start of Your Application 259
Minimizing Total Application Size in Memory 259
Minimizing Memory Required by Graphics 261
Minimizing Stack Use 263

Chapter 12 Responding to Mouse Events 265
Mouse Events 266
Positioning a Control with the Mouse 267
Graphical Mouse Applications 267
The Button and Shift Arguments 271
Dragging and Dropping 277

Chapter 13 Using the Grid Control 285
The Grid Control 285
Creating and Sizing Rows and Columns 286
Using Text in the Grid Control 288
Using Graphics in the Grid Control 291
Selecting and Highlighting Cells 293
Adding and Removing Rows 296

Chapter 14 Multiple-Document Interface (MDI) Applications 297
Using MDI in Visual Basic 298
Creating an MDI Application 298
MDI NotePad — A Document-Centered Application 301
Menus in MDI Applications 308
Creating a Toolbar 313

Chapter 15 Creating Graphics for Applications 317
The Fundamentals of Graphics 317
Using Graphical Controls 321
Using Graphics Methods 339
Using Graphics Properties 358

Chapter 16 Displaying and Printing Information 377
The Fundamentals of Fonts 378
Setting Font Characteristics 379
Displaying Text in Forms and Picture Boxes 383
Displaying Tabular Data 387
Formatting Numbers, Dates, and Times 389
The Fundamentals of Printing 392
Printing with the Printer Object 394
Printing with the PrintForm Method 403

Chapter 17 Interacting with the Environment 405
Working with Selected Text 405
Using the Clipboard Object 406
Using Timer Controls 411
Writing Idle Loops with DoEvents 416
Writing Low-Level Keyboard Handlers 419

Chapter 18 Using the File-System Controls 425
Examining the File System 425
A File Seeker Application 431

Chapter 19 Processing Files 435
Overview of File Access Types 435
Random Access 437
Sequential Access 442
Binary Access 446

Chapter 20 Accessing Databases with the Data Control 453
Getting a Quick Start 454
Using the Data Control 457
Opening a Database 458
Working with Bound Controls 462
Manipulating Records with Code 467
Controlling Transactions, Validation, and Data Updates 476
Working with Database Design 481
Working with Database Structure 483

Chapter 21 Communicating with Other Applications 493
What Is Dynamic Data Exchange? 494
Creating DDE Links at Design Time 497
Link Properties 499

Link Events ... 505
Allowing Users to Create Their Own Automatic Links ... 509
Using Methods to Perform DDE Operations ... 513
Handling Errors ... 517
Sending Keystrokes to Other Applications ... 521

Chapter 22 Object Linking and Embedding (OLE) ... 523
Creating Front-End Applications ... 523
OLE Fundamentals ... 524
Using the OLE Control ... 529
Working with Files ... 541
Comparing OLE and DDE ... 543

Chapter 23 Programming Other Applications' Objects ... 547
What Is OLE Automation? ... 548
Accessing OLE Objects ... 548
Creating Invisible Objects ... 550
Using Existing Objects ... 550

Chapter 24 Calling Procedures in DLLs ... 557
Overview ... 558
Declaring a DLL Procedure ... 558
Calling a DLL Procedure ... 559
Special Considerations When Declaring DLL Procedures ... 559
Calling DLL Procedures with Specific Data Types ... 563
Converting Common Declarations ... 570

Chapter 25 Distributing Your Applications ... 573
The Visual Basic SetupWizard ... 573
How to Use the Visual Basic Setup Toolkit ... 577
How to Use the Setup Toolkit ... 578

Appendix A ASCII File Formats ... 591
ASCII Forms Format ... 591
Loading and Saving ASCII Forms ... 602
Project File (.MAK) Format ... 606

Appendix B Icon Library ... 609
Arrows and Pointers ... 610
Communication ... 611
Computers ... 612
Drag-and-Drop ... 613

Elements 614
Flags 614
Industry and Transportation 615
Mail 615
Miscellaneous 616
Office 618
Traffic Signs 619
Writing 620

Appendix C Compatibility with Other Versions 623
Visual Basic 1.0 for Windows 624
Visual Basic 2.0 for Windows 625
Visual Basic for MS-DOS 629

Appendix D Specifications and Limitations 639
Hardware and Software Requirements for Visual Basic Applications 640
Application Limitations 640
Form Limitations 641
Control Limitations 642
Code Limitations 642
Windows Limitations 647

Appendix E Microsoft Product Support Services 649
Product Support Within the United States 649
Support Services Worldwide 651

Appendix F Accessibility for People with Disabilities 659
Microsoft Support Services for People Who Are Deaf or Hard-of-Hearing . . . 660
Access Pack for Microsoft Windows 660
Keyboard Layouts for Single-Handed Users 661
Documentation on Audio Cassettes and Floppy Disks 661
Products for People Who Are Blind or Have Low Vision 662
Customizing Windows 662
Getting More Information 662

Index 665

Document Conventions

This manual uses the following typographic conventions.

Example of convention	Description
Sub, If, ChDir, Print, True, Dynaset	Words in bold with initial letter capitalized indicate language-specific reserved words.
BackColor, Click, Debug	Names of properties, events, and special objects appear with initial letter capitalized.
setup	Words you're instructed to type appear in bold.
event-driven	In text, italic letters indicate defined terms, usually the first time they occur in the book. Italics also are used occasionally for emphasis.
eventname	In syntax, italic letters indicate placeholders for information you supply.
[*expressionlist*]	In syntax, items inside square brackets are optional.
{**While**|**Until**}	In syntax, braces and a vertical bar indicate a choice between two or more items. You must choose one of the items unless all of the items also are enclosed in square brackets.
`Syntax error`	This font is used for user-defined variables and error messages.

`Sub HelloButton_Click ()`
`  Readout.Text = "Hello, world!"`
`End Sub`
	This font is used for code.

`Readout.Text = "This should all`
`➥ be on one line in the Code`
`➥ window."`
	The line-continuation character (➥) indicates that code continued from one line to the next in the book should be typed all on one line in the Code window.

CONSTANT.TXT	Words in all capital letters indicate file names.
ENTER	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R. (Key names in online Help have only the initial letter capitalized.)
ALT+F1	A plus sign (+) between key names indicates a combination of keys. For example, ALT+F1 means to hold down the ALT key while pressing the F1 key.

Example of convention	Description
DOWN ARROW	Individual direction keys are referred to by the direction of the arrow on the key top (LEFT, RIGHT, UP, or DOWN). The phrase "arrow keys" is used when describing these keys collectively.
BACKSPACE, HOME	Other navigational keys are referred to by their specific names.
Multiple-document interface (MDI)	Acronyms are usually spelled out the first time they are used.

Programming Style in This Manual

The following guidelines are used in writing programs in this manual. For more information, see Chapter 6, "Programming Fundamentals."

- Keywords appear with initial letters capitalized, while constants appear in all capital letters:

```
' Sub, If, ChDir, Print, and True are keywords.
Print "Title Page"

' Constants used with the ZOrder method.
Global Const BRINGTOFRONT = 0
Global Const SENDTOBACK = 1
```

- Line labels are used instead of line numbers. The use of line labels is restricted to error-handling routines:

```
OpenError
If Err = 71 Then
    Print Error
    Resume
Else
    End
End If
```

- An apostrophe (') introduces comments:

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in **Sub** and **Function** procedures are indented from the enclosing code:

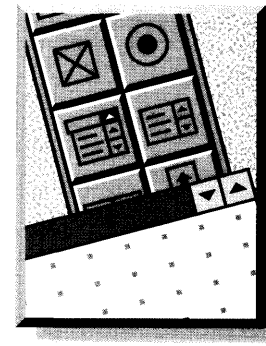
```
Sub cmdRemove_Click ()
    Dim Ind As Integer

    Ind = lstClient.ListIndex       ' Get index.
    If Ind >= 0 Then            ' Make sure list item is selected.
        lstClient.RemoveItem Ind    ' Remove it from list box.
        lblDisplay.Caption = lstClient.ListCount     ' Display number.
    Else
        Beep                        ' If nothing selected, beep.
    End If
End Sub
```

- As noted earlier, lines too long to fit on one line in this manual may be continued on the next line using a line-continuation character (➥).

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
```

CHAPTER 1

Introduction



Welcome to Microsoft® Visual Basic™—the quickest and easiest way to create applications for the Microsoft® Windows™ operating system. The Visual Basic programming system allows you to create attractive and useful applications that fully exploit the graphical user interface (GUI).

Visual Basic makes you more productive by providing appropriate tools for the different aspects of GUI development. You create the graphical user interface for your application by drawing objects in a graphical way. You set properties on these objects to refine their appearance and behavior. Then you make this interface react to the user by writing code that responds to events that occur in the interface.

Using Visual Basic, you can create powerful, full-featured applications that exploit the key features of Microsoft Windows, including multiple-document interface (MDI), object linking and embedding (OLE), dynamic data exchange (DDE), graphics, and more. And Visual Basic can be extended by adding custom controls and by calling procedures in dynamic-link libraries (DLLs). Your finished application is a true .EXE file that uses a run-time DLL you can freely distribute.

This chapter shows you how to set up Visual Basic on your computer and introduces the other parts of the documentation. If you are new to Visual Basic, you should read through this chapter completely. If you have used a previous version of Visual Basic, you can skim this chapter; you may be particularly interested in "New Features in Visual Basic 3.0" at the end of the chapter.

Contents

- Setting Up
- Getting Started
- Using Online Documentation
- New Features in Visual Basic 3.0
- Microsoft Product Support Services

Setting Up

You install Visual Basic on your computer using the program SETUP.EXE. The Setup program installs Visual Basic itself, the Help system, sample applications, the Icon Library, and other product components from the distribution disks to your hard disk.

Important You cannot simply copy files from the distribution disks to your hard disk and run Visual Basic. You must use the Setup program, which decompresses and installs the files in the appropriate directories.

Before You Run Setup

Before you install Visual Basic, make sure that your computer meets the minimum requirements and that your Visual Basic package contains the required items.

Check the Hardware and System Requirements

To run Visual Basic, you must have certain hardware and software installed on your computer. The system requirements include:

- Any IBM®-compatible machine with an 80286 processor or higher.
- A hard disk.
- A 5 1/4" or 3 1/2" floppy drive.
- An EGA, VGA, 8514, Hercules®, or compatible display.
- One megabyte of memory.
- A mouse.
- Microsoft® MS-DOS® version 3.1 or later.
- Windows version 3.0 or later in standard or enhanced mode.

Check the Visual Basic Package

These items should be in the Standard Edition of the Visual Basic package:

- Registration card
- *Programmer's Guide*
- *Language Reference*
- Disks

Note that the disk labels indicate how many disks should be present. Also, a file called PACKING.LST lists all the files that should be on the Visual Basic disks. To read the file, double-click PACKING.LST in the File Manager, or use the **Type** command in MS-DOS.

The Professional Edition of Visual Basic also includes the following print documentation:

Book 1

- *Custom Control Reference*
- *Custom Development Guide*
- *Help Compiler Guide*

Book 2

- *Data Access Guide*
- *Crystal Reports for Visual Basic User's Guide*

If any pieces are missing, contact the retailer from whom you bought Visual Basic.

Make Backup Copies of the Distribution Disks

Before you run Setup, make backup copies of the Visual Basic disks using one of the following commands:

- The Copy command on the File menu in the Windows File Manager.
- The Copy Disk command on the Disk menu in the Windows File Manager.
- The **Copy** or **Diskcopy** command in MS-DOS.

Read the README.TXT File

The README.TXT file lists any changes to the Visual Basic documentation since its publication. To read the file, double-click README.TXT in the File Manager, or use the **Type** command in MS-DOS. Check the first section of the file for any new information about installing Visual Basic.

Running Setup

When you run the Setup program, you'll set a path for Visual Basic and then select the Visual Basic files you want to install.

▶ **To start Setup**

1. Insert Disk 1 in drive A.
2. From the File menu of the Program Manager or File Manager, choose Run.
3. Type **a: setup**
4. Follow the Setup instructions on the screen.

Changing Options After Quitting Setup

You can run Setup as many times as necessary. For example, you can run Setup to reinstall Visual Basic in another directory or to install other portions of Visual Basic.

Once you have completed the Setup procedure, you can start Visual Basic by double-clicking the Visual Basic icon in the new program group. For more information, see Chapter 2, "Your First Visual Basic Application."

Getting Started

The Visual Basic documentation set (this manual, the *Language Reference*, and the online documentation) assumes you know how to use a mouse, open a menu, and choose menu and dialog options. To review these techniques, consult the documentation for Windows.

Note If you're experienced with other Basics, you'll easily learn Visual Basic. You can even transfer code from other Basics to Visual Basic after making certain changes. (In some cases, you can transfer large amounts of code directly, but you will need to rewrite interface code.) For more information about transferring code to Visual Basic, see Appendix C, "Compatibility with Other Versions."

Visual Basic Documentation

The Visual Basic package includes several documentation tools, each designed to help you learn and use a particular aspect of the product. The following table lists the documentation supplied with the Standard Edition of Visual Basic.

Documentation	Description
Programmer's Guide	A guide to creating applications with Visual Basic. It explains programming techniques, using frequent examples to show how the techniques work in typical applications.
Language Reference	A summary of the Visual Basic language, listing the methods, properties, and events that apply to each object. It features a detailed A–Z reference for functions, statements, methods, properties, events, and objects.
Learning Microsoft Visual Basic	A set of seven online lessons that walk you through the process of creating, debugging, and customizing an application. You access these lessons by choosing *Learning Microsoft Visual Basic* from the Help menu.

Documentation	Description
Online Help (VB.HLP)	The fundamental online reference for Visual Basic. It contains much of the same information found in the written documentation: language syntax, programming techniques, and control definitions and descriptions.
Sample applications (In the \SAMPLES directory of the main Visual Basic directory)	A set of applications written in Visual Basic that show the programming techniques described in the *Programmer's Guide*.

The following table lists the documentation supplied with the Professional Edition of Visual Basic.

Documentation	Description
Control Development Guide	A guide to creating your own custom controls.
Custom Control Reference	A comprehensive reference documenting the custom controls provided with the Professional Edition.
Data Access Guide	A guide to creating and manipulating databases using Visual Basic.
Help Compiler Guide	A guide to authoring Help files for Microsoft Windows-based applications.
Crystal Reports for Visual Basic User's Guide	A guide to the features of Crystal Reports.
Visual Design Guide	A set of online lessons that teach the fundamentals of good user-interface design.
Online Help (VB.HLP)	The fundamental online reference for Visual Basic. It contains much of the same information found in the written documentation: language syntax, programming techniques, and control definitions and descriptions.
Online Help (CTRLREF.HLP)	The online reference for the custom controls documented in the *Custom Control Reference*. The "See Also" reference lists the object types for each of the supplied controls as well as information on creating, running, and distributing EXE files. This Help file includes information on which DLLs and VBXs should be distributed with applications using various custom controls or language features.
Online Help (DATAMGR.HLP)	Help file documenting the Data Manager tool.
Online Help (HELPREF.HLP)	The online reference for the *Help Compiler Guide*. It contains information on creating, testing, and debugging Help files.

Documentation	Description
Online Help (ICONWRKS.HLP)	A sample Help file documenting the IconWorks sample application. Examples of the source code for this Help file are referenced throughout the *Help Compiler Guide*.
Online Help (VBKNOWLG.HLP)	A Help file containing technical articles written by the Visual Basic Product Support team. This Help file provides tips, critical information, references, and other information that may not be addressed in the manuals. It includes articles covering advanced programming tips, companion products, custom controls, designs and limits, and common problems.
Online Help (WIN31API.HLP)	A Help file containing declarations, type structures, and constants used with the Windows 3.1 API. You can copy and paste these declarations and constants into your Visual Basic application.
Online Help (WIN31WH.HLP)	A Help file documenting the API functions and other common DLLs available under Microsoft Windows 3.1. The "Function Groups" section provides a grouped list of different API functions that can help you find a function when you don't know its name.
Online Help (SETUPWIZ.HLP)	A Help file documenting the Visual Basic SetupWizard.
Online Help (CRW.HLP)	A Help file documenting Crystal Reports.

How This Book Is Organized

The chapters of this manual can be grouped into five parts:

- Chapters 1–5 describe the preliminary steps to using this product, including setup, creating Visual Basic interfaces, and managing projects.
- Chapters 6–11 explain the fundamentals of programming, including writing code for an interface, debugging, error handling, and optimizing Visual Basic code.
- Chapters 12–17 explain the methods for writing code to respond to mouse events, using the grid control, using MDI forms, creating graphics, and displaying and printing output.
- Chapters 18–25 explain advanced programming techniques, including handling files and communicating with other applications.
- Appendixes A–F provide reference information about Visual Basic, including an explanation of ASCII file formats, the Icon Library, guidelines for adapting code written in other versions of Microsoft Basic, a description of the specifications and limitations of Visual Basic, information on contacting Microsoft's worldwide product support services, and accessibility information for people with disabilities.

Sample Applications

In addition to the documentation, the Visual Basic package includes sample applications that you can load into Visual Basic. Not only are these applications useful, but they are also excellent learning tools. You can copy any part of them into your own applications, modifying them as necessary.

Note Throughout the documentation, you'll find sample code and applications that illustrate programming techniques. Many of the files for these applications are included on disk. If you installed the sample applications, you will find them in the \SAMPLES subdirectory of the main Visual Basic directory.

In many cases, you'll find that the applications on disk contain more code or features than the sample code discussed in the text. The intent is to provide a shortened version of the code in text to facilitate discussion, yet still provide you with the complete application on disk for your own exploration.

What to Do Next

For the most thorough introduction to Visual Basic:

1. Run *Learning Microsoft Visual Basic* by choosing it from the Help menu. Work through each lesson, or choose the lessons and topics that interest you the most.
2. Read Chapters 1–5 of this manual, preferably at the computer, to understand the fundamentals of working with forms, modules, and controls.
3. Read Chapters 6–11 to learn the most frequently used programming techniques.
4. Refer to the online documentation or the *Language Reference* while you work.

Alternatively, if you are familiar with event-driven programming and with the Windows environment, you might try a faster track:

1. Look at *Learning Microsoft Visual Basic* or Chapter 2 of this manual ("Your First Visual Basic Application") for an overview of the Visual Basic application development process.
2. Skim through Chapters 1–10 of this manual to learn about the most frequently used development techniques.
3. Refer to online Help for information on specific tasks, parts of the programming environment, or keywords in the Visual Basic language.

Using Online Documentation

The online documentation system references nearly all aspects of Visual Basic. It includes *Learning Microsoft Visual Basic*, a set of seven online lessons, and online Help, which contains extensive information on the Visual Basic programming environment and language.

Learning Microsoft Visual Basic

To start the online lessons, choose *Learning Microsoft Visual Basic* from the Help menu (Figure 1.1). For instructions on how to navigate through the lessons, choose the Instructions button from the main menu.

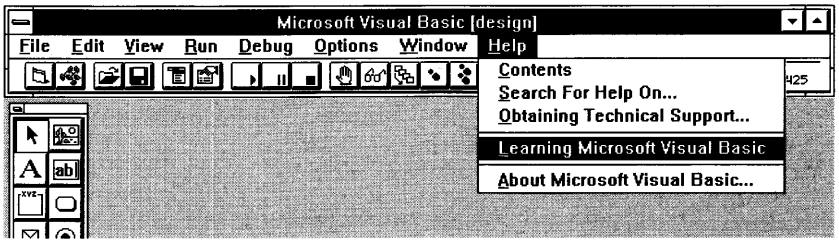


Figure 1.1 The Help menu

The lessons of *Learning Microsoft Visual Basic* are described briefly in the following table.

Lesson	Contents
"How Visual Basic Works"	Introduces the Visual Basic development environment.
"Creating an Application"	Walks you through the process of creating a simple application in Visual Basic.
"Writing Event-Driven Programs"	Introduces event-driven programming and compares it with traditional programming.
"Working with Forms and Controls"	Shows you how to build a form, add controls, and set properties.
"Adding Menus"	Explains how to add menus to your forms.
"Debugging Your Application"	Describes how to write Visual Basic code and use the debugging tools.
"Using Color and Graphics"	Demonstrates design ideas such as the use of graphics, color, and layout.

Online Help

You can access Help through the Help menu Contents command, by searching for specific topics with the Help Search tool, or by pressing F1 to get context-sensitive Help on the Visual Basic programming environment or keywords in the language.

Help Contents

For information on Help topics, choose Contents from the Help menu or press F1 and click the Contents button. You can use the Contents screen (Figure 1.2) to jump to topics that tell you how to use Visual Basic, or to get quick access to key reference topics.

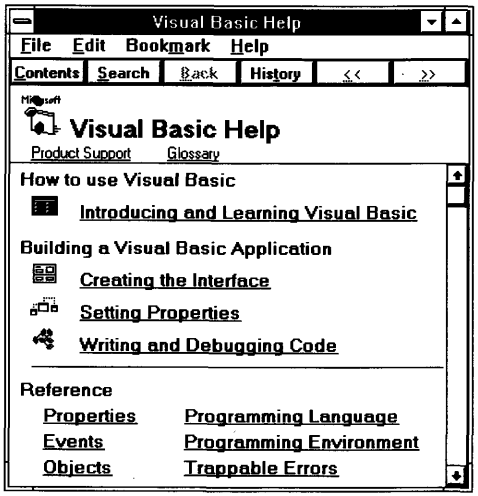


Figure 1.2 The Contents screen

Help Search

The fastest way to find a particular topic in Help is to use the Search dialog box (Figure 1.3). To display the Search dialog box, you can either choose Search from the Help menu or click the Search button on any Help topic screen.

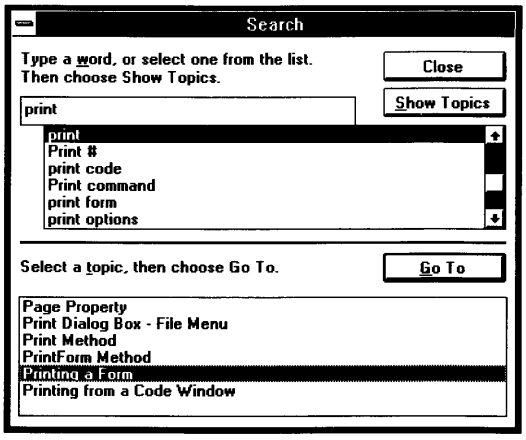


Figure 1.3 The Search dialog box

▶ To search Help

1. From the Help menu, choose Search. (You can also choose the Search button from any Help topic window.)
2. In the Search dialog box, type a word, or select one from the list by scrolling up or down. Press ENTER or choose Show Topics to display a list of topics related to the word you specified.
3. Select a topic name, and then press ENTER or choose Go To to view the topic.

Context-Sensitive Help (F1)

Many parts of Visual Basic are context-sensitive. *Context-sensitive* means you can get Help on these parts directly without having to go through the Help menu. For example, to get Help on any reserved word in the Visual Basic language, place the insertion point on that word in the Code window and press F1.

You can press F1 from any context-sensitive part of the Visual Basic interface to display information about that part. The context-sensitive parts are:

- Every window in Visual Basic (Properties window, Code window, and so on)
- Items on the toolbar
- Controls in the Toolbox
- Objects on a form
- Properties in the Properties window

- Event procedures in the Code window
- Visual Basic reserved words (statements, functions, properties, methods, events, and special objects)
- Error messages

Once you've opened Help, you can press F1 for information about how to use Help itself.

Running Code Examples from Help

Many of the language topics in Help contain code examples that you can run from Visual Basic. The following procedures show you how to copy and run a code example from Help.

Note The following procedure assumes that the code example does not contain global declarations.

▶ **To copy a code example from Help**

1. Create a new form by choosing New Form from the File menu, or use an existing form. (For more information on creating and using forms, see Chapter 2, "Your First Visual Basic Application.")
2. Press F1 to open Help, or choose Contents from the Help menu.
3. In Help, search for *graphics*, and go to the topic called "FillColor Property."
4. In the FillColor Property window, click the Example jump, located in the nonscrolling region near the top of the window. (A *jump* is a word that you can click to jump to another topic. Jumps are underlined, and on color monitors, the jump text is green.)
5. Choose the Copy button in the FillColor Property Example window.

   Help displays a Copy dialog box.
6. Select the subroutine portion of the example.

   Note that the first "Sub" marks the beginning of the subroutine and the last "End Sub" marks the end of the subroutine, as shown in Figure 1.4.

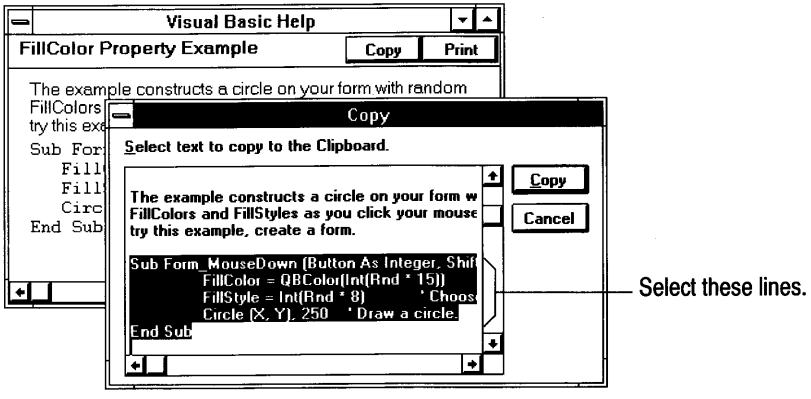


Figure 1.4 Selecting the subroutine portion of the example

7. Choose the Copy button in the Copy dialog box.

   Help copies the text to the Clipboard.

8. Return to the form you created and double-click the form to display the Code window.
9. In the Object box in the Code window, select "(general)."
10. From the Edit menu, choose Paste.

    The example now appears in the Code window.

11. From the Options menu, choose Project.
12. Select the StartUpForm line, then in the Setting box, type or select the name of the form that contains the example code. When the name is selected, choose OK.
13. From the Run menu, choose Start, or press F5.
14. Click the form to run the example code.

Note Some code examples require you to draw controls on the form. For more information on drawing controls, see Chapter 3, "Creating and Using Controls."

New Features in Visual Basic 3.0

The following table lists the new features in the Standard Edition of Visual Basic 3.0 and tells you where to look for more information about them.

New feature	For more information
Data control and standard data-aware (bound) controls	Chapter 20, "Accessing Databases with the Data Control"
Data Manager tool	Online Help (DATAMGR.HLP)
OLE Automation	Chapter 23, "Programming Other Applications' Objects"
OLE control updated to support OLE2 features	Chapter 22, "Object Linking and Embedding (OLE)"
Common dialog custom control	Chapter 4, "Menus and Dialogs"
Pop-up menus	Chapter 4, "Menus and Dialogs"
Automatically save project before running option	Chapter 5, "Managing Projects"
SetupWizard	Chapter 25, "Distributing Your Applications" and Online Help (\VB\SETUPKIT\KITFILES\SETUPWIZ.HLP)

The following table lists additional new features available in the Professional Edition of Visual Basic 3.0.

New feature	For more information
Data access support	*Professional Features*, Book 2
Crystal Reports for Visual Basic	*Professional Features*, Book 2
Updated external ODBC database support	*Professional Features*, Book 2
Outline custom control	*Professional Features*, Book 1
CDK enhancements	*Professional Features*, Book 1
Data-aware custom controls (masked edit, 3D panel, and 3D check box)	*Professional Features*, Book 1

Microsoft Product Support Services

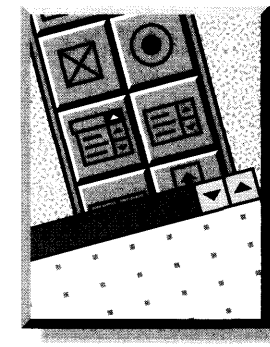
Microsoft offers a variety of support options to help you get the most from Visual Basic.

If you have a question about the product, first look in the print documentation, or consult online Help. If you can't find the answer, contact Microsoft Product Support Services.

Support services are available both within the United States and through subsidiary offices worldwide. For complete information, see Appendix E, "Microsoft Product Support Services."

CHAPTER 2

Your First Visual Basic Application



It takes just a few minutes to build your first Visual Basic application. You create the user interface by "drawing" controls, such as text boxes and command buttons on a form. Next, you set properties for the form and controls to specify such values as captions, color, and size. Finally, you write code to bring the application to life. The basic steps you take in creating your first application will show you principles that you'll use with every other application you develop.

This chapter provides an overview of the application development process, describes the terms and skills you need to use Visual Basic, and takes you step by step through several simple applications.

Contents

- Starting Visual Basic
- Steps to Creating a Visual Basic Application
- Working with Sample Applications
- Saving, Debugging, and Distributing Your Applications

BUTTERF.MAK and PICVIEW.MAK

Two of the sample applications in this chapter (BUTTERF.MAK and PICVIEW.MAK) are included on disk. If you installed the sample applications, you will find these applications in the \FIRSTAPP subdirectory of the main Visual Basic directory (\VB\SAMPLES\FIRSTAPP).

Starting Visual Basic

When you run the Visual Basic Setup program, Setup automatically creates a new program group and new program items for Visual Basic in Windows. You are then ready to start Visual Basic from Windows.

▶ To start Visual Basic from Windows

- Double-click the Visual Basic icon.

You can also start Visual Basic from either the File Manager or the MS-DOS prompt.

When you first start Visual Basic, you see the interface of the programming environment, as shown in Figure 2.1.

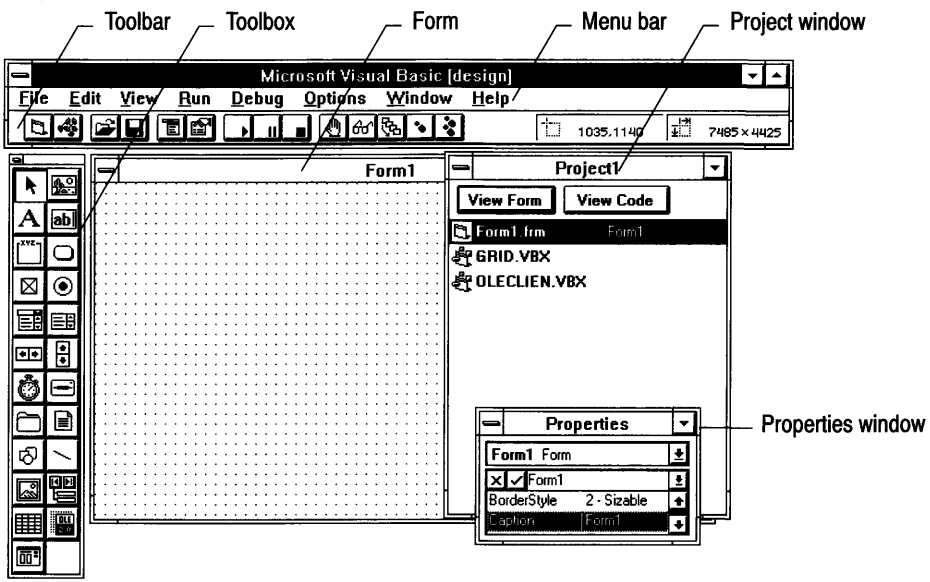


Figure 2.1 The Visual Basic Programming Environment

The Visual Basic interface consists of the following elements.

Toolbar Provides quick access to commonly used commands in the programming environment. You click an icon on the toolbar once to carry out the action represented by that icon.

Icon	Action	Menu equivalent
	Creates a new form	New Form command on the File menu
	Creates a new module	New Module command on the File menu
	Opens an existing project	Open Project command on the File menu
	Saves the current project	Save Project command on the File menu
	Displays the Menu Design window	Menu Design command on the Window menu
	Displays the Properties window	Properties command on the Window menu
	Starts an application in design mode	Start command on the Run menu
	Stops execution of a program while it's running	Break command on the Run menu
	Stops execution of an application and returns to design mode	End command on the Run menu
	Toggles breakpoint on the current line	Toggle Breakpoint on the Debug menu
	Displays the value of the current selection in the Code window	Instant Watch on the Debug menu
	Displays the structure of active calls	Calls command on the Debug menu
	Executes code one statement at a time in the Code window	Single Step command on the Debug menu
	Executes code one procedure or statement at a time in the Code window	Procedure Step command on the Debug menu

Toolbox Provides a set of tools that you use at design time to place controls on a form. For information on the specific controls, see Chapter 3, "Creating and Using Controls."

Menu Bar Displays the commands you use to build your application.

Form Serves as a window that you customize as the interface of your application. You add controls, graphics, and pictures to a form to create the look you want.

Project Window Lists the form, code modules, and custom control files that make up your current project. A *project* is the collection of files you use to build an application. For information on projects, see Chapter 5, "Managing Projects."

Properties Window Lists the property settings for the selected form or control. A *property* is the value of an object, such as size, caption, or color.

Steps to Creating a Visual Basic Application

There are three main steps to creating an application for Windows in Visual Basic:

1. Create the interface.
2. Set properties.
3. Write code.

To see how this is done, follow along in the rest of this section to create a simple application that consists of a text box and a command button. When you click the command button, the message "Hello, world!" appears in the text box.

Creating the Interface

The first step in building a Visual Basic application is to draw the objects that make up the interface. For this first application, you'll use two controls from the Toolbox.

Control	Name
	Text box
	Command button

▶ **To draw a control using the Toolbox**

1. Click the tool for the control you want to draw—in this case, the text box.
2. Move the pointer onto your form. The pointer becomes a cross hair, as shown in Figure 2.2.

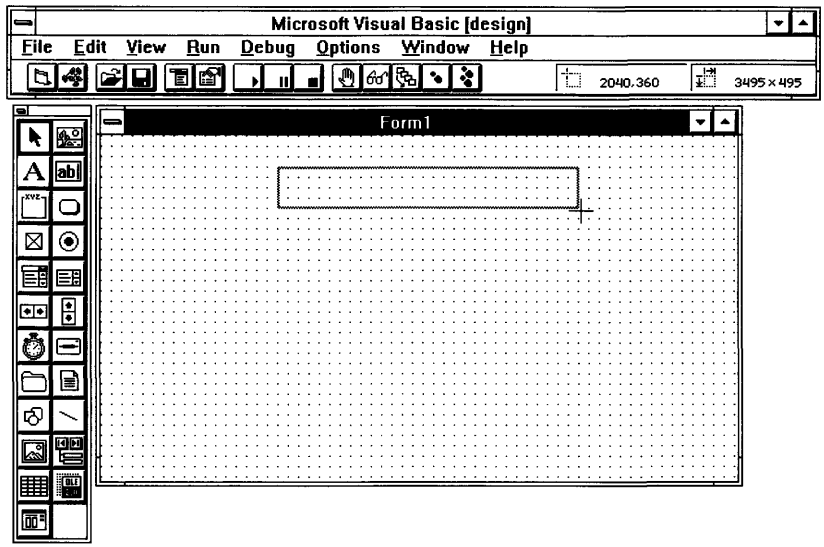


Figure 2.2 Drawing a text box with the Toolbox

3. Place the cross hair where you want the upper-left corner of the control.
4. Drag the cross hair until the control is the size you want. (*Dragging* means holding the left mouse button down while you move an object with the mouse.)
5. Release the mouse button.

   The control appears on the form.

A simple way to add a control to a form is to double-click the icon for that control in the Toolbox. This creates a default-size control located in the center of the form.

Notice that small rectangular boxes called *sizing handles* appear at the corners of the control; you'll use these in the next step as you resize the control.

▶ To resize a control

1. Select the control you want to resize by clicking it with the mouse.

   Sizing handles appear on the control.
2. Position the mouse pointer on a sizing handle, and drag it until the control is the size you want.

   The corner handles resize controls horizontally and vertically, while the side handles resize in only one direction.
3. Release the mouse button.

▶ **To move a control**

- Position the mouse pointer anywhere on the control other than on a sizing handle, and drag the control to a new location on the form.

You now have the interface for the "Hello, world!" application, shown in Figure 2.3.

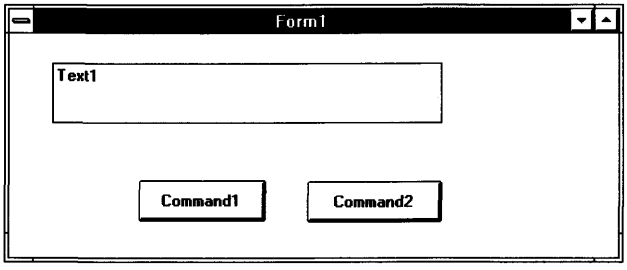


Figure 2.3 The interface for the "Hello, world!" application

Setting Properties

The next step is to set properties for the objects you've created. The Properties window (Figure 2.4) provides an easy way to set properties for all objects on a form. To open the Properties window, choose the Properties command from the Window menu, or click the Properties button on the toolbar.

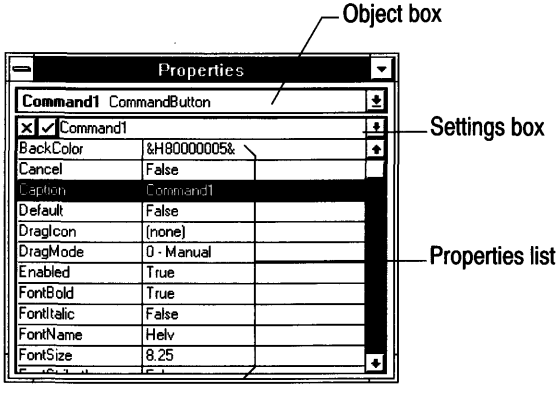


Figure 2.4 The Properties Window

The Properties window consists of the following elements:

- Object box—Displays the name of the object for which you can set properties. Click the arrow to the right of the box to select a form or control name from the list of objects for the current form.
- Settings box—Lets you edit the setting for the property selected in the Properties list. Some settings can be changed by clicking the arrow to the right of the box; this will display a list of choices. You can then click an item in the list to select it.
- Properties list—The left column displays all of the properties for the selected object; the right column displays the current setting for each property.

For the "Hello, world!" example, you'll need to change three property settings.

Object	Property	Setting
Form	Caption	Hello, world!
Text box	Text	(Empty)
Command button	Caption	OK

▶ **To set properties from the Properties window**

1. From the Window menu, choose Properties, or click the Properties button on the toolbar.

   The Properties window displays the settings for the selected form or control.

2. From the Properties list, select the name of a property.

   The current setting of the property appears in the Settings box.

3. In the Settings box, type or select the new property setting.

   Enumerated properties have a predefined list of settings. You can display the list by clicking the down arrow at the right of the Settings box, or you can cycle through the list by double-clicking a list item.

Writing Code

The *Code window* is where you write Visual Basic code for your application. Code consists of language statements, constants, and declarations. Using the Code window, you can quickly view and edit any of the code in your application.

▶ To open the Code window

- Double-click the form or control on the form for which you want to write code.

  –Or–

- From the Project window, select the name of the form and choose the View Code button.

Figure 2.5 shows the Code window that appears when you double-click the command button control.

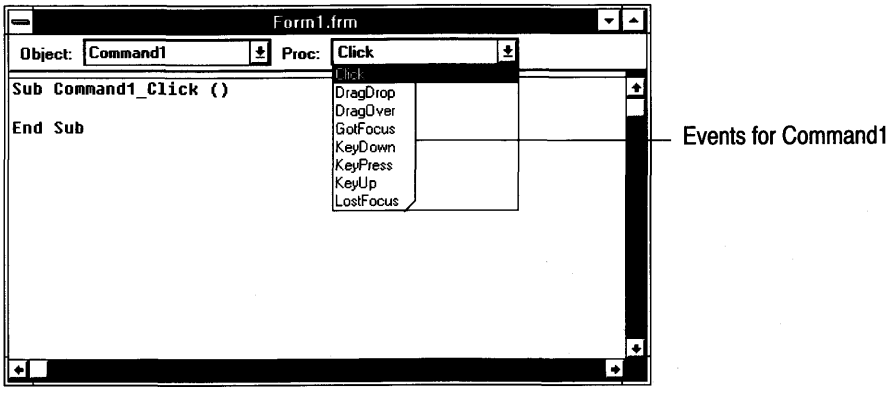


Figure 2.5 The Code Window

The Code window includes the following elements:

- Object box—Displays the name of the selected object. Click the arrow to the right of the list box to display a list of all objects associated with the form.
- Procedure list box—Lists the procedures for an object. The box displays the name of the selected procedure—in this case, Click. Choose the arrow to the right of the box to display all the procedures for the object.

Code in a Visual Basic application is divided into smaller blocks called *procedures*. An *event procedure*, such as those you'll create here, contains code that is executed when an event occurs (such as a user clicking a button). For more information on other types of procedures and event-driven programming in general, see Chapter 6, "Programming Fundamentals."

▶ **To create an event procedure**

1. In the Object box, select the name of an object in the active form. (The *active* form is the form that has the focus.)

   For this example, choose the command button, Command1.

2. In the Procedure list box, select the name of an event for the selected object.

   Here, the Click procedure is already selected, since it's the default procedure for a command button. Note that a *template* for the event procedure is now displayed in the Code window.

3. Type the following code between the **Sub** and **End Sub** statements:

```
Text1.Text = "Hello, world!"
```

   The event procedure should look like this:

```
Sub Command1_Click ()
    Text1.Text = "Hello, world!"
End Sub
```

You'll note here that the code is simply changing the Text property of the control named Text1 to read "Hello, world!" The syntax for this example takes the form of *object.property*, where `Text1` is the object and `Text` is the property. You can use this syntax to change any property setting for any form or control in response to events that occur while your application is running.

Running the Application

To run the application, choose Start from the Run menu, or click the Start icon on the toolbar (you can also press F5). Click the button you've created on the form, and you'll see "Hello, world!" displayed in the text box.

Working with Sample Applications

Visual Basic provides you with a wealth of tools beyond the ones you've used in this first application, so you'll soon be using many other features to manage and customize your applications. Reviewing sample applications can be an excellent way to learn more about Visual Basic. The following simple applications illustrate how easy it can be to add special features to your applications.

Simple Animation

The Butterfly application demonstrates how an *image control* can be used to create a simple animated butterfly. The application consists of three image controls. The first is loaded with an "open" butterfly bitmap and the second with a "closed" butterfly bitmap (see Figure 2.6). When the third image control is loaded first with the "open" bitmap and then with the "closed" bitmap, the butterfly appears to be opening and closing its wings and moving across the screen.

Figure 2.6 Butterfly bitmap (open and closed states)

To make the image control load the bitmaps at specific intervals, you use the *timer control*. The timer measures in increments of milliseconds, so this example will use an interval setting of 500 to make the wings open and close every 0.5 seconds.

To make the butterfly "fly" across the screen, you use the **Move** method. (A *method* is a Visual Basic reserved word that acts on a particular object.) In this case, the **Move** method works to move the image control across the screen. The syntax for specifying a method (*object.method*) is similar to the syntax for setting a property (*object.property*). The **Move** method allows you to specify how far to move the bitmap horizontally and vertically across the screen.

For More Information For information on the **Move** method, see Chapter 15, "Creating Graphics for Applications"; the *Language Reference*; or search Help for *Move*.

You begin creating this application by choosing New Project from the File menu. Visual Basic creates a new project and displays a new form. To draw the interface, you use image controls and a timer control.

Icon	Control
	Image control
	Timer control

Use the Toolbox to draw three image controls and a timer control on the form, as shown in Figure 2.7.

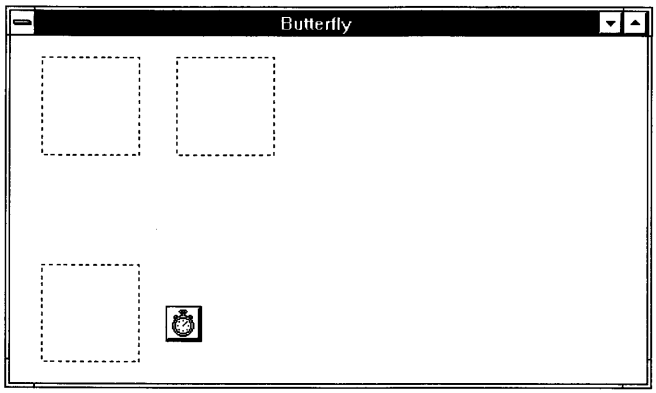


Figure 2.7 Form for the Butterfly sample application

Note The timer control can be placed anywhere on the form; it won't be visible when you run the program.

In the Properties window, set properties for the objects according to the following table.

Object	Property	Setting
Form	Caption	Butterfly
	WindowState	2 - Maximized
	BackColor	(White)
	ScaleMode	3 - Pixel
Timer	Interval	500
Image1	Name	OpenWings
	Picture	BFLY1.BMP
	Visible	**False**
Image2	Name	CloseWings
	Picture	BFLY2.BMP
	Visible	**False**
Image3	Name	Main
	Picture	BFLY1.BMP
	Visible	**True**

Setting the WindowState property of the form to 2-Maximized and the BackColor property to White clears the background on which the bitmap will be displayed. The Visible property determines whether you can see the control at run time. Since two of the image controls (OpenWings and CloseWings) are containers for the bitmaps, you want to make them invisible at run time by setting their Visible properties to **False**. The Visible property for the third image control (Main) is **True**.

All the code for the application is contained in the Timer1_Timer event procedure. Double-click the timer to display the Code window, and then type the following code:

```
Sub Timer1_Timer ()
Static PickBmp As Integer
    Main.Move Main.Left + 20, Main.Top - 5
    If PickBmp Then
        Main.Picture = OpenWings.Picture      ' Load open butterfly.
    Else
        Main.Picture = CloseWings.Picture     ' Load closed butterfly.
    End If
    PickBmp = Not PickBmp                     ' Toggle the value.
End Sub
```

The local variable `PickBmp` determines which bitmap is loaded into `Main.Picture`. Declaring `PickBmp` to be **Static** maintains its value (**True** or **False**) between Timer events. `PickBmp` is an integer variable, and the default value for an integer variable is 0, which is the same as **False**.

Every 0.5 seconds, when the Timer event is generated, the procedure uses the **Not** operator to toggle the value of `PickBmp` between **True** and **False**; this in turn controls whether the Main image control displays the OpenWings or the CloseWings image control.

Finally, the **Move** method causes the Main image control to move up and to the right of the screen by specifying new settings for the Top and Left properties of the control.

Note To control how the butterfly moves across the screen, you might want to experiment with different Top and Left values.

A Picture Viewer

With just a few lines of code, you can create a useful application that allows you to view bitmaps, icons, or Windows metafiles. The application consists of five controls: a drive list box, a directory list box, a file list box, an image control, and a label. The form for the Picture Viewer is shown in Figure 2.8.

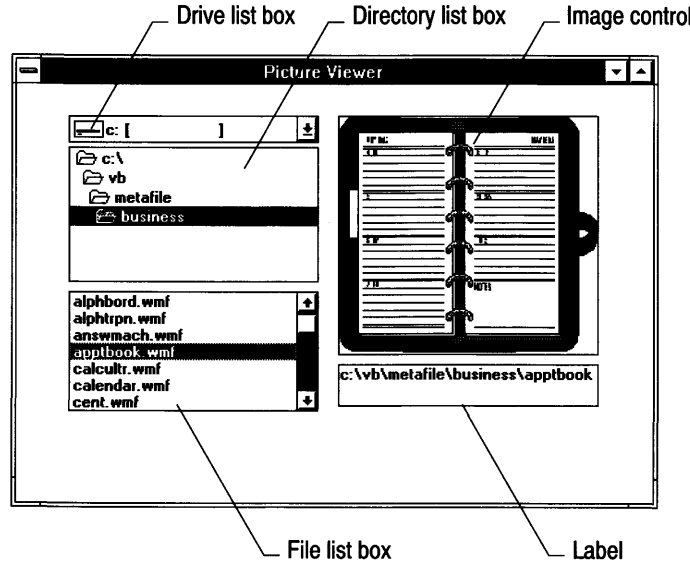


Figure 2.8 Form for the Picture Viewer sample application

When you run the application, you can display a picture by double-clicking the file name of a bitmap, icon, or metafile in the file list box.

The drive, directory, and file list boxes work together, the same way they do in other Windows–based dialog boxes: You select the drive and then use the scroll bar to find the directory you want. When you select the directory, a list of files in that directory is displayed in the file list box.

To create this application, start by choosing New Project from the File menu; then draw the controls on the form according to Figure 2.8. You'll be using the controls in the following table.

Icon	Control
	Drive list box
	Directory list box
	File list box
	Image control
	Label control

After you design the form, you'll need to set the following properties. (Use the default settings for all other properties and objects.)

Object	Property	Setting
Form	Caption	Picture Viewer
File list box	Pattern	*.BMP; *.WMF; *.ICO
Image control	BorderStyle	1 - Fixed Single
	Name	Open
	Stretch	**True**
Label	BorderStyle	1 - Fixed Single
	Caption	(Empty)

The Pattern property specifies the types of files displayed in the file list box. Since you want to view bitmaps, icons, and metafiles, you can specify their file name extensions to prevent other file types from being displayed.

In the Picture Viewer application, you'll create event procedures for three different controls.

Drive1_Change and Dir1_Change

To make the drive, directory, and file list boxes work in conjunction with each other, you need to add the following code to the Drive1_Change and Dir1_Change event procedures:

```
Sub Drive1_Change ()
    Dir1.Path = Drive1.Drive            ' Update directory path.
End Sub

Sub Dir1_Change ()
    File1.Path = Dir1.Path              ' Update files.
End Sub
```

File1_DblClick

The File1_DblClick event procedure for the file list box displays the full path name of the selected file in the label control and displays the picture itself in the image control. You'll need the following code:

```
Sub File1_DblClick ()
    If Right(File1.Path, 1) <> "\" Then
        Label1.Caption = File1.Path & "\" & File1.FileName
    Else                            ' If root directory
        Label1.Caption = File1.Path & File1.FileName
    End If
    Form1.Open.Picture = LoadPicture(Label1.Caption)
End Sub
```

Notice how the full path name is constructed:

- `File1.Path` returns the drive and directory path, `"\"` adds a backslash separator, and `File1.FileName` returns the file name.
- The **Right** function checks to see whether the path name is the root directory (\); if not, the full path name is assigned to the label's caption. If the path name is the root directory, the backslash is omitted.
- The **LoadPicture** function loads the bitmap file specified by the path name in the label caption.

For More Information For information on the **Right** and **LoadPicture** functions, see the *Language Reference*, or search Help for *Right* and *LoadPicture*.

Working with Multiple Procedures

When you create multiple procedures for an application like Picture Viewer, you'll often want to switch between different procedures as you write them. To switch between procedures in the Code window, you use the View Procedures dialog box, as shown in Figure 2.9.

▶ To switch between procedures in an application

1. In the Code window, choose Procedures from the Window menu, or press F2 to display the View Procedures dialog box.

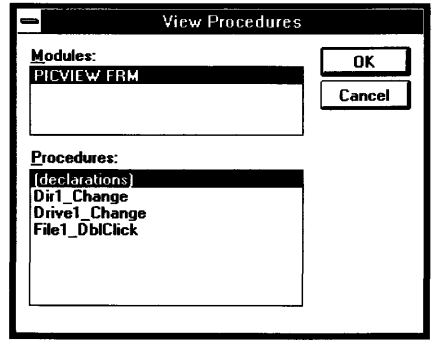


Figure 2.9 The View Procedures dialog box

2. Select the name of an object and procedure from the dialog box, and then choose OK.

   The selected procedure is displayed in the Code window.

▶ To cycle through procedures in a module

- Press CTRL+UP ARROW to display the previous procedure or CTRL+DOWN ARROW to display the next procedure.

For More Information For a list of additional keys you can use in the Code window, search Help for *Code window*.

Saving, Debugging, and Distributing Your Applications

Before you test an application, it's a good idea to save your work. To save in Visual Basic, choose Save Project from the File menu, or click the Save Project icon on the toolbar. Visual Basic will prompt you separately to save the form and then the project.

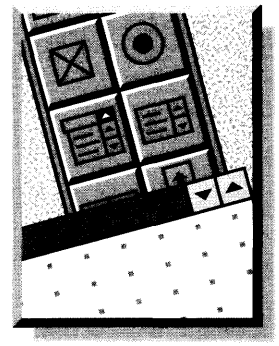
If your program doesn't run as predicted, check to make sure you have typed the lines of code as intended. While Visual Basic catches most syntax errors as they are typed, some errors—such as typing code in the wrong event procedure—won't be apparent until you run the program. For information on how to track down errors in code, see Chapter 9, "Debugging."

Once you've finished a project, you can save the application as an executable (.EXE) file by using the Make EXE File command on the File menu. And once you've saved your application as an executable file, you're ready to distribute it.

Note Visual Basic ships with SETUP1.MAK, a sample setup program you can use as a kind of template for setup programs you write for any Visual Basic applications you may want to distribute. The source code for SETUP1.MAK is in the \SETUPKIT\SETUP1 subdirectory of the main Visual Basic directory. For more information, search Help for *Setup Kit*.

CHAPTER 3

Creating and Using Controls



The first step to creating an application with Visual Basic is to create the interface: the forms, controls, and other objects the user will see and use. Then you set the properties and write Visual Basic code to make the interface active.

You use controls to get user input and to display output. Some of the controls you can use in your applications include text boxes, command buttons, and list boxes. Each type of control has its own set of properties and events. This chapter introduces you to many of the controls in Visual Basic.

Contents

- Visual Basic Controls
- Clicking Buttons to Perform Actions
- Displaying and Entering Text
- Controls That Present Choices to the User
- Getting and Setting the Focus of an Object
- Setting the Tab Order
- Enabling, Disabling, and Controlling Visibility at Run Time
- Working with Control Arrays

CONTROLS.MAK and CALC.MAK

Many of the code examples in this chapter are taken from the CONTROLS.MAK and CALC.MAK sample applications. If you installed the sample applications, you will find these applications in the \CONTROLS and \CALC subdirectories of the main Visual Basic directory (\VB\SAMPLES\CONTROLS and \VB\SAMPLES\CALC).

Visual Basic Controls

The Visual Basic Toolbox contains the tools you use to draw controls on your forms. Each tool in the Toolbox (Figure 3.1) represents a control.

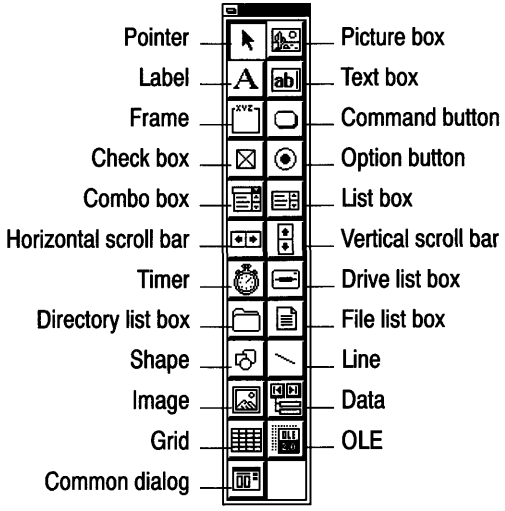


Figure 3.1 The Visual Basic Toolbox

The following table summarizes the Visual Basic controls found in the Toolbox. You may recognize some of these tools from earlier versions of Visual Basic.

Icon	Control	Description
	Pointer	Provides a way to move and resize forms and controls. (Note that this is *not* a control.)
	Picture box	Displays bitmaps, icons, or Windows metafiles. Provides an area in which to display text or acts as a visual container of other controls. See Chapter 15, "Creating Graphics for Applications."
	Label	Displays text a user cannot interact with or modify.

Icon	Control	Description
	Text box	Provides an area to input or display text.
	Frame	Provides a visual and functional container for controls.
	Command button	Carries out a command or action when a user chooses it.
	Check box	Displays a True/False or Yes/No option. Any number of check boxes on a form can be checked at one time.
	Option button	As part of an option group with other option buttons, displays multiple choices, from which a user can select only one.
	Combo box	Combines a text box with a list box. Allows a user to type in a selection or select an item from a drop-down list.
	List box	Displays a list of items that a user can choose from.
	Horizontal scroll bar Vertical scroll bar	Allows a user to select a value within a range of values. (These are used as separate controls and are not the same as the built-in scroll bars found with many controls.)
	Timer	Executes timer events at specified time intervals. See Chapter 17, "Interacting with the Environment."
	Drive list box	Displays and allows a user to select valid disk drives. See Chapter 18, "Using the File-System Controls."
	Directory list box	Displays and allows a user to select directories and paths. See Chapter 18, "Using the File-System Controls."
	File list box	Displays and allows a user to select from a list of files. See Chapter 18, "Using the File-System Controls."
	Shape	Adds a rectangle, square, ellipse, or circle to a form. See Chapter 15, "Creating Graphics for Applications."
	Line	Adds a straight-line segment to a form. See Chapter 15, "Creating Graphics for Applications."

Icon	Control	Description
	Image	Displays bitmaps, icons, or Windows metafiles; acts like a command button when clicked. See Chapter 15, "Creating Graphics for Applications."
	Data	Enables you to connect to an existing database and display information from it on your forms. See Chapter 20, "Accessing Databases with the Data Control."
	Grid	Displays a series of rows and columns and allows you to manipulate the data in its cells. See Chapter 13, "Using the Grid Control."
	OLE	Embeds data into a Visual Basic application. See Chapter 22, "Object Linking and Embedding (OLE)."
	Common dialog	Provides a standard set of dialog boxes for operations such as opening, saving, and printing files or selecting colors and fonts. See Chapter 4, "Menus and Dialogs."
	Menu	Creates ,menus in your Visual Basic applications. For information about menu controls, see Chapter 4, "Menus and Dialogs." You work with menu controls in the Menu Design window, which you can access either by choosing Menu Design from the Window menu or by clicking the Menu icon on the toolbar.

You can also refer to the summary tables found in the *Language Reference* and in Help for supported methods, properties, and events for each type of control.

Object Naming Conventions

When you first create an *object* (form or control), Visual Basic sets its Name property to a default value. For example, all command buttons have their Name property initially set to Command*n*, where *n* is 1, 2, 3, and so on. Visual Basic names the first command button drawn on a form Command1, the second Command2, and the third Command3.

There is nothing wrong with keeping the default name; however, when you have several controls of the same type, it makes sense to change their Name properties to something more descriptive. Since it may be difficult to distinguish the Command1 button on MyForm from the Command1 button on YourForm, a naming convention can help. This is especially true when an application consists of several form and code modules.

For example, you can use a prefix to describe the object type, followed by a descriptive name for the control. This makes the code more self-documenting and alphabetically groups similar objects together in the Properties window in the Object list box.

The following naming conventions for Visual Basic objects are used throughout this manual.

Table 3.1 Object Naming Conventions for Visual Basic

Object	Prefix	Example
Form	frm	frmFileOpen
Check box	chk	chkReadOnly
Combo box	cbo	cboEnglish
Command button	cmd	cmdCancel
Data	dat	datBiblio
Directory list box	dir	dirSource
Drive list box	drv	drvTarget
File list box	fil	filSource
Frame	fra	fraLanguage
Grid	grd	grdPrices
Horizontal scroll bar	hsb	hsbVolume
Image	img	imgIcon
Label	lbl	lblHelpMessage
Line	lin	linVertical
List box	lst	lstPolicyCodes
Menu	mnu	mnuFileOpen
OLE	ole	oleObject1
Option button	opt	optFrench
Picture box	pic	picDiskSpace
Shape (circle, square, oval, rectangle, rounded rectangle, and rounded square)	shp	shpCircle
Text box	txt	txtGetText
Timer	tmr	tmrAlarm
Vertical scroll bar	vsb	vsbRate

Object names, like variable names, follow the same rules as everything else you create in Visual Basic. The names you give to forms and controls:

- Must begin with a letter.
- Must contain only letters, numbers, and the underscore character (_); punctuation characters and spaces are not allowed.
- Must be no longer than 40 characters.

For more information on variable names and other naming conventions in Visual Basic, see Chapter 6, "Programming Fundamentals."

Clicking Buttons to Perform Actions

The easiest way to allow the user to interact with an application is to provide a button to click. You can use the command button control provided by Visual Basic, or you can create your own "button" using an image control containing a graphic, such as an icon.

Using Command Buttons

Most Visual Basic applications have command buttons that allow the user to simply click them to perform an action. When the user chooses the button, it not only carries out the appropriate action, it also looks as if it's being pushed in and released. Whenever the user clicks a button, the Click event procedure is invoked. You place code in the Click event procedure to perform any action you choose.

There are many ways to choose a command button at run time:

- Use a mouse to click the button.
- Move the focus to the button by pressing the TAB key, and then choose the button by pressing the SPACEBAR or ENTER. (See "Getting and Setting the Focus of an Object" later in this chapter.)
- Press an access key (ALT+the underlined letter) for a command button.
- Set the command button's Value property to **True** in code. Command1.Value = True
- If the command button is the default command button for the form, pressing ENTER chooses the button, even if you change the focus to a different control. At design time, you specify a default command button by setting that button's Default property to **True**.
- If the command button is the default Cancel button for the form, then pressing ESC chooses the button, even if you change the focus to another control. At design time, you specify a default Cancel button by setting that button's Cancel property to **True**.

All these actions cause Visual Basic to invoke the Click event procedure.

Example

You use the Caption property to display text on the button that tells the user what the button does. In Figure 3.2, the Test Buttons example from the Control Examples application (CONTROLS.MAK) contains a command button with its Caption property set to "Change Signal."

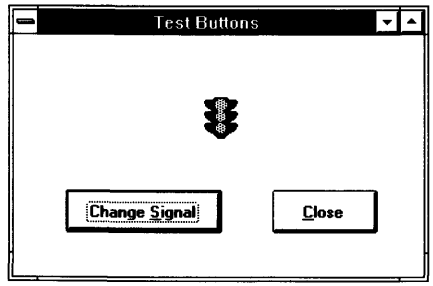


Figure 3.2 Command button with caption

When a user clicks the command button, the code in the command button's Click event procedure is executed. In the example, a different icon is displayed each time the button is clicked.

Using an Image Control for User Interaction

An image control also recognizes the Click event, so you can use this control anywhere you'd use a command button. Grouping several image controls together horizontally across the top of the screen—usually within a picture box—allows you to create a toolbar in your application.

For instance, the Test Buttons example also shows an image control that a user can choose like a command button. When the form is first displayed, the control displays one of the traffic icons from the Icon Library included with Visual Basic. Each time the image control is clicked, a new icon is displayed. Figure 3.3 shows the image control with one of the traffic icons (TRFFC10A.ICO).

Figure 3.3 Image control with traffic icon

To create a border around the image control, set the BorderStyle property to 1-Fixed Single.

Note Unlike command buttons, image controls do not appear pushed in when clicked. This means that there is no visual cue to the user that the "button" is being pushed.

For information on displaying a graphic image in an image control, see Chapter 15, "Creating Graphics for Applications."

Displaying and Entering Text

Use labels and text boxes when you want your application to display text on a form. Labels contain text that can only be read, while text boxes contain text that can be edited.

Using Labels to Display Text

A label control displays text that the user cannot directly change. Labels can also be used to identify controls, such as text boxes and scroll bars, that do not have their own Caption property.

By default, the caption is the only visible part of the label control. If you set the BorderStyle property to 1 (which you can do only at design time), however, the label appears with a border—giving it a look similar to a text box.

Note You work with forms and controls, set their properties, and write code for their events at *design time*, which is any time you're building an application in the Visual Basic environment. *Run time* is any time you are actually running the application and interacting with the application as the user would.

Sizing a Label to Fit Its Contents

Single-line label captions can be specified at design time in the Properties window. But what if you want to enter a longer caption, or a caption that will change at run time? Labels have two properties that help you size the controls to fit larger or smaller captions: AutoSize and WordWrap.

The AutoSize property determines if a control should be automatically resized to fit its contents. If set to **True**, the label grows horizontally to fit its contents, as shown in Figure 3.4.

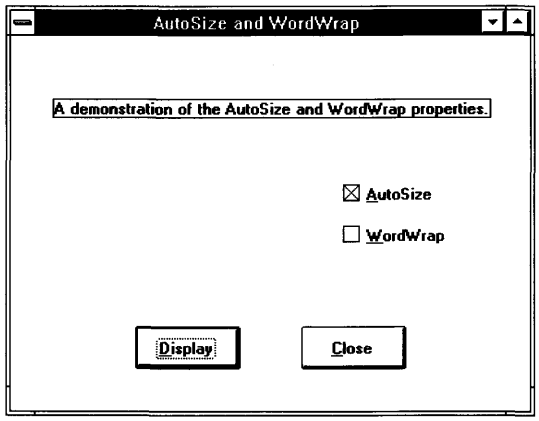


Figure 3.4 AutoSize example

The WordWrap property causes the label to grow vertically to fit its contents, while retaining the same width, as shown in Figure 3.5.

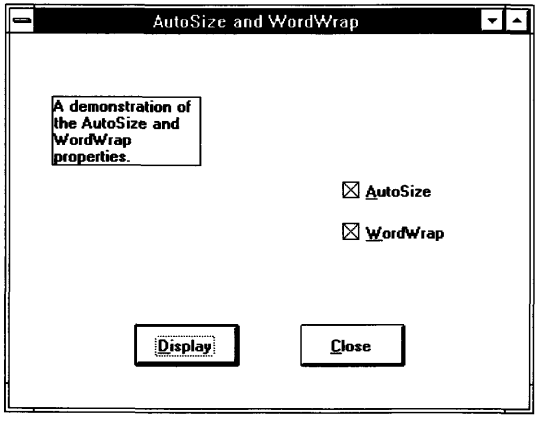


Figure 3.5 WordWrap example

Note If you run the AutoSize example from CONTROLS.MAK, you'll notice that for the WordWrap example to actually work, both check boxes must be selected on the form before you click the Display button. For the label's WordWrap property to take effect, AutoSize must be set to **True**. The width of the label is increased only if the width of a single word exceeds the current width of the control.

Using Labels to Create Access Keys

You can create access keys for any control that has a Caption property by adding an ampersand (&) before the letter in the caption you want to use as the access key. To assign an access key to controls that don't have captions, use a label with the control. Since labels can't receive focus, focus automatically moves to the next control in the tab order. Use this technique to assign access keys to text boxes, picture boxes, combo boxes, list boxes, drive list boxes, directory list boxes, grids, and images.

▶ **To assign an access key to a control with a label**

1. Draw the label first; then draw the control.

   –or–

   Draw the controls in any order and set the TabIndex property of the label to one less than the control.

2. Use an ampersand in the label's Caption property to assign the access key for the label.

Working with Text Boxes

Text boxes are versatile controls that can be used to get input from the user or to display text. Since they allow you to modify their contents, text boxes should not be used to display text that you don't want the user to change.

Multiline Text Boxes and Word Wrap

The look and behavior of a text box is strongly influenced by two properties, MultiLine and ScrollBars, which are set only at design time.

Note The ScrollBars property should not be confused with scroll bar controls, which are not attached to text boxes and have their own set of properties.

Setting MultiLine to **True** enables a text box to accept or display multiple lines of text at run time. A multiline text box automatically manages word wrap as long as there is no horizontal scroll bar. The ScrollBars property is set to 0-None by default. Automatic word wrap saves the user the trouble of inserting line breaks at the end of lines. When a line of text is longer than what can be displayed on a line, the text box wraps the text to the next line.

Line breaks cannot be entered in the Properties window at design time. Within a procedure, you create a line break by inserting a newline character—a carriage return followed by a linefeed (ANSI characters 13 and 10). For example, the following event procedure puts two lines of text into a multiline text box (Text1) when the form is loaded:

```
Sub Form_Load ()
    NL = Chr(13) + Chr(10)          ' Define newline character.
    Text1.Text = "Here are two lines" & NL & "in a text box"
End Sub
```

Creating a Password Text Box

A password box is a text box that allows a user to type in his or her password while displaying placeholder characters, such as asterisks. Visual Basic provides two text box properties, PasswordChar and MaxLength, which make it easy to create a password text box.

PasswordChar specifies the character displayed in the text box. For example, if you want asterisks displayed in the password box, you specify * for the PasswordChar property in the Properties window. Regardless of what character a user types in the text box, an asterisk is displayed, as shown in Figure 3.6.

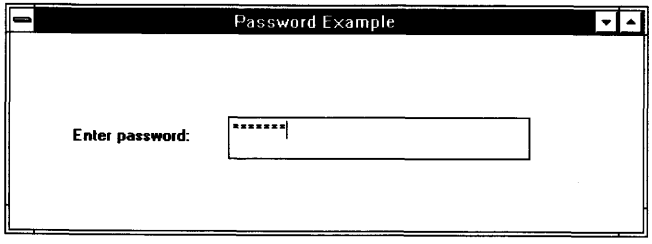


Figure 3.6 Password example

MaxLength lets you determine how many characters can be typed in the text box. After MaxLength is exceeded, the text box beeps and does not accept any further characters.

Canceling Keystrokes in a Text Box

You can use the KeyPress event to restrict or transform characters as they are typed. The KeyPress event uses one argument, *KeyAscii*. This argument is an integer that represents the numeric (ASCII) equivalent of the character just typed in the text box.

The next example demonstrates how to cancel keystrokes as they are typed. If the character typed is not within the specified range, the procedure cancels it by setting `KeyAscii` to 0. The text box for this example is named txtEnterNums, and the procedure prevents the text box from receiving any characters other than digits. Compare `KeyAscii` directly to the numeric (**Asc**) values of various characters.

```
Sub txtEnterNums_KeyPress (KeyAscii As Integer)
    If KeyAscii < Asc("0") Or KeyAscii > Asc("9") Then
        KeyAscii = 0                          ' Cancel the character.
        Beep                                  ' Sound error signal.
    End If
End Sub
```

Printing Quotation Marks in a String

Sometimes quotation marks (" ") appear in a string of text.

```
She said, "You deserve a treat!"
```

Because strings assigned to a variable or property are surrounded by quotation marks (" "), you must insert an additional set of quotation marks for each set to display in a string. Visual Basic interprets two quotation marks in a row as an embedded quotation mark.

For example, to create the preceding string, use the following code:

```
Text1.Text = "She said, ""You deserve a treat!"" "
```

To achieve the same effect, you can use the ASCII character (34) for a quotation mark:

```
Text1.Text = "She said, " & Chr(34) + "You deserve a treat!" & Chr(34)
```

Controls That Present Choices to the User

In most applications, you'll want to provide choices for the user. These can be in the form of option buttons or check boxes they select, list entries they choose, or scroll bars they use to select a value on a scale. The following table summarizes these controls.

To provide this feature	Use this control
A set of options from which a user can choose just one.	Option buttons (use frames if additional groups are needed)
A set of choices from which a user can choose one or more options.	Check boxes
A scrollable list of choices from which the user can choose.	List box
A list box also containing a text edit field. The user can either choose from the list or type a choice in the edit field.	Combo box
A range of choices corresponding to a numeric scale.	Horizontal or vertical scroll bar

Selecting Individual Options with Check Boxes

A check box indicates whether a particular condition is on or off. You use check boxes in an application to give users a True/False or Yes/No option. Since check boxes work independently of each other, a user can select any number of check boxes at the same time. So in Figure 3.7, AutoSize and WordWrap can both be checked.

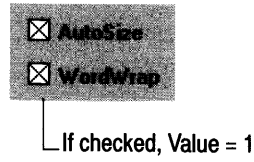


Figure 3.7 Check boxes

Example

The Check Box example uses a check box to determine whether the text is displayed in regular or italic font.

The application has a text box and two check boxes, as shown in Figure 3.8.

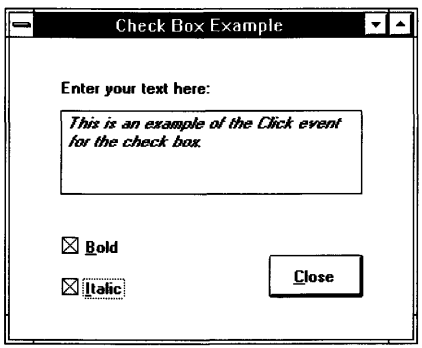


Figure 3.8 Check box example

The following table lists the property settings for the objects in the application.

Object	Property	Setting
Form	Name	frmCheck
	Caption	Check Box Example
Text box	Name	txtDisplay
	Text	(Empty)
Check box	Name	chkBold
	Caption	&Bold
Check box	Name	chkItalic
	Caption	&Italic

When you check Bold or Italic, the check box's Value property is set to 1; when unchecked, its Value property is set to 0.

The Click event for the check box occurs as soon as you click the box. The Click event procedure for the check box tests to see whether the check box has been selected (that is, if Value = 1). If so, the text is converted to bold or italic by setting the FontBold or FontItalic property to **True**:

```
Sub ChkBold_Click ()
    If ChkBold.Value = 1 Then           ' If checked
        txtDisplay.FontBold = True
    Else                                ' If not checked
        txtDisplay.FontBold = False
    End If
End Sub
```

```
Sub ChkItalic_Click ()
    If ChkItalic.Value = 1 Then          ' If checked
        TxtDisplay.FontItalic = True
    Else                                 ' If not checked
        TxtDisplay.FontItalic = False
    End If
End Sub
```

Grouping Options with Option Buttons

Option buttons present a set of choices to the user. Unlike check boxes, however, option buttons should always work as part of a group; selecting one option button immediately clears all the other buttons in the group. Defining an option button group tells the user, "Here is a set of choices, from which you can choose one and only one."

For example, in the option button group shown in Figure 3.9, the user can select one of three option buttons.

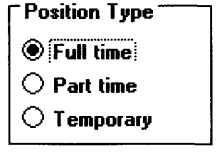


Figure 3.9 Selecting an option button

Creating Option Button Groups

All of the option buttons placed directly on a form—not in a frame or picture box—constitute one group. If you want to create additional option button groups, you must place some of them inside frames or picture boxes.

All the option buttons inside any given frame constitute a separate group, as do all the option buttons inside a picture box. When you create a separate group this way, always draw the frame or picture box first, and then draw the option buttons on top of it. Figure 3.10 shows two option button groups.

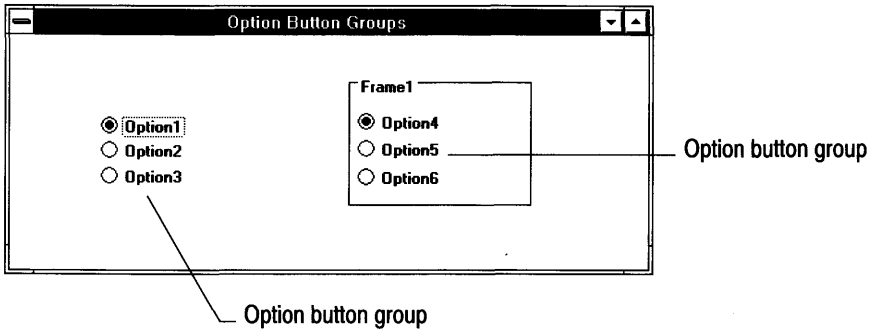


Figure 3.10 Option button groups

A frame provides a visual and functional grouping for controls. Frames are commonly used to group option buttons or check boxes. A user can select only one option button in the group when you draw option buttons in a frame.

▶ **To group controls in a frame**

1. Choose the Frame button from the Toolbox and draw the frame.
2. Draw controls within the frame.

Drawing the frame first and then adding the controls allows you to move the frame and controls together. If you try to move existing controls onto a frame, the controls will not move with the frame.

Note If you have existing controls that you want to group in a frame, you can select all the controls and cut and paste them into a frame or picture control.

Containers for Controls

While controls are independent objects, a certain *parent and child relationship* exists between forms and controls. Figure 3.10 demonstrates how option buttons can be contained within a form or within a frame control.

To understand the concept of containers, you need to understand that all controls are children of the form on which they are drawn. In fact, most controls support the read-only Parent property, which returns the form on which a control is located. Being a child affects the placement of a control on the parent form. The Left and Top properties of a control are relative to the parent form, and controls cannot be moved outside the boundaries of the parent. Moving a form moves the controls as well.

When a control is contained within a frame or picture control, the Left and Top properties of the control are relative to its container. This means that when you move the container, the control's position relative to the container's Left and Top properties does not change because the control moves with the container.

Selecting or Disabling Option Buttons

When the user selects an option button, the button appears shaded. An option button can be selected by:

- Clicking it at run time with the mouse.
- Tabbing to the option button group and then using the arrow keys to select an option button within the group.
- Assigning its Value property to **True** in code:

```
Option1.Value = True
```

- Using a shortcut key specified in the caption of a label. (For more information on how to do this, see "Assigning Access Keys and Shortcut Keys" in Chapter 4, "Menus and Dialogs.")

To select a default button in an option button group, set its Value property to **True** at design time. It remains selected until a user selects a different option button or code changes it.

To disable an option button, set its Enabled property to **False**. The button will be unavailable (grayed) when the program is run.

Example

The form shown in Figure 3.11 uses option buttons to determine which of three number systems (octal, decimal, or hexadecimal) to use. When the user selects a button, the number displayed in the text box is converted to the new number system.

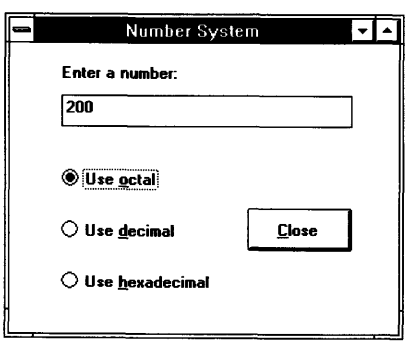


Figure 3.11 Option button example

Create a form with a text box and three option buttons. The properties are set as shown in the following table.

Object	Property	Setting
Text box	Name	txtNumber
	Text	(Empty)
First option button	Name	optOctButton
	Caption	Use &octal
	Value	**True**
Second option button	Name	optDecButton
	Caption	Use &decimal
Third option button	Name	optHexButton
	Caption	Use &hexadecimal

The Number System application responds to events as follows:

- The Change event for the text box reads in the value (using the appropriate number system) and stores it in a form-level numeric variable, `CurrentNum`.
- The Click event optOctButton returns `CurrentNum` in octal.
- The Click event for optDecButton returns `CurrentNum` in decimal.
- The Click event for optHexButton returns `CurrentNum` in hexadecimal.

What's key to this approach is the use of a form-level variable, `CurrentNum`. This variable represents the value of the text box (txtNumber) in numeric form. The Change event keeps this value current, so all the Click event procedures have to do is return the number in the proper number system. `CurrentNum` must be declared in the Declarations section of the form code:

```
Dim CurrentNum As Long
```

By default, the variable is initialized to 0. That's an acceptable default, so there's no need for initialization code.

The Change event procedure checks to see which number system (octal, decimal, or hexadecimal) is in effect and then reads in the number. This procedure looks at the *optionbutton*.`Value` condition to determine the number system to use:

```
Sub txtNumber_Change ()
    If optOctButton.Value = True Then
        CurrentNum = Val("&O" & LTrim(txtNumber.Text) & "&")
    ElseIf optDecButton.Value = True Then
        CurrentNum = Val(LTrim(txtNumber.Text) + "&")
    Else
        CurrentNum = Val("&H" & LTrim(txtNumber.Text) & "&")
    End If
End Sub
```

The **Val** function is used to translate the string to a number, and can recognize octal, decimal, and hexadecimal strings. The **LTrim** function strips the text of leading blanks. The "&O" prefix causes a digit string to be interpreted as octal, and the "&H" prefix causes it to be interpreted as hexadecimal.

The Click event procedures for the option buttons use the **Oct** or **Hex** functions to display `CurrentNum` in the appropriate number system:

```
Sub optOctButton_Click ()
    txtNumber.Text = Oct(CurrentNum)
End Sub

Sub optDecButton_Click ()
    txtNumber.Text = Format(CurrentNum)
End Sub

Sub optHexButton_Click ()
    txtNumber.Text = Hex(CurrentNum)
End Sub
```

For More Information For information on the **Val** and **LTrim** functions or the **Oct** or **Hex** functions, see the *Language Reference*, or search Help for *Val*, *LTrim*, *Oct*, or *Hex*.

Using List Boxes and Combo Boxes

List boxes and combo boxes present a list of choices to the user. By default, the choices are displayed vertically in a single column, although you can set up multiple columns as well. If the number of items exceeds what can be displayed in the list box, scroll bars will automatically appear on the control. The user can then scroll up and down or left to right through the list.

Figure 3.12 shows a single-column list box.

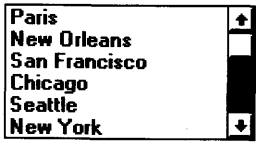


Figure 3.12 Single-column list box

A combo box control combines the features of a text box and a list box. This control allows the user to make a selection either by typing text into the combo box or by selecting an item from its list.

Combo Box Styles

There are three combo box styles:

- Drop-down combo box (Style 0)
- Simple combo box (Style 1)
- Drop-down list box (Style 2)

Figure 3.13 shows the three styles of combo boxes.

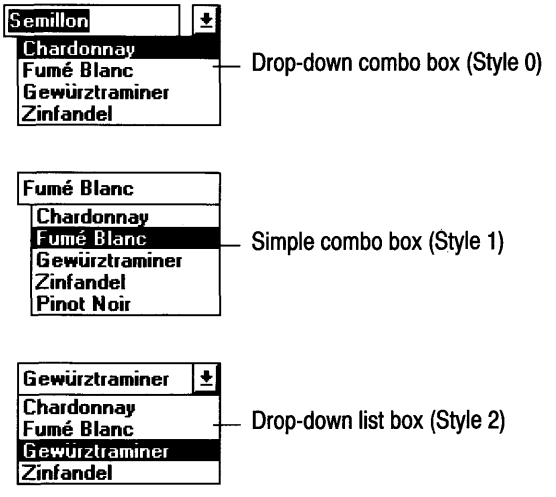


Figure 3.13 Combo box styles

Drop-down Combo Box (Style 0)

With the default setting (Style = 0), a combo box is a drop-down combo box. The user can either enter text directly (as in a text box) or click the detached arrow at the right of the combo box to open a list of choices. Selecting one of the choices puts it into the text portion at the top of the combo box. The user also can open the list by pressing ALT+DOWN ARROW when the control has the focus. When you want to allow the user to choose only from a list, but you want to conserve screen space, you can use a drop-down list box.

Simple Combo Box (Style 1)

Setting the Style property of a combo box to 1 specifies a simple combo box in which the list is displayed at all times. To display all entries in the list, you must draw the list box large enough to display the entries. A vertical scroll bar is automatically inserted when there are more entries than can be displayed. The user can still enter text directly or select from the list. As with a drop-down list box, a simple combo box also allows users to enter choices not on the list.

Drop-down List Box (Style 2)

A drop-down list box is much like a regular list box—it displays a list of items from which a user must choose. Unlike list boxes, however, the list is not displayed until you click the arrow to the right of the box. Use this type of list box when space is at a premium. The Add Watch dialog box in Visual Basic contains two drop-down list boxes. You can see this dialog box by choosing Add Watch from the Debug menu.

When to Use a Combo Box Instead of a List Box

Generally, a combo box is appropriate when there is a list of *suggested* choices, and a list box is appropriate when you want to limit input to what is on the list. A combo box contains an edit field, so choices not on the list can be typed in this field.

In addition, combo boxes save space on a form. Since the full list is not displayed until the user clicks the down arrow (except for Style 1, which is always dropped down), a combo box can easily fit in a small space where a list box would not fit.

The following table lists unsupported combo box events.

Style property	Unsupported event
0 (Drop-down combo)	DblClick
1 (Simple combo)	DropDown
2 (Drop-down list)	Change, DblClick

A recommended practice for list box events (especially when the list box appears as part of a dialog box) is to add a command button to use with the list box. The Click event procedure for this button should make use of the list-box selection, carrying out whatever action is appropriate for your application.

Double-clicking an item in the list should have the same effect as selecting the item and then clicking the command button. To do this, have the DblClick procedure for the list box call the Click procedure for the command button:

```
Sub List1_DblClick ()
    Command1_Click
End Sub
```

Or, set the value of the command button's Value property to **True**, which will automatically invoke the event procedure:

```
Sub List1_DblClick ()
    Command1.Value = True
End Sub
```

This provides mouse users with a shortcut, yet does not prevent keyboard users from performing the same action. Note that there is no keyboard equivalent for the DblClick event.

Adding Items to a List

To add items to a list box or combo box, use the **AddItem** method, which has the following syntax:

box.**AddItem** *item*[, *index*]

Argument	Description
box	Name of the list or combo box.
item	String expression to add to the list. If *item* is a literal constant, enclose it in quotation marks.
index	Specifies where the new item is to be inserted in the list. An index of 0 represents the first position. If *index* is omitted, the item is inserted at the end (or in the proper sorted order).

Note If the Sorted property is set to **True** and no index is specified, then Visual Basic always keeps the items sorted alphabetically, regardless of the order in which you place them there. When the Sorted property is set to **True**, using the **AddItem** method with the *index* argument can lead to unpredictable, unsorted results.

While list items are commonly added in the Form_Load event procedure, you can use the **AddItem** method at any time. This gives you the ability to add items to the list dynamically (in response to user actions).

Example

The following code places "Germany," "India," "France," and "USA" into a list box named List1:

```
Sub Form_Load ()
    List1.AddItem "Germany"
    List1.AddItem "India"
    List1.AddItem "France"
    List1.AddItem "USA"
End Sub
```

Whenever the form is loaded at run time, the list appears as shown in Figure 3.14.

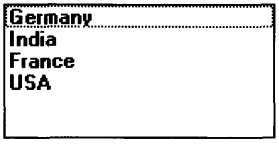


Figure 3.14 "Countries" list box

Adding an Item at a Specified Position

To add an item to a list at a specific position, specify an index value after the new item. For example, the next line of code inserts "Japan" into the first position, adjusting the position of the other items downward:

```
List1.AddItem "Japan", 0
```

Notice that it is 0, not 1, that specifies the first item in a list (see Figure 3.15).

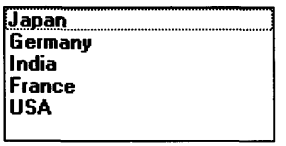


Figure 3.15 Adding an item to a list

Sorting a List

You can specify that items be added to a list in alphabetical order by setting the Sorted property to **True** and omitting the index. The sort is not case-sensitive; thus, the words "sort" and "Sort" are treated the same.

Removing Items from a List

You can use the **RemoveItem** and **Clear** methods to delete items from a list or combo box. RemoveItem has one argument, *index*, which specifies the item to remove:

control.**RemoveItem** *index*

The arguments *control* and *index* are the same as for **AddItem.**

For example, to remove the first entry in a list, you would add the following line of code:

```
List1.RemoveItem 0
```

To remove all the entries in a list, use the **Clear** method:

```
List1.Clear
```

Getting List Contents with the Text Property

Usually, the easiest way to get the value of the currently selected item is to use the Text property:

- With list boxes, the Text property always corresponds to a list item a user selects at run time.
- With combo boxes, the Text property corresponds to whatever is entered in the text-box portion of the control at run time. This can be either a selected list item or a string that a user types in the text box.

For example, the following code displays information for Canada if a user selects "Canada" from a list box:

```
Sub List1_Click ()
    If List1.Text = "Canada" Then
        Text1.Text = "Canada has 24 million people."
    End If
End Sub
```

The Text property contains the currently selected item in the List1 list box. The code checks to see if "Canada" has been selected and, if true, displays the population in the Text box.

Accessing List Items with the List Property

The List property provides access to all items in the list. This property contains an array in which each item in the list is an element of the array. Each item is represented in string form. To refer to an item in the list, use this syntax:

control.**List**(*index*)

The argument *control* is a reference to a list box or combo box, and *index* is the position of the item. The top item has an index of 0, the next has an index of 1, and so on. For example, the following statement displays the third item (*index* = 2) in a list in a text box:

```
Text1.Text = List1.List(2)
```

Determining Position with the ListIndex Property

If you want to know the position of the selected item in a list, use the ListIndex property. This property sets or returns the index of the currently selected item in the control and is available only at run time. Setting the ListIndex property for a list or combo box also generates a Click event for the control.

The value of this property is 0 if the first (top) item is selected, 1 if the next item down is selected, and so on. ListIndex is –1 if no item is selected or if a user enters a choice in a combo box (Style 0 or 1) instead of selecting an existing item in the list.

Note The NewIndex property allows you to keep track of the index of the last item added to the list. This can be useful when inserting an item into a sorted list. For details about NewIndex, see the *Language Reference*, or search Help for *NewIndex*.

Returning the Number of Items with the ListCount Property

To return the number of items in a list box or combo box, use the ListCount property. For example, the following statement uses the ListCount property to determine the number of entries in a list box:

```
Text1.Text = "You have " & List1.ListCount & " entries listed"
```

Example

This example shows how the **AddItem**, **RemoveItem**, and **Clear** methods can be used with the ListIndex and ListCount properties to add and remove list entries at run time. The example in Figure 3.16 lets a user type a client's name in a text box, which can be added to the list box if the Add button is chosen. A user can remove a current list item by selecting the item and choosing the Remove button.

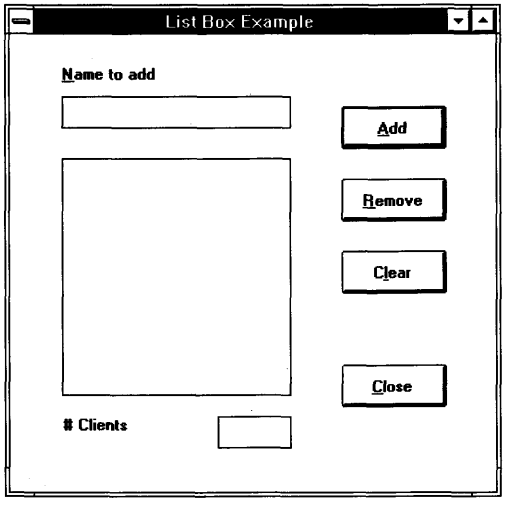


Figure 3.16 A list box using AddItem, RemoveItem, and Clear methods

The number of clients in the list box is displayed in a label that looks like a text box (BorderStyle is set to 1-Fixed Single). This label is updated every time a client name is added or removed. Since the Sorted property for the list box is set to **True**, items are added to the list box in alphabetical order.

Create a form with a text box, a list box, three labels, and three command buttons. The following table lists the property settings for the objects in the application.

Object	Property	Setting
Top text box	Name	txtName
	Text	(Empty)
Top label	Name	lblName
	Caption	Name to add
Bottom label	Name	lblClients
	Caption	# Clients
Number of clients label	Name	lblDisplay
	Caption	(Empty)
	BorderStyle	1-Fixed Single
List box	Name	lstClient
	Sorted	**True**
Add command button	Name	cmdAdd
	Caption	&Add
Remove command button	Name	cmdRemove
	Caption	&Remove
Clear command button	Name	cmdClear
	Caption	C&lear

Add this code to the cmdAdd_Click event procedure:

```
Sub cmdAdd_Click ()
    Client = txtName.Text                          ' Get client name.
    lstClient.AddItem Client                       ' Add it to list box.
    txtName.Text = ""                              ' Clear text box.
    lblDisplay.Caption = lstClient.ListCount       ' Display number.
End Sub
```

Add this code to the cmdRemove_Click event procedure:

```
Sub cmdRemove_Click ()
    Dim Ind As Integer

    Ind = lstClient.ListIndex          ' Get index.
    If Ind >= 0 Then                   ' Make sure list item is selected.
        lstClient.RemoveItem Ind       ' Remove it from list box.
        lblDisplay.Caption = lstClient.ListCount    ' Display number.
    Else
        Beep                           ' If nothing selected, beep.
    End If
End Sub
```

Add this code to the cmdClear_Click event procedure:

```
Sub cmdClear_Click ()
    lstClient.Clear
    lblDisplay.Caption = lstClient.ListCount        ' Display number.
End Sub
```

Creating Multiple-Column and Multiple-Selection List Boxes

The Columns property allows you to specify the number of columns in a list box. This property can have the following values.

Value	Description
0	Normal single-column list box with vertical scrolling.
1	Single-column list box with horizontal scrolling.
>1	Multiple-column list box with horizontal scrolling.

Visual Basic takes care of wrapping list items to the next line and adding a horizontal scroll bar to the list if needed; if the list fills a single column, no scroll bar is added. Wrapping to the next column also occurs automatically as needed. Note that if a list box entry is wider than the width of a column, the text is truncated.

You can allow users to select multiple items from a list. Multiple selection is handled by setting the MultiSelect property, which can have the following values.

Value	Type of selection	Description
0	None	Standard list box.
1	Simple multiple selection	Keystrokes (for example, CTRL+LEFT ARROW, SHIFT+LEFT ARROW, or the SPACEBAR) or mouse dragging selects or deselects additional items in the list.
2	Extended multiple selection	The SHIFT+CLICK or SHIFT+an arrow key extends the selection to include all the items between the current and previous selections.

Example

To create a multiple-column, multiple-selection list box, you need to set both the Columns and the MultiSelect properties of a list box. In the following example, these properties are used to create such a list box.

You'll notice that when you run the application, the list box contains two columns, as shown in Figure 3.17.

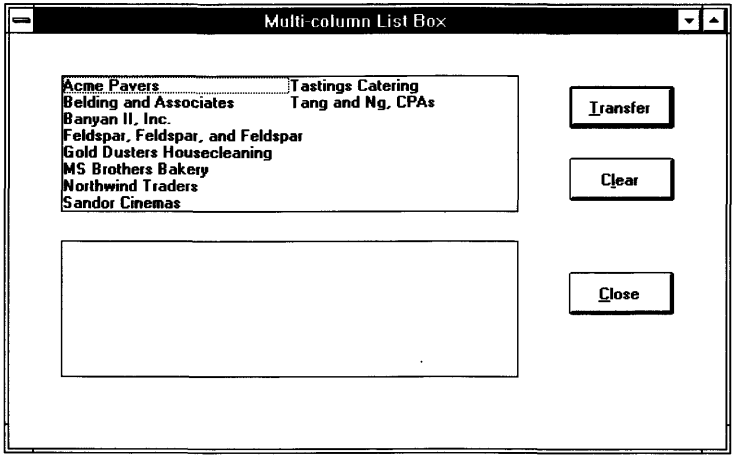


Figure 3.17 Multi-column list box

If you draw the list box large enough to hold all the items in one column, the second column will be empty; the other items will wrap, and horizontal scroll bars will appear automatically only if the list box is not long enough. Try resizing the top list box and adding additional list items to see how Visual Basic automatically handles multiple columns.

The example uses the Selected property—a Boolean array containing the selection status of a list box—to determine which items are selected. Each entry in the array corresponds to a list item and is set to **True** if the item is selected, or **False** if it is not selected. After the user selects items from the list, each array entry is checked to see if it is set (**True**). If so, the entry is added to the second list, a normal single-column list box, using the **AddItem** method.

Set the properties for the example as indicated in the following table.

Object	Property	Setting
Form	Caption	Multi-column List Box
Top list box	Name	lstTop
	Columns	2
	MultiSelect	2-Extended
Bottom list box	Name	lstBottom
First command button	Name	cmdTransfer
	Caption	&Transfer
Second command button	Name	cmdClear
	Caption	C&lear

The MultiSelect property allows you to select a range of values. If you click the first list item, then press SHIFT and click the last item in the range (or use the SHIFT+DOWN ARROW key), all the items in the range become selected.

Add code to the Form_Load procedure to initialize the first list:

```
Sub Form_Load ()
    lstTop.AddItem "Acme Pavers"
    lstTop.AddItem "Belding and Associates"
    lstTop.AddItem "Banyan II, Inc."
    lstTop.AddItem "Feldspar, Feldspar, and Feldspar"
    lstTop.AddItem "Gold Dusters Housecleaning"
    lstTop.AddItem "MS Brothers Bakery"
    lstTop.AddItem "Northwind Traders"
    lstTop.AddItem "Sandor Cinemas"
    lstTop.AddItem "Tastings Catering"
    lstTop.AddItem "Teng and Ng, CPAs"
End Sub
```

Add the following code to the Click event procedure for the Transfer command button:

```
Sub cmdTransfer_Click ()
    For n = 0 To (lstTop.ListCount - 1)
        If lstTop.Selected(n) = True Then          ' If selected,
            lstBottom.AddItem lstTop.List(n)       ' add to list.
        End If
    Next
End Sub
```

Notice how the array index values start from 0 and go to `ListCount - 1`.

Add the following code to the Click event procedure for the Clear command button:

```
Sub cmdClear_Click ()
    lstBottom.Clear
End Sub
```

Using Scroll Bars as Input Devices

Although scroll bars are often tied to text boxes or windows, you'll sometimes see them used as input devices. Since these controls can indicate the current position on a scale, scroll bar controls can be used individually to control input to a program—for example, to control the sound volume or to view the elapsed time in a timed process.

These types of controls operate independently from other controls and have their own set of events, properties, and methods. Scroll bar controls are not the same as the built-in scroll bars that are attached to text boxes, list boxes, combo boxes, or MDI forms (text boxes and MDI forms have a ScrollBars property to add or remove scroll bars that are attached to the control).

For example, several items in the Microsoft Windows Control Panel use scroll bars, including the Mouse dialog box shown in Figure 3.18.

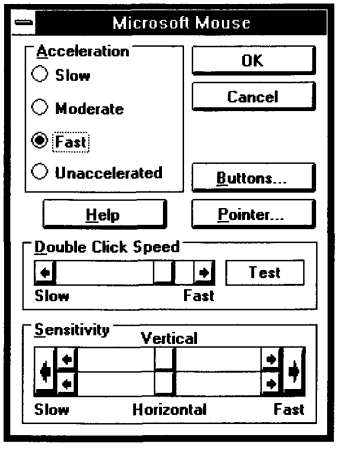


Figure 3.18 Scroll bars in the Microsoft Mouse Control Panel

Visual Basic allows you to use scroll bars the same way. With the Toolbox, you can create your own horizontal and vertical scroll bar controls.

The Value Property

The Value property (which, by default, is 0) is an integer value corresponding to the position of the scroll box in the scroll bar. When the scroll box position is at the minimum value, it moves to the leftmost position (for horizontal scroll bars) or the top position (for vertical scroll bars). When the scroll box is at the maximum value, the scroll box moves to the rightmost or bottom position. Similarly, a value halfway between the bottom and top of the range places the scroll box in the middle of the scroll bar.

In addition to using mouse clicks to change the scroll bar value, a user can also drag the scroll box to any point along the bar. The resulting value depends on the position of the scroll box, but it is always within the range of Min to Max properties set by the user.

Note Min can be larger than Max if you want your scroll bar to display information changing from a larger to a smaller value.

Scroll Bar Events

The following events allow you to monitor the movement of the scroll bar.

Event	Description
Change	Occurs after the scroll box is moved.
Scroll	Occurs as the scroll box is moved. Does not occur if the scroll arrows or scroll bar is clicked.

Using the Scroll event provides access to the scroll bar value as it is being dragged. The Change event occurs after the scroll box is released or when the scroll bar or scroll arrows are clicked. You can use the Scroll event and Change event together as shown in the following example to get an immediate update of the scroll value with regular scroll bar behavior.

Example

Colors provide an interesting demonstration of scroll bar controls. The scroll bar demo shown in Figure 3.19 allows the user to interactively change the background color of a text box by adjusting three horizontal scroll bars.

The scroll bars correspond to the three arguments of the **RGB** function: *red*, *green*, and *blue*. (The **RGB** function allows you to set colors in a Visual Basic application.) When a scroll bar value is changed, the RGB color is also changed and is then assigned to the BackColor property of the text box.

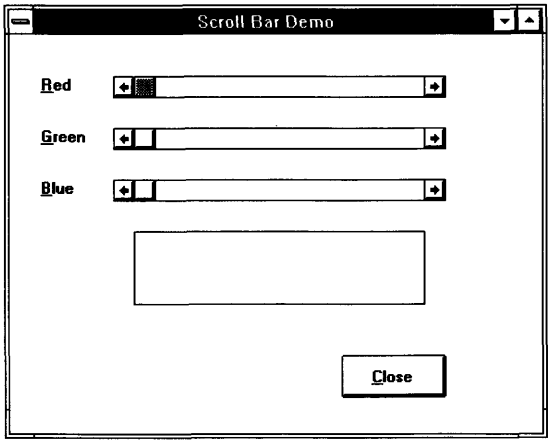


Figure 3.19 Using scroll bars to control color

Start by choosing New Project from the File menu and then draw three horizontal scroll bars, three labels, and a text box, as shown in Figure 3.19. The following table lists the property settings for the objects in the application.

Object	Property	Setting
Form	Caption	Scroll Bar Demo
Top label	Caption	&Red
Second label	Caption	&Green
Third label	Caption	&Blue
Top scroll bar	Name	hsbRed
	Max	255
	Min	0
	LargeChange	30
	SmallChange	5
Second scroll bar	Name	hsbGreen
	Max	255
	Min	0
	LargeChange	35
	SmallChange	5
Third scroll bar	Name	hsbBlue
	Max	255
	Min	0
	LargeChange	35
	SmallChange	5
Text box	Name	txtColor
	Text	(Empty)

Min and Max specify the range of values that the scroll bar can take. Zero and 255 correspond to the valid values for the **RGB** function.

The Scroll procedure for each of the scroll bars responds by calling a general procedure, DoColor, to recalculate the color. The Change procedure is used to call the Scroll procedure. Using the Scroll event and Change event together results in scroll bars that change instantaneously as the scroll box is dragged. If you do not need the scroll bars' value to change until after the mouse is released, you can use the Change event alone.

Scroll Event Procedures

To create the event procedures for the scroll bars, double-click the top scroll bar to display the Code window, and then select the Scroll event from the Procedures list. Add a line that calls the DoColor procedure to each event procedure:

```
Sub hsbRed_Scroll ()
    DoColor             ' Call the DoColor general procedure.
End Sub

Sub hsbGreen_Scroll ()
    DoColor             ' Call the DoColor general procedure.
End Sub

Sub hsbBlue_Scroll ()
    DoColor             ' Call the DoColor general procedure.
End Sub
```

Change Event Procedures

Next, select the Change event from the Procedures list. Add a line that calls the Scroll event procedure:

```
Sub hsbRed_Change ()
    hsbRed_Scroll               ' Call the Scroll event.
End Sub

Sub hsbGreen_Change ()
    hsbGreen_Scroll             ' Call the Scroll event.
End Sub

Sub hsbBlue_Change ()
    hsbBlue_Scroll              ' Call the Scroll event.
End Sub
```

Tip The three scroll bars can also be created as a control array. For example, the first scroll bar could be hsbColor(0), the second scroll bar hsbColor(1), and the third scroll bar hsbColor(2). This would allow you to have one Change event procedure for all three controls. For instructions on how to create a control array, see the section "Working with Control Arrays" later in this chapter.

DoColor General Procedure

The DoColor procedure uses the values of the three scroll bars, combines them with the **RGB** function, and assigns the result to the background color of the text box. The **RGB** function takes red, green, and blue input values, each running from 0 to 255.

▶ **To create the DoColor general procedure**

1. Select "(general)" from the Object list box.
2. From the View menu, choose New Procedure.
3. In the New Procedure dialog box, select the Sub option.
4. Type **DoColor** in the Name box.

Visual Basic automatically creates a code template for the DoColor procedure in the Code window. Type the following line between the **Sub** and **End Sub** statements:

```
Sub DoColor ()
    txtColor.BackColor = RGB(hsbRed.Value, hsbGreen.Value,
    ➥ hsbBlue.Value)
End Sub
```

Getting and Setting the Focus of an Object

Focus is the ability to receive user input through the mouse or keyboard. When an object has the focus, it can receive input from a user; for example, when a text box has the focus, a user can type text in it.

The GotFocus and LostFocus events occur when an object receives or loses focus.

Event	Description
GotFocus	Occurs when an object receives focus.
LostFocus	Occurs when an object loses focus. A LostFocus event procedure is primarily used for verification and validation updates, or for reversing or changing conditions you set up in the object's GotFocus procedure.

An object can receive focus in these ways:

- Selecting an object at run time.
- Using an access key to select a control at run time.
- Using the **SetFocus** method in code.

You can see when some objects have the focus. For example, when command buttons have the focus, they appear with a highlighted border around the caption (See Figure 3.20).

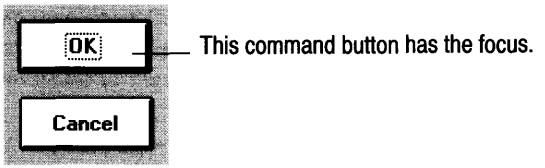


Figure 3.20 A command button showing focus

An object can receive focus only if its Enabled and Visible properties are set to **True**. The Enabled property allows the object to respond to user-generated events such as keypress and mouse events. The Visible property determines whether an object is visible on the screen.

Note Frames, labels, menus, lines, shapes, images, and timers cannot receive focus. A form can receive focus only if it doesn't contain any controls that can receive the focus.

To determine which control has the focus at run time, you can use the ActiveControl or ActiveForm properties of the Screen object. For example, you can use the ActiveControl property and the **If...TypeOf** statement to determine if the control with the focus is a text box:

```
If TypeOf Screen.ActiveControl Is Textbox Then
    .
    .
    .
End If
```

Setting the Tab Order

The *tab order* is the order in which a user moves from one control to another by pressing TAB. Normally, the tab order is the same as the order in which you created the controls.

For example, assume you first create two text boxes, Text1 and Text2, and then a command button, Command1. When the application starts, Text1 has the focus. Pressing TAB moves the focus between controls in the order they were created, as shown in Figure 3.21.

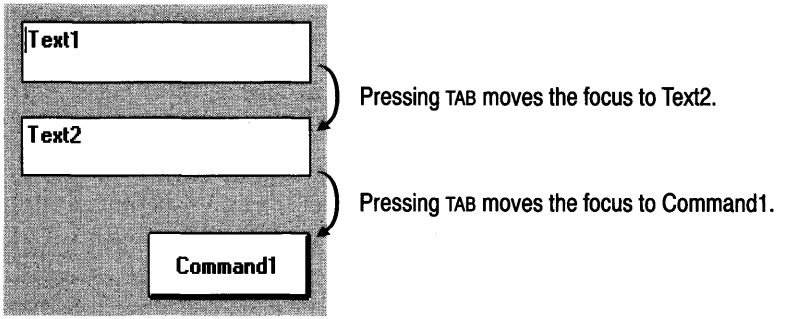


Figure 3.21 Tab example

▶ To change the tab order for a control

- Set the TabIndex property for the control.

By default, the first control drawn has a TabIndex value of 0, the second has a TabIndex of 1, and so on. When you change a control's tab order position, Visual Basic automatically renumbers the tab order positions of the other controls to reflect insertions and deletions.

For example, you could set the Command1 button to be first in the tab order with this statement:

```
Command1.TabIndex = 0
```

The TabIndex values for the other controls are automatically adjusted upward, as listed in the following table.

Control	TabIndex before Command1 is first in Tab order	TabIndex after Command1 is first in Tab order
Text1	0	1
Text2	1	2
Command1	2	0

The highest TabIndex setting is always one less than the number of controls in the tab order (because numbering starts at 0). Even if you set the TabIndex property to a number higher than the number of controls, Visual Basic converts the value back to the number of controls minus 1. If you set the TabIndex property to a negative number, Visual Basic generates an error.

Note Controls that cannot get the focus, as well as disabled and invisible controls, are not included in the tab order. As a user presses the TAB key, these controls are skipped.

Removing a Control from the Tab Order

Normally, pressing TAB at run time selects each control in the tab order. You can remove a control from the tab order by setting its TabStop property to **False** (0).

A control whose TabStop property has been set to **False** still maintains its position in the actual tab order, even though the control is skipped when you cycle through the controls with the TAB key.

Note An option button group has a single tab stop. The selected button (that is, the button with its Value set to **True**) has its TabStop property automatically set to **True**, while the other buttons have their TabStop property set to **False**.

Enabling, Disabling, and Controlling Visibility at Run Time

You can change the state of a control at run time by setting properties in code. In the case of command buttons, there are several actions you may find useful, as shown in the following table.

Action	Technique	Purpose
Disabling	Set Enabled property to **False**.	Prevents user from choosing the command.
Enabling	Set Enabled property to **True**.	Allows user to choose the command again.
Making invisible	Set Visible property to **False**.	Disables the control and removes it from the visible form.

When you disable or enable a command button, you're controlling the user's access to the command. Generally, you disable a command because the command is inappropriate at the time—for example, when a command is supposed to delete a specified file, but the file is read-only.

The Enabled property is an example of a Boolean property; it takes the value **True** or **False**. Visual Basic stores Boolean values as integers. All nonzero integers equate to **True**, and zero equates to **False**.

For example, you could check to see whether a command button is enabled by checking its Enabled property:

```
If Command1.Enabled = True Then
    Text1.Text = "The button is enabled."
Else
    Text1.Text = "The button is disabled."
End If
```

All controls except frame, shape, line, and label controls have the Enabled property; you can use similar code to disable a control or to find out if it is enabled.

Working with Control Arrays

A *control array* is a group of controls that share the same name and type. They also share the same event procedures. A control array has at least one element and can grow to as many as 254. Elements of the same control array have their own property settings. Common uses for control arrays include menu controls and option button groupings.

Why Use Control Arrays?

Control arrays are useful if you want several controls to share code. For example, if three option buttons are created as a control array, the same code is executed regardless of which button was clicked.

If you want to create a new control at run time, that control must be a member of a control array. With a control array, each new element inherits the common event procedures of the array.

Without the control array mechanism, creating new controls at run time is not possible, since a completely new control would not have any event procedures. Control arrays solve this problem, because each new control inherits the common event procedures already written for the array. For example, if your form has several text boxes that each receive a date value, a control array can be set up so that all of the text boxes share the same validation code.

The Calculator sample application shown in Figure 3.22 contains two control arrays—the number buttons and the operator buttons. If you installed the sample applications, you will find this application in the \CALC subdirectory of the main Visual Basic directory (\VB\SAMPLES\CALC).

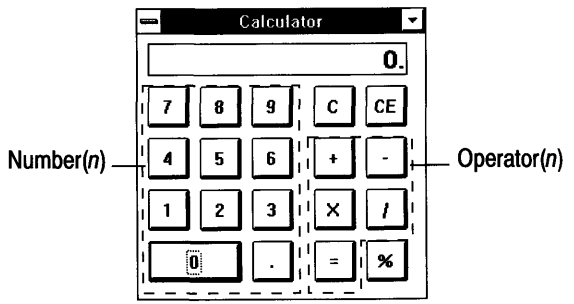


Figure 3.22 Control array example

The Name and Index property values for the control arrays in the Calculator example are listed in the following table.

Number(*n*)	**Operator(*n*)**
0 = Number(0)	+ = Operator(1)
1 = Number(1)	– = Operator(2)
2 = Number(2)	X = Operator(3)
3 = Number(3)	/ = Operator(4)
4 = Number(4)	= = Operator(5)
5 = Number(5)	
6 = Number(6)	
7 = Number(7)	
8 = Number(8)	
9 = Number(9)	

Notice how each control is referred to with the syntax *controlname*(*index*). You specify the index of a control when you create it. In fact, specifying any index for a control at design time makes that control part of an array.

The Index property distinguishes one element of the control array from another. When one of the controls in the array recognizes an event, Visual Basic calls a common event procedure and passes an argument (the Index property) to identify which control actually recognizes the event.

For example, the first line of the Number_Click event procedure is:

```
Sub Number_Click (Index As Integer)
```

If Number(0) recognizes the event, Visual Basic passes 0 as the *Index* argument, and if Number(1) recognizes the event, Visual Basic passes 1 as the *Index* argument. Other than the index value, the remainder of the Number_Click code that is executed is the same for both Number(0) through Number(9).

Creating a Control Array at Design Time

There are three ways to create a control array at design time:

- Assign the same name to more than one control.
- Copy an existing control and then paste it on the form.
- Set the Index property to a value that is not null.

Note You must create menu control arrays in the Menu Design window. For details on how to do this, see the section "Creating a Menu Control Array" in Chapter 4, "Menus and Dialogs."

▶ **To add a control array element by changing its name**

1. Draw the controls you want to be in the control array. (The controls must all be of the same type.) Decide which control will become the first element in the array.
2. Select one of the controls and change its Name setting to the Name setting for the first element in the array.
3. When you type an existing Name for a control in the array, Visual Basic displays a dialog box asking you to confirm that you want to create a control array. Choose Yes to confirm the action.

For example, if the name of the first element in a control array is cmdCtlArr, you would choose a command button to add to the array and then set its name to cmdCtlArr. The message "You already have a control named 'cmdCtlArr.' Do you want to create a control array?" is displayed. Choose Yes to confirm the operation.

Controls added this way share only their Name property and control type; all other properties remain the same as when the control was originally drawn.

▶ **To add a control array element by copying an existing control**

1. Draw a control in the control array.
2. While the control has the focus, choose Copy from the Edit menu.
3. Choose Paste from the Edit menu. Visual Basic displays a dialog box asking you to confirm that you want to create a control array. Choose Yes to confirm the action.

   This control is assigned an index value of 1. The first control you drew has a value of 0.

The index value of each new array element corresponds to the order in which the element was added to the control array. When controls are added this way, most of the visual properties, such as height, width, and color, are copied from the first control in the control array to the new controls. Properties for control array elements added at run time can't be changed in the Properties window.

Adding to a Control Array at Run Time

You can add and remove controls at run time using the **Load** and **Unload** statements. However, the control to be added must be an element of an existing control array. Use this syntax:

Load *control(index%)*

Unload *control(index%)*

Argument	Description
control	Name of the control to add to or delete from the control array.
index%	The control's index value in the array.

When you load a new element of a control array, most of the property settings are copied from the lowest existing element in the array—in this example, the element with the 0 index value. The Visible, Index, and TabIndex property settings are not automatically copied to new elements of a control array.

Note Visual Basic generates an error if you attempt to use the **Load** statement with an index number already in use in the array.

You can use the **Unload** statement to remove any control created with **Load**. However, you cannot use **Unload** to remove controls created at design time, regardless of whether or not they are part of a control array.

Example

The control array example demonstrates how controls—in this case, option buttons—are added and deleted at run time. The example allows the user to add option buttons that change the background color of a picture box.

Start with a form, and then draw a picture box, two option buttons, a label, and two command buttons, as shown in Figure 3.23.

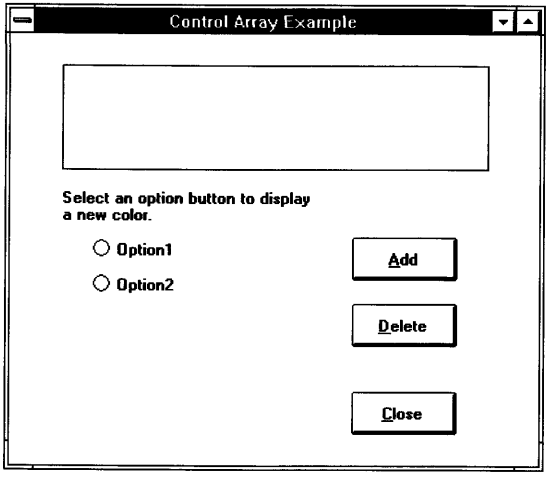


Figure 3.23 Adding controls at run time

The following table lists the property settings for the objects in the application.

Object	Property	Setting
Form	Caption	Control Array Example
Picture Box	Name	picDisplay
Option1	Name Index	optButton 0
Option2	Name Index	optButton 1
Label	Caption	Select an option button to display a new color.
First command button	Name Caption	cmdAdd &Add
Second command button	Name Caption	cmdDelete &Delete

Next you need to add the event procedures for the option buttons and command buttons. Start by adding the Form declaration:

```
Dim MaxId As Integer
```

The Click event procedure is shared by all the option buttons:

```
Sub optButton_Click (Index As Integer)
    picDisplay.BackColor = QBColor(Index + 1)
End Sub
```

New option buttons are added by the Click event procedure for the Add command button. The code checks to be sure no more than ten option buttons are loaded before the **Load** statement is executed. Once a control is loaded, its Visible property must be set to True.

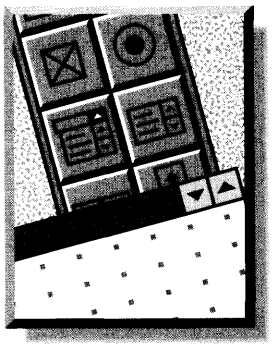
```
Sub cmdAdd_Click ()
    If MaxId = 0 Then MaxId = 1              ' Set total option buttons.
    If MaxId > 8 Then Exit Sub               ' Only ten buttons allowed.
    MaxId = MaxId + 1                        ' Increment button count.
    Load OptButton(MaxId)                    ' Create new button.
    OptButton(0).SetFocus                    ' Reset button selection.
    ' Set new button under previous button.
    OptButton(MaxId).Top = OptButton(MaxId - 1).Top + 400
    OptButton(MaxId).Visible = True          ' Display new button.
    OptButton(MaxId).Caption = "Option" & MaxId + 1
End Sub
```

Option buttons are removed by the Click event procedure for the Delete command button:

```
Sub cmdDelete_Click ()
    If MaxId = 1 Then Exit Sub           ' Keep first two buttons.
    Unload OptButton(MaxId)              ' Delete last button.
    MaxId = MaxId - 1                    ' Decrement button count.
    OptButton(0).SetFocus                ' Reset button selection.
End Sub
```

CHAPTER 4

Menus and Dialogs



Many simple applications consist of one form and several controls, but you can enhance your Visual Basic applications by adding menus and dialog boxes. This chapter shows you how to create menus and dialog boxes and use them in an application.

Contents

- Creating Menus at Design Time
- Controlling Menus at Run Time
- Working with Dialog Boxes

TEXTEDIT.MAK

Many of the code examples in this chapter are taken from the Text Editor sample application (TEXTEDIT.MAK). If you installed the sample applications, you will find this application in the \MENUS subdirectory of the main Visual Basic directory (\VB\SAMPLES\MENUS).

Creating Menus at Design Time

If you want your application to provide a set of commands to users, menus offer a convenient and consistent way to group commands and an easy way for users to access them.

Figure 4.1 illustrates the elements of a menu interface on an untitled Visual Basic form.

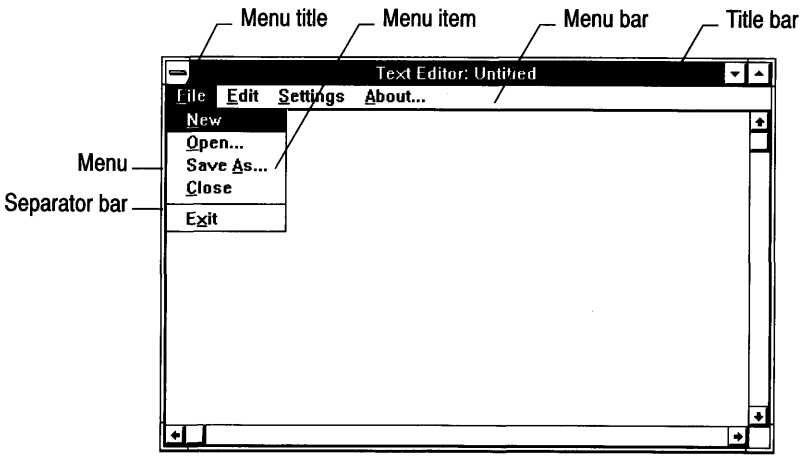


Figure 4.1 The elements of a menu interface on a Visual Basic form

The *menu bar* appears immediately below the *title bar* on the form and contains one or more *menu titles*. When you click a menu title (such as File), a menu containing a list of menu items drops down. Menu items can include commands (such as New and Exit), separator bars, and submenu titles. Each menu item the user sees corresponds to a menu control you define in the Menu Design window.

To make your application easier to use, you should group menu items according to their function. In Figure 4.1, for example, the file-related commands New, Open, and Save As are all found on the File menu.

Some menu items perform an action directly; for example, the Start menu item on the Run menu runs the currently loaded project. Other menu items display a *dialog box*, a window that requires the user to supply information needed by the application to perform the action. For example, the Open Project command on the File menu displays the Open Project dialog box.

Using the Menu Design Window

Menus are created using the Menu Design window. You add menu items to a menu at design time by creating menu controls and setting properties to define their appearance.

▶ **To display the Menu Design window**

- Choose Menu Design from the Window menu.

–Or–

- Choose the Menu Design button on the toolbar.

This opens the Menu Design window, shown in Figure 4.2.

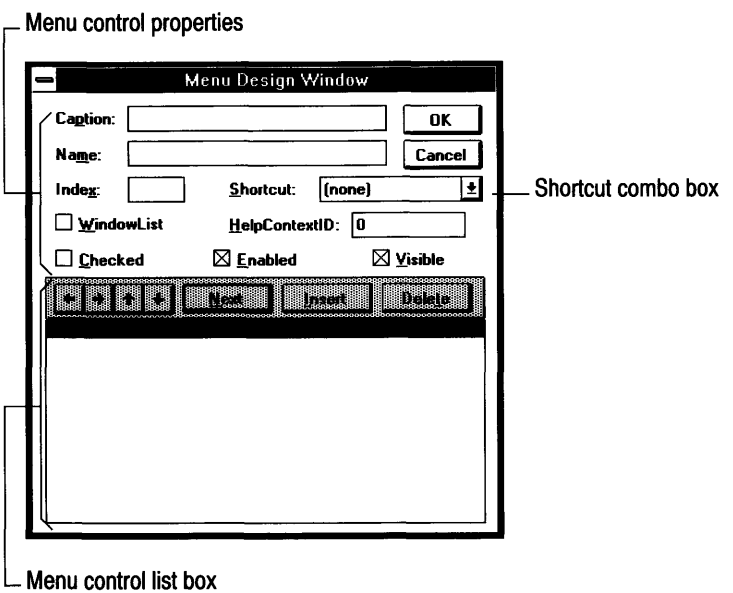


Figure 4.2 A Menu Design window

All the menu control design-time properties are shown in the Menu Design window. The two most important properties for menu controls are:

- Name—This is the name you use to reference the menu control from code.
- Caption—This is the text that appears on the control.

Other properties in the Menu Design window, including Index and Checked, are described later in this chapter.

The menu control list box lists all the menu controls for the current form. When you type a menu item in the Caption text box, that item also appears in the menu control list box. Selecting an existing menu control from the list box allows you to edit the properties for that control.

For example, Figure 4.3 shows the menu controls for the File menu in the Text Editor application.

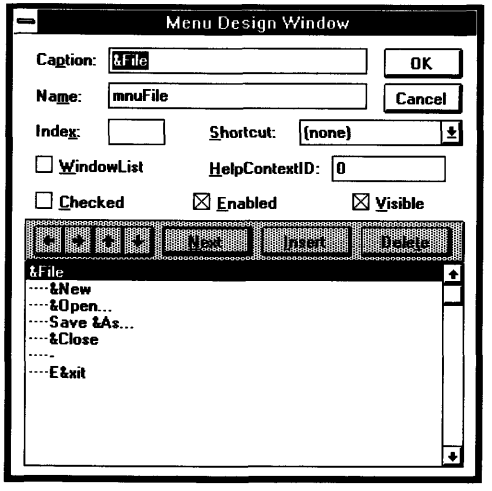


Figure 4.3 File menu controls in the Menu Design window

The position of the menu control in the menu control list box determines whether the control is a menu title, menu item, submenu title, or submenu item:

- A menu control that appears flush left in the list box is displayed on the menu bar as a menu title.
- A menu control that is indented once in the list box is displayed on the menu bar when the user clicks the preceding menu title.
- A menu control followed by menu controls that are further indented becomes a submenu title. Menu controls indented below the submenu title become items of that submenu.
- A menu control with a hyphen (-) as its Caption property setting appears as a separator bar. A *separator bar* divides menu items into logical groups.

▶ To create menu controls in the Menu Design window

1. Select the form.
2. From the Window menu, choose Menu Design.

   –or–

   Choose the Menu Design button on the toolbar.
3. In the Caption text box, type the text for the first menu title that you want to appear on the menu bar.

   The menu title text is displayed in the menu control list box.
4. In the Name text box, type the name that you will use to refer to the menu control in code.
5. Use the left arrow and right arrow buttons to change the indentation level of the control.
6. Set other properties for the control, if desired.
7. Choose Next to create another menu control.

   –or–

   Use the Insert button to add a menu control between existing controls.

   You can also use the up arrow and down arrow buttons to move the control among the existing menu controls.
8. Choose OK to close the Menu Design window when you have created all the menu controls for that form.

   The menu titles you create are displayed on the form. Click a menu title to drop down its corresponding menu items.

For More Information For information on menu control properties, see the *Language Reference*, or search Help for *Menu*.

Creating a Menu Control Array

A *menu control array* is a set of menu items on the same menu that share the same name and event procedures. Use a menu control array to:

- Create a new menu item at run time, when it must be a member of a control array. Text Editor, for example, uses a menu control array to store a list of opened files while the application is running.
- Simplify code, since common blocks of code can be used for all menu items.

Each menu control array element is identified by a unique index value, indicated in the Index property box on the Menu Design window. When a member of a control array recognizes an event, Visual Basic passes its Index property value to the event procedure as an additional argument. Your event procedure must include code to check the value of the Index property, so you can determine which control you're working with. (For more information on control arrays, see Chapter 3, "Creating and Using Controls.")

▶ To create a menu control array in the Menu Design window

1. From the Window menu, choose Menu Design.
2. Create the menu item that will become the first element in the array by setting its Caption and Name properties.
3. Set the Index property for the first element in the array to 0.
4. Create a second menu item at the same level of indentation as the first.
5. Set the Name property of the second element to the same as the first element and set its Index property to 1.
6. Repeat steps 2–5 for subsequent elements of the array.

Important Elements of a menu control array must be contiguous in the menu control list box and must be at the same level of indentation. When you're creating menu control arrays, be sure to include any separator bars that appear on the menu.

In the Text Editor application, the Edit menu controls Cut, Copy, and Paste are set up as a control array with the following property settings.

Caption property	Name property	Index property
Cu&t	mnuEditItem	0
C&opy	mnuEditItem	1
&Paste	mnuEditItem	2

Note Ampersands within a caption (for example, Cu&t) indicate the access key for the menu item. For more information on access keys, see the section "Assigning Access Keys and Shortcut Keys" later in this chapter.

Writing Code for Menu Controls

When the user chooses a menu control, a Click event occurs. You need to write a Click event procedure in code for each menu control. All menu controls except separator bars (and disabled or invisible menu controls) recognize the Click event.

Visual Basic displays a menu automatically when the menu title is chosen; therefore, it is not necessary to write code for a menu title's Click event procedure unless you want to perform another action, such as disabling certain menu items each time the menu is displayed.

Note At design time, the menus you create are displayed on the form when you close the Menu Design window. Choosing a menu item displays the Click event procedure for that menu control.

Writing Code for the Edit Menu

You refer to an element in a control array by specifying its index value along with its name. In the preceding Edit menu from the Text Editor application, for example, mnuEditItem(0) refers to Cut, the first menu item on the Edit menu and the first element in the mnuEditItem control array.

The index value of the selected menu item is passed to the event procedure when the user clicks that item on the menu. Since all the elements in the array share the same event procedure code, you can use conditional statements such as **If...Then** or **Select Case** to determine what code will be executed. For example, this code uses **Select Case** to cut, copy, and paste with the Clipboard:

```
Sub mnuEditItem_Click (Index As Integer)
    Select Case Index
        Case 0                      ' If Index = 0, user chose Cut.
            ' Copy selected text to Clipboard.
            Clipboard.Clear         ' Clear the Clipboard.
            Clipboard.SetText txtEdit.SelText
            ' Clear selected text from the document.
            txtEdit.SelText = ""
        Case 1                      ' If Index = 1, user chose Copy.
            Clipboard.Clear         ' Clear the Clipboard.
            ' Copy selected text to Clipboard.
            Clipboard.SetText txtEdit.SelText
        Case 2                      ' If Index = 2, user chose Paste.
            ' Paste Clipboard text (if any) into document.
            txtEdit.SelText = Clipboard.GetText()
    End Select
End Sub
```

For More Information For information about the Clipboard object, see the *Language Reference*, or search Help for *Clipboard*.

Creating Submenus

Each menu you create can include up to four levels of submenus. A *submenu* branches off another menu to display its own menu items. You may want to use a submenu when:

- The menu bar is full.
- A particular menu control is seldom used.
- You want to emphasize one menu control's relationship to another.

If there is room on the menu bar, however, it's better to create an additional menu title instead of a submenu. That way, all the controls are visible to the user when the menu is dropped down. It's also good programming practice to restrict the use of submenus so users don't get lost trying to navigate your application's menu interface. (Most applications use only one level of submenus.)

Figure 4.4 displays a menu interface with four levels of submenus.

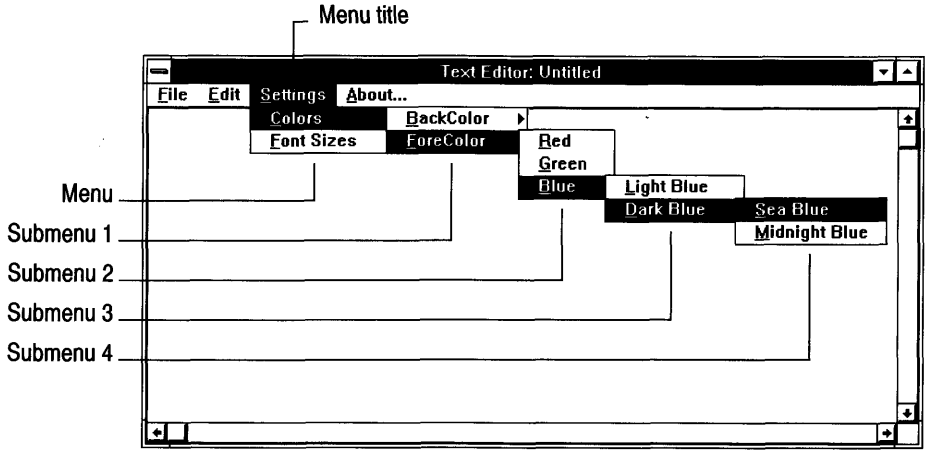


Figure 4.4 A menu interface with four levels of submenus

Figure 4.5 displays the same submenus individually. Notice that all menu controls that display submenus have an arrowhead symbol at their right edge. Visual Basic provides this visual cue automatically.

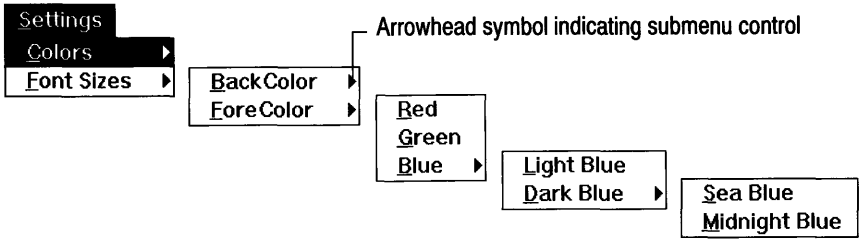


Figure 4.5 Visual cues indicating submenus

In the Menu Design window, any menu control indented below a menu control that is *not* a menu title is a *submenu control.* In general, submenu controls can include submenu items, separator bars, and submenu titles. The fourth-level submenu can include submenu items and separator bars, but not submenu titles. Figure 4.6 shows how the submenu titles and submenu items from the previous example are indented in the menu control list box in the Menu Design window.

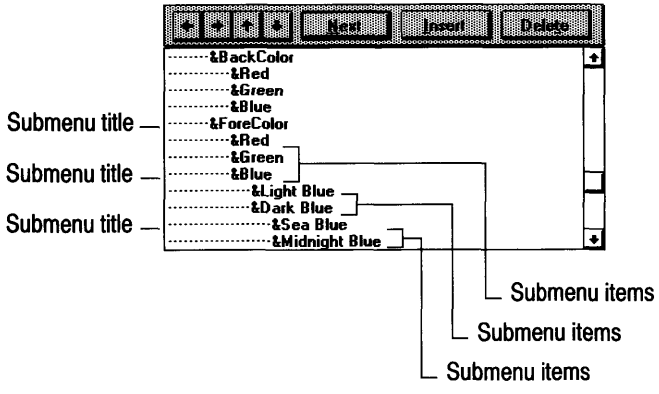


Figure 4.6 Submenu titles and submenu items in the menu control list box

▶ **To create a submenu**

1. Create the menu item that you want to be the submenu title.
2. Create the items that will appear on the new submenu, and indent them by choosing the right arrow button.

   Each indent level is preceded by four dots (....) in the Menu Design window. To remove one level of indentation, use the left arrow button.

Note If you're considering submenus, think about using a dialog box instead. Dialog boxes allow users to specify several choices in one place. For information on using dialog boxes, see the section "Working with Dialog Boxes" later in this chapter.

Separating Menu Controls

A separator bar is displayed as a horizontal line between items on a menu. On a menu with many items, you can use a separator bar to divide items into logical groups. For example, the Help menu in Visual Basic uses separator bars to divide its menu items into three groups, as shown in Figure 4.7.

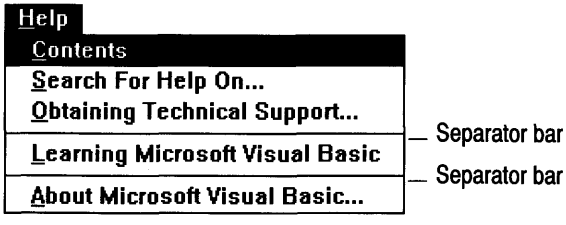


Figure 4.7 Separator bars

▶ **To create a separator bar in the Menu Design window**

1. If you are adding a separator bar to an existing menu, choose the Insert button to insert a menu control between the menu items you want to separate.
2. If necessary, click the right arrow button to indent the new menu item to the same level as the menu items it will separate.
3. Type a hyphen (-) in the Caption text box.
4. Set the Name property.
5. Choose OK to close the Menu Design window.

Note Although separator bars are created as menu controls, they do not respond to the Click event, and users cannot choose them.

Assigning Access Keys and Shortcut Keys

You can improve keyboard access to menu controls by defining access keys and shortcut keys.

Access Keys

Access keys allow the user to open a menu by pressing the ALT key and typing a designated letter. Once a menu is open, the user can choose a control by pressing the letter (the access key) assigned to it. For example, ALT+E might open the Edit menu, and P might select the Paste menu item. Access-key assignments appear as an underlined letter in the menu control's caption, as shown in Figure 4.8.

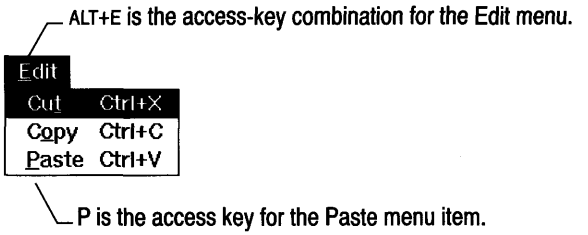


Figure 4.8 Access keys

▶ To assign an access key to a menu control in the Menu Design window

1. Select the menu item to which you want to assign an access key.
2. In the Caption box, type an ampersand (&) in front of the letter you want to be the access key.

For example, if the Edit menu shown in Figure 4.8 is open, the following Caption property settings respond to the corresponding keys.

Menu control caption	Caption property	Access keys
Cut	Cu&t	t
Copy	C&opy	o
Paste	&Paste	p

Shortcut Keys

Shortcut keys execute a menu item immediately when pressed. They include function-key and control-key combinations, such as CTRL+F1 or CTRL+A. Shortcut key assignments appear on the menu to the right of the corresponding menu item, as shown in Figure 4.9.

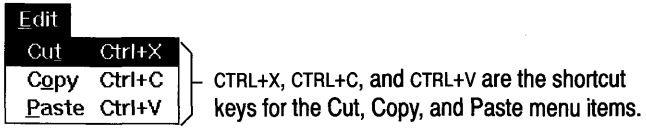


Figure 4.9 Shortcut keys

▶ **To assign a shortcut key to a menu item**

1. Open the Menu Design window.
2. Select the menu item.
3. Select a function key or key combination in the Shortcut combo box.

   To remove a shortcut key assignment, choose "(none)" from the top of the list.

Controlling Menus at Run Time

The menus you create at design time can also respond dynamically to run-time conditions. For example, if a menu item action becomes inappropriate at some point, you can prevent users from selecting that menu item by *disabling* it. In the Text Editor application, for example, if no files are open and there is no current file to close, the Close menu item is dimmed on the File menu at startup, and users cannot select it.

You can also program your application to use a check mark to indicate which of several commands was last selected. The Fonts menu item from the Text Editor application displays a submenu with a check mark next to the last font size selected. Other menu control features described in this section include code that makes a menu item visible or invisible and that adds or deletes menu items.

Enabling and Disabling Menu Commands

All menu controls have an Enabled property, and when this property is set to **False**, the menu is disabled and does not respond to user actions. A disabled menu control appears gray, like the Close menu item in Figure 4.10.

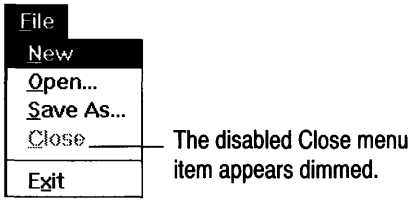


Figure 4.10 A disabled menu item

For example, this statement disables the Close menu item on the File menu of the Text Editor application:

```
mnuFileItem(3).Enabled = False
```

Disabling a menu title in effect disables the entire menu, since the user cannot access any menu item without first clicking the menu title. For example, the following code disables the File menu of the Text Editor application:

```
mnuFile.Enabled = False
```

Displaying a Check Mark on a Menu Control

Using the Checked property, you can place a check mark on a menu to:

- Tell the user the status of an on/off condition. Choosing the menu command alternately adds and removes the check mark.
- Indicate which of several modes is in effect. The Settings menu of the Text Editor application uses a check mark to indicate the current font size, as shown in Figure 4.11.

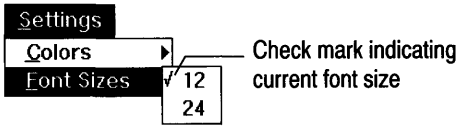


Figure 4.11 A checked menu item

You create check marks in Visual Basic with the Checked property. Set the initial value of the Checked property in the Menu Design window by selecting the check box labeled Checked. To add or remove a check mark from a menu control at run time, set its Checked property from code. For example:

```
Sub mnuFontSizesItem_Click (Index As Integer)
' Perform action based on Index property value of menu control.
    Select Case Index
    Case 0
        ' If Index = 0, then user chose font size 12.
        ' Set FontSize property of text box to 12.
        txtEdit.FontSize = 12
        mnuFontSizesItem(0).Checked = True          ' Display check mark.
        mnuFontSizesItem(1).Checked = False      ' Remove check mark.
    Case 1
        ' Set FontSize property to 24.
        txtEdit.FontSize = 24
        mnuFontSizesItem(0).Checked = False      ' Remove check mark.
        mnuFontSizesItem(1).Checked = True          ' Display check mark.
    End Select
End Sub
```

You can also set the Checked property by eliminating the submenu and changing the Caption property of the menu item. For example:

```
Sub mnuSettingsItem (Index as Integer)
    Select Case
        .
        .
        .
        Case 1
            If mnuSettingsItem(1).Caption = "Font Size 12" Then
                mnuSettingsItem(1).Caption = "Font Size 24"
            Else
                mnuSettingsItem(1).Caption = "Font Size 12"
            End If
        .
        .
        .
    End Select
End Sub
```

Making Menu Controls Invisible

In the Menu Design window, you set the initial value of the Visible property for a menu control by selecting the check box labeled Visible. To make a menu control visible or invisible at run time, set its Visible property from code. For example:

```
mnuFileArray(0).Visible = True          ' Make the control visible.

mnuFileArray(0).Visible = False         ' Make the control invisible.
```

When a menu control is invisible, the rest of the controls in the menu move up to fill the empty space. If the control is on the menu bar, the rest of the controls on the menu bar move left to fill the space.

Note Making a menu control invisible effectively disables it, since the control is inaccessible from the menu. If the menu title is invisible, all the controls on that menu are unavailable.

Adding Menu Controls at Run Time

A menu can grow at run time. In Figure 4.12, for example, as files are opened in the Text Editor application, menu items are dynamically created to display the path names of the opened files.

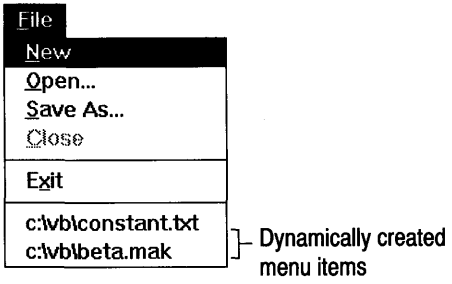


Figure 4.12 Menu control array elements created and displayed at run time

You must use a control array to create a control at run time. Because the mnuFileArray menu control is assigned a value for the Index property at design time, it automatically becomes an element of a control array—even though no other elements have yet been created.

Each time you open a file at run time, additional menu controls are loaded into the array, making the menu grow. The Text Editor application handles this in the UpdateMenu general procedure:

```
Sub UpdateMenu ()
    ' Make the initial element visible and display separator bar.
    frmEditor.mnuFileArray(0).Visible = True
    ' Increment Index property of control array.
    Index = Index + 1
    ' Create a new menu control.
    Load frmEditor.mnuFileArray(Index)
    ' Set the caption of the new menu item.
    frmEditor.mnuFileArray(Index).Caption = FileName
    ' Make the new menu item visible.
    frmEditor.mnuFileArray(Index).Visible = True
End Sub
```

Controls created at run time can be hidden by using the **Hide** method or setting the control's Visible property to **False**. If you want to remove a control in a control array from memory, use the **Unload** statement. For example, the following code deletes the last menu item created in the UpdateMenu procedure:

```
Unload frmEditor.mnuFileArray(Index)
```

Displaying Pop-up Menus

A pop-up menu is a menu that is displayed over a form, independent of the menu bar, as shown in Figure 4.13.

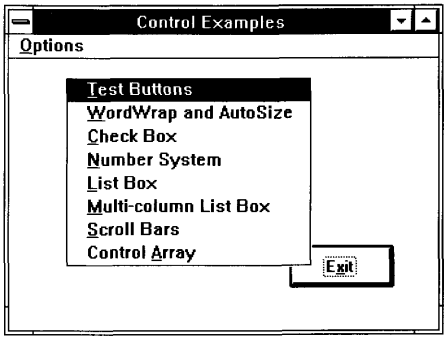


Figure 4.13 A pop-up menu

Any menu that has at least one submenu can be displayed at run time as a pop-up menu. To display a pop-up menu, use the **PopupMenu** method. This method uses the following syntax:

[*form.*]**PopupMenu** *MenuName* [, *Flags* [,*X* [, *Y*]]]

For example, the following code displays a menu named mnuFile when the user clicks a form with the right mouse button:

```
Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button = 2 Then      ' Check if right mouse button was clicked.
        PopupMenu mnuFile   ' Display the file menu as a pop-up.
    End If
End Sub
```

Some applications that implement pop-up menus display the pop-up menu when the user clicks the right mouse button. As the preceding example shows, you can use the MouseUp or MouseDown event to detect when the user clicks the right mouse button.

Only one pop-up menu can be displayed at a time. While a pop-up menu is displayed, calls to the **PopupMenu** method are ignored. Calls to the **PopupMenu** method are also ignored whenever a menu control is active.

Any code following a call to the **PopupMenu** method is not run until the user selects an item in the menu or cancels the menu.

Often you want to use as a pop-up a menu that is not normally available on the menu bar. You do this by making the menu you want to use as a pop-up invisible at design time (make sure the Visible check box in the Menu Design window is not checked). When Visual Basic displays a pop-up menu, the Visible property of the specified menu is ignored.

There are three optional arguments to the **PopupMenu** method: *X*, *Y*, and *Flags*. You use the *X* and *Y* arguments to specify the X and Y coordinates where the pop-up menu is to be displayed (coordinates use the scale mode of the specified form). The X and Y coordinates define where the pop-up is displayed, relative to the specified form. For example, the following code displays a pop-up menu in the upper-left corner of Form2 when the user clicks Form1:

```
' User clicks Form1.
Sub Form_Click ()
    ' Display pop-up menu in upper-left corner of Form2.
    Form2.PopupMenu mnuAction, POPUPMENU_LEFTALIGN, 0, 0
End Sub
```

If the X and Y coordinates are not included, the pop-up menu is displayed at the current location of the mouse pointer.

You use the *Flags* parameter to further define the location and behavior of a pop-up menu. The following table lists the flags available to describe a pop-up's location.

Location constants	Value	Description
POPUPMENU_LEFTALIGN	0 (default)	The specified *X* location defines the left edge of the pop-up menu.
POPUPMENU_CENTERALIGN	4	The pop-up menu is centered around the specified *X* location.
POPUPMENU_RIGHTALIGN	8	The specified *X* location defines the right edge of the pop-up menu.

The following table lists the flags available to describe a pop-up's behavior.

Behavior constants	Value	Description
POPUPMENU_LEFTBUTTON	0 (default)	The pop-up menu recognizes when the user clicks a menu item with the left mouse button only.
POPUPMENU_RIGHTBUTTON	2	The pop-up menu recognizes when the user clicks a menu item with either the right or left mouse button.

Note The *Flags* argument has no effect on applications running under Microsoft Windows version 3.0.

To specify a flag, you combine one constant from each group using the **Or** operator. The following code displays a pop-up menu with its top border centered in a form when the user clicks a command button. The pop-up menu triggers Click events for menu items that are clicked with either the right or left mouse button.

```
Sub Command1_Click ()
    ' Define constants.
    Const POPUPMENU_CENTERALIGN = 4
    Const POPUPMENU_RIGHTBUTTON = 2
    ' Dimension X and Y variables.
    Dim xloc, yloc

    ' Set X and Y variables to center of form.
    xloc = ScaleWidth / 2
    yloc = ScaleHeight / 2

    ' Display the pop-up menu.
    PopupMenu mnuEdit, POPUPMENU_CENTERALIGN Or
➥   POPUPMENU_RIGHTBUTTON, xloc, yloc
End Sub
```

Note The constants used in this chapter are contained in the file CONSTANT.TXT, located in the main Visual Basic directory. You can use these constants in your code if you load this file into any of your application's code modules.

Working with Dialog Boxes

In Windows–based applications, dialog boxes are used to:

- Prompt the user for data needed by the application to continue.
- Display information to the user.

In Visual Basic, for example, you use the Open Project dialog box to display an existing project. An example of a dialog box used to convey information is the About dialog from the Text Editor application. When the user clicks the About menu title on the title bar, the dialog is displayed (see Figure 4.14).

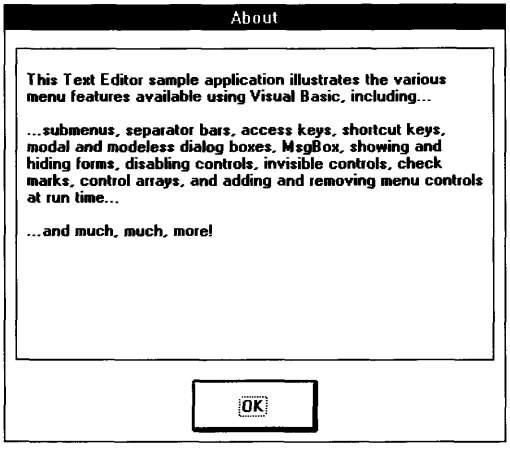


Figure 4.14 The About dialog box from the Text Editor application

Modal and Modeless Dialog Boxes

Dialog boxes are either modal or modeless. A *modal* dialog box must be closed (hidden or unloaded) before you can continue working with the rest of the application. For example, a dialog box is modal if it requires you to click OK or Cancel before you can switch to another form or dialog box.

The About dialog in Figure 4.14 is a modal dialog. Dialog boxes that display important messages should always be modal—that is, the user should always be required to close the dialog box or respond to its message before proceeding.

Modeless dialog boxes let you shift the focus between the dialog box and another form without having to close the dialog box. You can continue to work elsewhere in the current application while the dialog box is displayed. Use modeless dialog boxes to display frequently used commands or information. The Visual Basic Project window, shown in Figure 4.15, is a modeless dialog box.

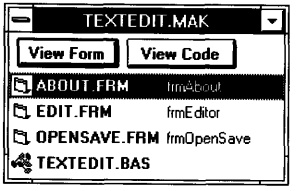


Figure 4.15 You can shift the focus from a modeless dialog box without closing it.

▶ To display a modal dialog box

- Use the **Show** method with a *style* argument of 1.

  For example:

```
' Display frmAbout as a modal dialog.
frmAbout.Show 1
```

▶ To display a modeless dialog box

- Use the **Show** method without a *style* argument.

  For example:

```
' Display frmAbout as a modeless dialog.
frmAbout.Show
```

Using Predefined Dialog Boxes

The easiest way to add a dialog box to your application is to use a predefined dialog box. With a predefined dialog box, you don't have to worry about designing, loading, or showing the dialog box; however, your control over its appearance is limited. Predefined dialog boxes are always modal.

The following table lists the methods for adding predefined dialog boxes to your Visual Basic application.

Use this	To do this
InputBox function	Display a prompt in a dialog box, and return the text entered by the user.
MsgBox statement	Display a message in a dialog box.
MsgBox function	Display a message in a dialog box, and return a value indicating the command button clicked by the user.

Prompting for Input with InputBox

Use the **InputBox** function to solicit text data from the user. The **InputBox** function displays a modal dialog box that asks the user to enter a string or **Variant**. The text input box shown in Figure 4.16 prompts the user for the name of the file to open.

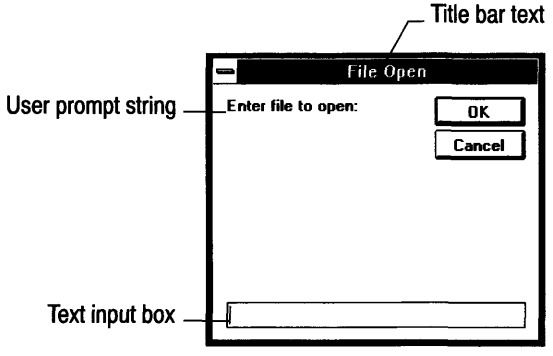


Figure 4.16 A File Open dialog using the InputBox function

The following code displays the input box shown in Figure 4.16:

```
FileName = InputBox("Enter file to open:", "File Open")
```

Note Remember that when you use the **InputBox** function, you have little control over the components of the dialog box. You can change only the text in the title bar, the prompt displayed to the user, and the position of the dialog box on the screen.

For More Information For information on **InputBox** syntax, see the *Language Reference*, or search Help for *InputBox*.

Displaying Information with MsgBox

Use the **MsgBox** statement to display brief messages such as errors, warnings, or alerts in a dialog box. After reading the message, the user chooses a button to close the dialog box.

The Text Editor application displays the message dialog box shown in Figure 4.17 if a file cannot be opened.

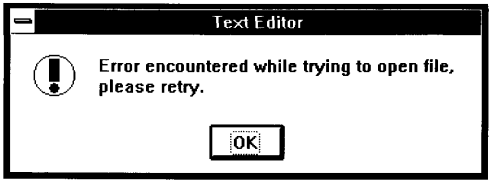


Figure 4.17 An error message dialog box created using the MsgBox statement

The following code displays the message box shown in Figure 4.17:

```
MsgBox "Error encountered while trying to open file, please retry.",
➥ 48, "Text Editor"
```

Note Use the **MsgBox** function to return a value indicating which button the user chose.

For More Information For information on the **MsgBox** statement and the **MsgBox** function, see the *Language Reference*, or search Help for *MsgBox*.

Creating a Custom Dialog Box

A *custom dialog box* is a form you create containing controls—including command buttons, option buttons, and text boxes—that let the user supply information to the application. You customize the appearance of the form by setting property values. You also write code to display the dialog box at run time.

To create a custom dialog box, you can start with a new form or customize an existing dialog box. Over time, you can build up a collection of dialog boxes that can be used in many applications.

▶ **To customize an existing dialog box**

1. From the File menu, choose Add File to add an existing form to your project.
2. From the File menu, choose Save File As and enter a new file name.
3. Customize the appearance of the form as desired.
4. Customize event procedures in the Code window.

You have considerable freedom to define the appearance of a custom dialog box. It can be fixed or movable, modal or modeless. It can contain different types of controls; however, dialog boxes do not normally include menu bars, window scroll bars, Minimize and Maximize buttons, status bars, or sizable borders. The remainder of this section discusses typical dialog box styles.

▶ **To create a new dialog box**

1. From the File menu, choose New Form.

   –or–

   Choose the New Form button on the toolbar to create a new form.
2. Customize the appearance of the form as desired.
3. Customize event procedures in the Code window.

Adding a Title

A dialog box should always have a title that identifies it. To create a title, set the form's Caption property to the text string that will appear in the title bar. Normally, this is done at design time using the Properties window, but it can also be done from code. For example:

```
frmAbout.Caption = "About"
```

Tip If you want to remove the title bar completely, set the form's ControlBox, MinButton, and MaxButton properties to **False**; its BorderStyle to a nonsizable setting (0, 1, or 3); and its Caption equal to an empty string ("").

Setting Standard Dialog Box Properties

Generally, the user responds to a dialog box by providing information and then closing the dialog box with an OK or Cancel command button. Because a dialog box is temporary, users normally don't need to move, size, maximize, or minimize it. As a result, the sizable border style, Control-menu box, Maximize button, and Minimize button that come with a new form are unnecessary on most dialog boxes.

You can remove these items by setting the BorderStyle, ControlBox, MaxButton, and MinButton properties. For example, the Text Editor application uses the following property settings.

Property	Setting	Effect
BorderStyle	1	Changes the border style to fixed single, thus preventing the dialog box from being sized at run time.
ControlBox	**False**	Removes the Control-menu box.
MaxButton	**False**	Removes the MaxButton, thus preventing the dialog box from being maximized at run time.
MinButton	**False**	Removes the MinButton, thus preventing the dialog box from being minimized at run time.

Remember that if you remove the Control-menu box (ControlBox = **False**), you must provide the user with another way to exit the dialog box. This is commonly done by adding an OK, Cancel, or Exit command button to the dialog box and adding code in the Click event for the button that hides or unloads the dialog.

Adding and Placing Command Buttons

Modal dialog boxes must contain at least one command button to exit the dialog box. To draw a command button on a form, use the Toolbox. (If you need information about a specific control, see Chapter 3, "Creating and Using Controls," or search Help for *Controls*.)

Normally, two command buttons are used: one to let the user start an action, and one to close the dialog box without making any changes. Typically, the Caption property settings for these buttons are OK and Cancel. In this scenario, the OK command button has its Default property set to **True** and the Cancel button has its Cancel property set to **True**. Although OK and Cancel are the most commonly used buttons, other button caption combinations work, including those shown in Figure 4.18.

Figure 4.18 Some command button caption combinations

Dialog boxes that display messages usually use a label control to display the error message or prompt, and one or two command buttons to perform an action. For example, you might assign the error message or prompt to the Caption property of the label, and Yes and No to the Caption property of two command button controls. When users choose Yes, one action takes place; when they choose No, another action occurs.

Command buttons on this type of dialog are usually placed on the bottom or right side of the dialog box, with the top or left button being the default button, as shown in Figure 4.19.

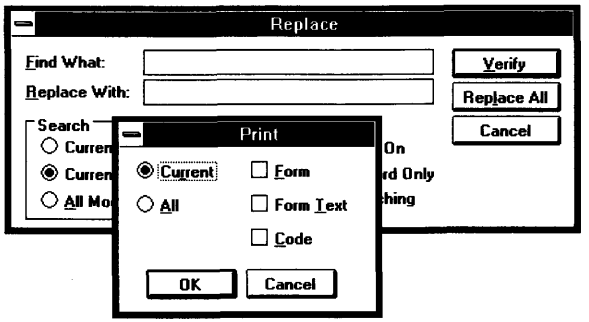


Figure 4.19 Command button placement on dialog boxes

A command button or menu can display another dialog box. For example, when you choose the Add command button in the Edit Watch dialog box, Visual Basic displays the Add Watch dialog box, as shown in Figure 4.20.

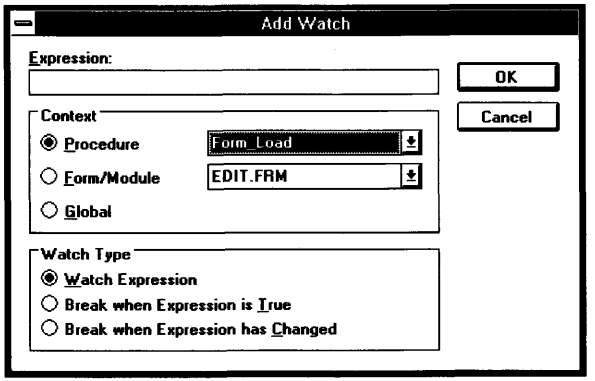


Figure 4.20 The Add Watch dialog box

Setting Default, Cancel, and Focus

Command button controls provide the following properties:

- Default
- Cancel
- Focus

The Default button is selected when the user presses ENTER. Only one command button on a form can have its Default property set to **True**. Pressing the ENTER key invokes the Click event for the default command button. This feature works in conjunction with an edit control such as a text box. For example, the user can type data in a text box and then press ENTER to generate a Click event instead of choosing an OK button.

▶ **To specify the default command button for a dialog box**

- Set the command button's Default property to **True**.

The Cancel button is selected when the user presses ESC. Only one command button on a form can have its Cancel property set to **True**. Pressing the ESC key invokes the Click event for the Cancel command button. The Cancel button can also be the default command button.

▶ **To specify the Cancel button for a dialog box**

- Set the command button's Cancel property to **True**.

Tip In general, the button that indicates the most likely or safest action should be the default action. For example, in a Text Replace dialog box, Cancel should be the default button, not Replace All.

You can also specify the button that will have the focus when the dialog is displayed. The control with the smallest TabIndex setting receives the focus when the form is displayed. Pressing the ENTER key invokes the Click event for the default command button, or the command button that has the focus when Click is invoked.

▶ **To give a command button the focus when the form is displayed**

- Set the command button's TabIndex property to 0 and its TabStop property to **True**.

For More Information The **SetFocus** method can also be used to give a specific control the focus when a form is displayed. For information on the **SetFocus** method, see the *Language Reference*, or search Help for *SetFocus*.

Disabling Controls on a Dialog

You can disable a control on a dialog by setting its Enabled property to **False**. For example, the Edit, Delete, and Delete All buttons on the Watch dialog, shown in Figure 4.21, are initially disabled because there are no existing watch variables. The user can choose only Add, to add a watch variable, or Close, to close the dialog.

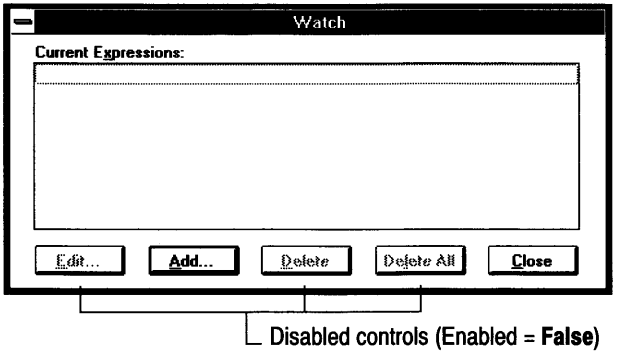


Figure 4.21 Disabled controls on a dialog

▶ **To disable a control on a dialog**

- Set each control's Enabled property to **False** with the following statements:

```
cmdWatchEdit.Enabled = False
cmdWatchDelete.Enabled = False
cmdWatchDeleteAll.Enabled = False
```

Displaying a Custom Dialog Box

You display a dialog box in the same way you display any other form in an application. The startup form loads automatically when the application is run. When you want a second form or dialog box to appear in the application, you write code to load and display it. Similarly, when you want the form or dialog box to disappear, you write code to unload it or hide it.

The following code from the Text Editor application displays the About dialog box when the user chooses the About menu:

```
Sub mnuAbout_Click ()
' The Show method with style = 1 is used here to display the dialog as
' modal. Unloading the dialog is handled in the forms cmdOK_Click event
' procedure.
    frmAbout.Show 1
End Sub
```

Note For details on writing code for specific controls, see Chapter 3, "Creating and Using Controls." For details on creating a dialog box containing file, directory, and drive list boxes, see Chapter 18, "Using the File-System Controls."

Display Options

The code you write determines how the dialog box is loaded into memory and displayed. The following table describes various form-display tasks and the keywords that are used to perform them.

Task	Keyword
Load a form into memory, but do not display it.	Use the **Load** statement, or reference a property or control on the form.
Load and display a modeless form.	Use the **Show** method.
Load and display a modal form.	Use the **Show** method with *style* = 1.
Display a loaded form.	Set its Visible property to **True**, or use the **Show** method.
Hide a form from view.	Set its Visible property to **False**, or use the **Hide** method.
Hide a form from view and unload from memory.	Use the **Unload** statement.

The **Show** method loads the form and sets its Visible property to **True**. The argument passed to the **Show** method indicates the style of the dialog box. If the argument is 0 (default), the dialog box is modeless; if it is 1, the dialog box is modal.

Important When you show a dialog box as modal, the next line of your code is not executed until the form is closed (hidden or unloaded). This is not true if you show a modeless dialog box. Because you usually need to obtain the values entered by the user in the dialog box, command button Click event procedures on a modal form will often hide—rather than unload—a modal form. The code that displays the dialog box can also access dialog box information entered by the user.

To exit the dialog box when the user chooses OK or Cancel, use either the **Unload** statement or the **Hide** method. For example:

```
Unload frmAbout

frmAbout.Hide
```

The **Unload** statement removes the dialog box from memory, while the **Hide** method merely removes the dialog box from view by setting its Visible property to **False**. When you unload a form, the form itself *and* its controls are unloaded from memory (including any controls that were loaded at run time). When you hide a form, the form and its controls remain in memory.

When you need to save space in memory, it's better to unload a form, since unloading a form frees memory. If the dialog box is used often, however, it makes sense to hide the form. Hiding a form retains any data attached to it, including property values, print output, and dynamically created controls. This way you can continue to refer to the properties and controls of a hidden form in code.

Using the Common Dialog Custom Control

The common dialog custom control allows you to display these commonly used dialog boxes:

- Open
- Save As
- Print
- Color
- Font

For more information on adding a custom control to a project, see Chapter 5, "Managing Projects."

▶ **To use the common dialog custom control**

1. Select the common dialog control in the Toolbox and draw a control on a form.

   When you draw a common dialog control on a form, it automatically resizes itself. Like the timer control, the common dialog control is invisible at run time.

2. At run time, set the control's Action property to display the desired dialog.

The following dialogs are displayed when the Action property is set at run time.

Setting	Dialog displayed
0	No action
1	Open
2	Save As
3	Color
4	Font
5	Print
6	Invokes WINHELP.EXE

Note Setting Action = 6 does not display a dialog, but invokes the Windows Help engine (WINHELP.EXE). For more information on calling WINHELP.EXE, search Help for *Help*.

Displaying Open and Save As Dialog Boxes

The Open dialog box allows the user to specify a drive, a directory, a file-name extension, and a file name. The file can also be opened for read-only operations.

The Save As dialog box is identical to the Open dialog in appearance, except for the dialog's caption. At run time, when the user chooses a file and closes the dialog, the FileName property is used to get the selected file name.

Figure 4.22 shows a typical Open dialog box.

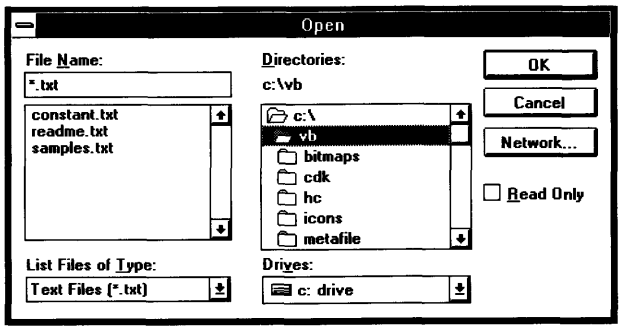


Figure 4.22 An Open dialog box

▶ To display the Open dialog box

1. Specify the list of file filters that are displayed in the "List Files of Type" list box.

   This is done using the Filter property. You set the Filter property using the following format:

   description1 | filter1 | description2 | filter2...

   Description is the string displayed in the list box—for example, "Text Files (*.txt)." *Filter* is the actual file filter—for example, "*.txt." Each *description* | *filter* set must be separated by a pipe symbol (|).

2. Display the dialog box (set Action = 1).

After the user chooses a file, use the FileName property to get the name of the selected file.

With all the common dialog boxes, when the CancelError property is **True**, an error is generated when the user clicks the dialog box's Cancel button. You detect that the Cancel button was pressed by trapping the error when the dialog box is displayed.

The following code displays a Open dialog and uses the selected file name as an argument to a procedure that opens a file:

```
Sub mnuFileOpen_Click ()
    ' CancelError is True.
    On Error GoTo ErrHandler
    ' Set filters.
    CMDialog1.Filter = "All Files (*.*)|*.*|Text Files
➥   (*.txt)|*.txt|Batch Files (*.bat)|*.bat"
    ' Specify default filter.
    CMDialog1.FilterIndex = 2

    ' Display the Open dialog box.
    CMDialog1.Action = 1
    ' Call the open file procedure.
    OpenFile (CMDialog1.FileName)

ErrHandler:
' User pressed Cancel button.
    Exit Sub
End Sub
```

Using the Color Dialog Box

The Color dialog box allows the user to select a color from a palette or to create and select a custom color. At run time, when the user chooses a color and closes the dialog box, you use the Color property to get the selected color.

Figure 4.23 shows the Color dialog box.

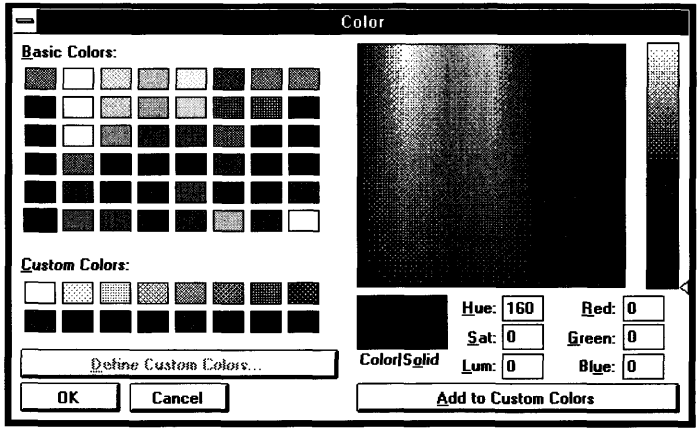


Figure 4.23 The Color dialog box

▶ To display the Color dialog box

1. Set the Flags property to the constant CC_RGBINIT (&H1&).
2. Display the dialog box (set Action = 3).

You use the Color property to get the RGB value of the color the user selects. Here's an example:

```
Sub Command1_Click ()
    ' Set Cancel to True.
    CMDialog1.CancelError = True
    On Error GoTo ErrHandler
    ' Set the RGB_INIT flag.
    CMDialog1.Flags = &H1&
    ' Display the Color dialog box.
    CMDialog1.Action = 3
    ' Set the form's background color to the selected color.
    Form1.BackColor = CMDialog1.Color

ErrHandler:
    ' User pressed Cancel button.
    Exit Sub
End Sub
```

Using the Font Dialog Box

The Font dialog box allows the user to select a font by specifying a typeface, point size, color, and style. Once the user makes selections in this dialog box, the following properties contain information about the user's selection.

Property	Determines
Color	The selected color. To use this property, you must first set the Flags property to &H100&.
FontBold	Whether **bold** was selected.
FontItalic	Whether *italic* was selected.
FontStrikethru	Whether ~~strikethru~~ was selected.
FontUnderline	Whether underline was selected.
FontName	The selected font name.
FontSize	The selected font size.

Figure 4.24 shows the Font dialog box.

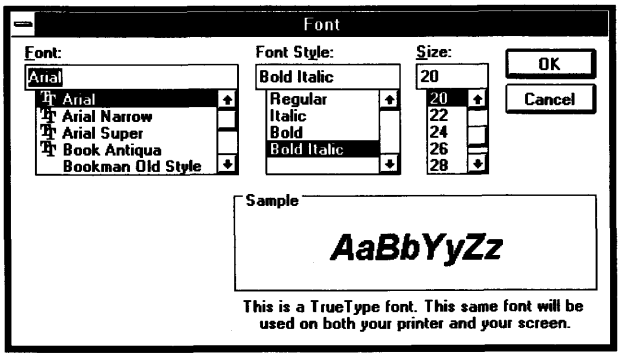


Figure 4.24 The Font dialog box

▶ To display the Font dialog box

1. Set the Flags property to one of the following constants:
   - CF_SCREENFONTS (&H1&)
   - CF_PRINTERFONTS (&H2&)
   - CF_BOTH (&H3&)

Caution You must set the Flags property to one of these values before displaying the Font dialog box. Otherwise, the error `No Fonts Exist` occurs.

---

2. Display the dialog box (set Action = 4).

The following code sets the font properties for a text box based on a user's selections in the Font dialog box:

```
Sub Command1_Click ()
    ' Set Cancel to True.
    CMDialog1.CancelError = True
    On Error GoTo ErrHandler
    ' Set the CF_BOTH and CF_EFFECTS flags.
    CMDialog1.Flags = &H3& Or &H100&
    ' Display the Font dialog box.
    CMDialog1.Action = 4
    ' Set text properties according to user's selections.
    Text1.FontName          = CMDialog1.FontName
    Text1.FontSize          = CMDialog1.FontSize
    Text1.FontBold          = CMDialog1.FontBold
    Text1.FontItalic        = CMDialog1.FontItalic
    Text1.FontUnderline     = CMDialog1.FontUnderline
    Text1.FontStrikethru    = CMDialog1.FontStrikethru
    Text1.ForeColor         = CMDialog1.Color

ErrHandler:
    ' User pressed Cancel button.
    Exit Sub
End Sub
```

Using the Print Dialog Box

The Print dialog box allows the user to specify how output should be printed. The user can specify a range of pages to be printed, a print quality, a number of copies, and so on. The dialog box also displays information about the currently installed printer and allows the user to configure or reinstall a new default printer.

Note This dialog box does not actually send data to a printer. It allows users to specify how they want data printed. You must write code to print the data in the format they select.

For more information on printing data, see Chapter 16, "Displaying and Printing Information."

At run time, when the user makes selections in this dialog box, the following properties contain information about the user's selection.

Property	Determines
Copies	The number of copies to print
FromPage	The page to start printing
ToPage	The page to stop printing
hDC	The device context for the selected printer

Figure 4.25 shows the Print dialog box.

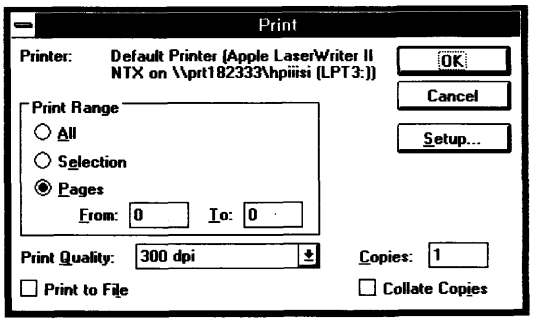


Figure 4.25 The Print dialog box

▶ To display the Print dialog box

1. Set any desired default settings for the dialog by setting the appropriate Print dialog properties.

   For example, to display 2 in the Copies text box when the dialog is displayed, set the Copies property to 2:

   ```
   CMDialog1.Copies = 2
   ```

2. Display the dialog box (set Action = 5).

The following code displays the Print dialog box:

```
Sub Command1_Click ()
    Dim BeginPage, EndPage, NumCopies, i

    ' Set Cancel to True.
    CMDialog1.CancelError = True
    On Error GoTo ErrHandler

    ' Display the Print dialog box.
    CMDialog1.Action = 5

    ' Get user-selected values from the dialog box.
    BeginPage   = CMDialog1.FromPage
    EndPage     = CMDialog1.ToPage
    NumCopies   = CMDialog1.Copies

    For i = 1 to NumCopies
        ' Put code here to send data to your printer.
    Next

ErrHandler:
    ' User pressed Cancel button.
    Exit Sub
End Sub
```

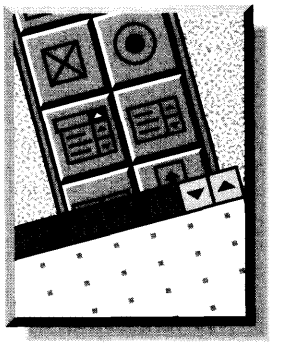
Note If the PrinterDefault property is set to **True**, you can print to Visual Basic's Printer object. In addition, when this property is **True,** any changes the user makes in the Setup portion of the Print dialog box are used to change the printer settings in the user's WIN.INI file .

For More Information For information on using common dialog boxes and a list of properties each dialog supports, search Help for *Common Dialog*.

Note The constants used in this chapter are contained in the file CONSTANT.TXT, located in the main Visual Basic directory. You can use these constants in your code if you load this file into any of your application's code modules.

CHAPTER 5

Managing Projects



To create an application with Visual Basic, you work with projects. A *project* is the collection of files you use to build an application. This chapter describes projects and how you build and manage them.

Contents

- Understanding Projects
- Creating, Opening, and Saving Projects
- Creating New Forms and Code Modules
- Adding, Removing, and Saving Files
- Making and Running an Executable File
- Setting Environment Options
- Editing the AUTOLOAD.MAK File

Understanding Projects

As you develop an application, you use a project to manage all the different files you create. A project consists of:

- One file for each form (.FRM)
- One file for each code module (.BAS)
- One file for each custom control (.VBX)
- One project file that keeps track of all the components (.MAK)

The *project file* does not contain any form or code modules; instead, it lists all the files associated with a particular project, as well as information on the environment options you set. This information is updated every time you save the project.

Note A project can also include binary data files (.FRX) if you use controls that have binary properties (for example, the Picture property of an image control). For more information on binary property values, see Appendix A, "ASCII File Formats."

When you have created all the files for a project, you can convert it into an executable file (.EXE), as shown in Figure 5.1.

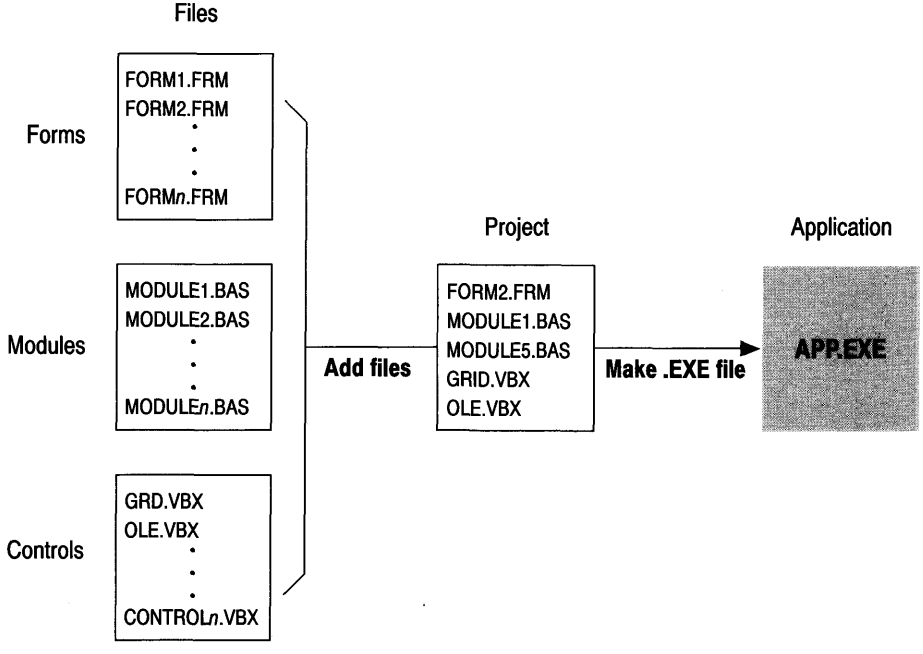


Figure 5.1 The structure of a Visual Basic application

▶ To create an application from a project

1. Create and save Visual Basic forms, modules, and custom controls.
2. Include the relevant forms, modules, and custom controls in a project.
3. Make an executable file from the project.

The Project Window

As you create, add, or remove files from a project, Visual Basic reflects your changes in the Project window, which contains a current list of the files in the project. A default project is automatically loaded when you first start Visual Basic, and every time thereafter that you create a new project. The Project window in Figure 5.2 shows the files that make up a default project.

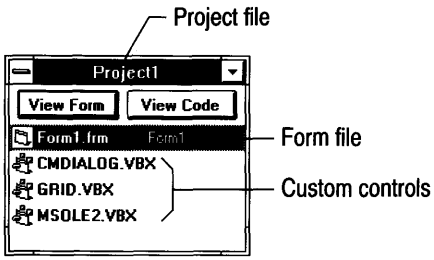


Figure 5.2 The Project Window

The Project File

Each time you save a project, Visual Basic updates a project file. Project files, which have a .MAK file-name extension, contain the same list of files that appears in the Project window. Project files also contain information about where you can find the files in your directories and how the Visual Basic environment appears when you open the project.

You can start a project file by double-clicking its icon in Windows, or at the MS-DOS prompt. If you start a project file from MS-DOS, you automatically start Visual Basic, adding the forms, modules, and custom controls listed in the project file.

▶ **To start a .MAK file from the MS-DOS prompt**

- Type **vb /run** *filename* [**/cmd** *commandline*]

  The *filename* you specify must have a .MAK extension for Visual Basic to load and run it. Any file with a different extension will be loaded into a new project as either a form or a code module. The *commandline* variable allows you to send a list of command-line arguments to the application.

In Visual Basic, project files have an ASCII (text) format. For information about the exact format of project files, see Appendix A, "ASCII File Formats."

The Elements of a Project

The following sections describe the three types of files—forms, modules, and custom controls—that you can include in a project.

Forms Forms have a .FRM file-name extension. They can contain graphical descriptions of the form and its controls, including their property settings. They can also contain form-level declarations of types, constants, variables, and external procedures; subroutines that handle events; and general procedures. For information about creating forms, see Chapter 2, "Your First Visual Basic Application," and Chapter 14, "Multiple-Document Interface (MDI) Applications."

Modules Code modules have a .BAS file-name extension. They can contain global- or module-level declarations of types, constants, variables, external procedures, and global procedures. For information about creating modules, see Chapter 6, "Programming Fundamentals," and Chapter 8, "Objects and Instances."

Custom Controls Custom controls have a .VBX file-name extension. They contain the information Visual Basic needs to provide new controls in the Toolbox. Three custom controls come with Visual Basic: the grid control (GRID.VBX), the OLE control (MSOLE2.VBX), and the common dialog control (CMDIALOG.VBX). When you install Visual Basic, these files are copied to your Windows \SYSTEM directory. When a custom control is added to a project, its icon is displayed in the Toolbox. Use the Add File command on the File menu to add a custom control to a project. You can then select the control from the Toolbox and draw it on a form as you would any other control.

The common dialog control is discussed in Chapter 4, "Menus and Dialogs." The grid control is discussed in Chapter 13, "Using the Grid Control." The OLE control is discussed in Chapter 22, "Object Linking and Embedding (OLE)."

For More Information Additional custom controls are available from third-party developers or in the Professional Edition of Visual Basic. For information about creating custom controls, see the *Control Development Guide* in the *Professional Features* manual, which comes with the Professional Edition.

Creating, Opening, and Saving Projects

Four commands on the File menu allow you to create, open, and save projects.

Menu command	Description
New Project	Closes the current project, prompting you to save any files that have changed; then creates a new project, adding a new form and any forms, modules, and custom controls listed in the AUTOLOAD.MAK file. (See the section "Editing the AUTOLOAD.MAK File" later in this chapter.)
Open Project	Closes the current project, prompting you to save any changes; then opens an existing project, including the forms, modules, and custom controls listed in its project (.MAK) file.
Save Project	Updates the project file of the current project.
Save Project As	Updates the project file of the current project, saving the project file under a file name you specify; also prompts you to save any forms or modules that have changed.

Note that although you can have only one project open at a time, you can share files between projects. A single file, such as a form, can also be part of more than one project. For more information about sharing files, see the section "Adding, Removing, and Saving Files" later in this chapter. For information on sharing files among different versions of Visual Basic, see Appendix C, "Compatibility with Other Versions."

Creating New Forms and Code Modules

Three commands on the File menu allow you to create new forms and modules.

Menu command	Description
New Form	Creates a new form and adds it to the project.
New MDI Form	Creates a new MDI form and adds it to the project.
New Module	Creates a new module and adds it to the project.

Adding, Removing, and Saving Files

Four commands on the File menu allow you to add, remove, and save files.

Menu command	Description
Add File	Adds an existing form, module, or custom control to a project.
Remove File	Removes a form, module, or custom control from a project.
Save File	Saves a form or module in a project.
Save File As	Saves a form or module in a project under a file name that you specify, allowing you to choose text or binary format.

If you *remove* a file from a project, Visual Basic updates this information in the project file when you save the project. If you delete a file outside Visual Basic, however, Visual Basic cannot update the project file; therefore, when you open the project, Visual Basic displays an error message warning you that a file is missing. To add a new or renamed file to a project, open the project and choose the Add File command from the File menu.

If you *add* a file to a project, you are simply including the existing file in the project; you are not adding a copy of the file to the project. Therefore, if you make changes to a file and save it, your changes will affect any project that includes the file. To change a file without affecting other projects, first choose Save File As from the File menu, and save the file under a new file name.

ASCII Text Format

Visual Basic allows you to add and save forms and code modules in binary or ASCII text format. You can specify a default save format in the Environment Options dialog box (see the section "Environment Options" later in this chapter). Also, Visual Basic allows you to load the code contained in forms and modules and save it as text.

If you save forms, modules, and code as text, you can:

- Evaluate the textual representation of forms.
- Use systems that manage source code.
- Generate forms using external tools of your own.

Creating Text Files

You can create text files in Visual Basic with the Save File As command.

Procedure	Contents of resulting text file
Save a form as a text file	A graphical description of—and the code contained in—the form and its controls
Save the code in a form as a text file	The code contained in the form and its controls
Save a module as a text file	The code contained in the module
Save the code in a module as a text file	The code contained in the module

Note that although you can save a module as a text file, or save the code in a module as a text file, the resulting files are identical.

Manipulating Text Files

You can add a text file to a project just as you would add any other file. Visual Basic adds the text file as a form or module according to the information it finds in the file.

▶ **To add a text file as a form or module**

1. From the File menu, choose Add File.
2. Select the name of the text file you want to load.
3. Choose the OK button.

If Visual Basic encounters problems when loading a text file as a form or module, it creates a file containing a log of error messages and displays a warning on your screen. For more information about error messages, see Appendix A, "ASCII File Formats."

▶ **To save a form or module as a text file**

1. Select the form or module.
2. From the File menu, choose Save File As.
3. Type the name of the text file.
4. Select the Save as Text check box.
5. Choose the OK button.

Visual Basic allows you to load text files as code by replacing the code of a form or module with the text file, merging the text file with the existing code of a form or module, or loading the text file into a new module. You can choose these actions from the Load Text dialog box, shown in Figure 5.3.

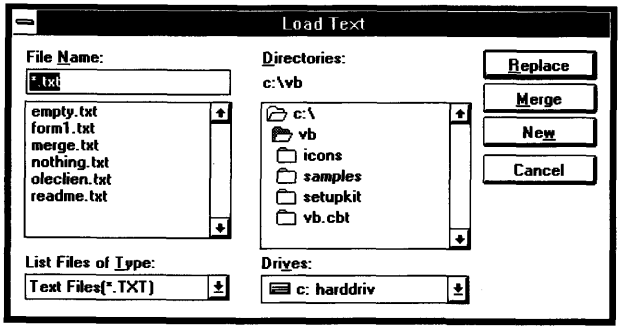


Figure 5.3 The Load Text dialog box

▶ To load a text file as code

1. If you want to replace the code in a form or module with the text file, or merge the text file with the existing code in a form or module, select the form or module.
2. From the File menu, choose Load Text.
3. Select the name of the text file you want to load.
4. To replace the code in the form or module with the text file, choose the Replace button.

   –or–

   To merge the text file with the code in the form or module, choose the Merge button.

   –or–

   To load the text file into a new module, choose the New button.

▶ To save code as a text file

1. Select the form or module that contains the code.
2. From the File menu, choose Save Text.
3. Type the name of the text file.
4. Choose the OK button.

Making and Running an Executable File

You can make an executable file from Visual Basic or from the MS-DOS prompt.

▶ **To make an executable file from Visual Basic**

1. From the File menu, choose Make EXE File.
2. Type a file name.
3. If you want to specify a new icon, in the Use Icon From box, type an icon name or choose one from the list.
4. If you want to specify a new name for the application, type a new name in the Application Title box.
5. Choose the OK button.

Building an executable file from the MS-DOS command line can be a useful option when you want to compile a large project programmatically.

▶ **To make an executable file from the MS-DOS prompt**

- Type **vb /make** *projectname*[**.mak**] [*exename*]

  Note that *projectname* is the name of the project file from which Visual Basic makes the executable file. The variable *exename* can be used to give a different name to the resulting executable file.

You can run the executable file like any other Windows–based application, by either creating a program item in Windows or typing a command line from the MS-DOS prompt.

Setting Environment Options

From Visual Basic, you can set environment and project options. Visual Basic saves the settings of environment options to the VB.INI file and saves the settings of project options to the AUTOLOAD.MAK file.

Environment Options

From the Options menu, choose Environment to modify any of the settings in the Environment Options dialog box, shown in Figure 5.4.

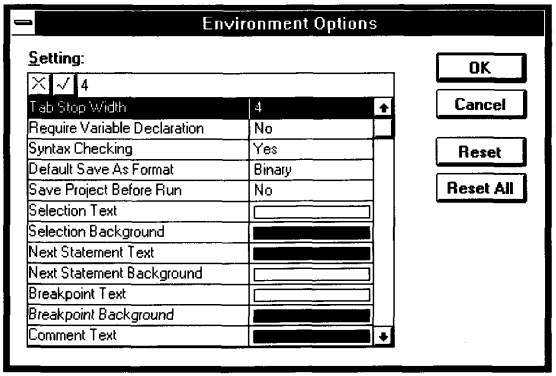


Figure 5.4 The Environment Options dialog box

The following table lists the available settings.

Options	Possible settings
The text and background colors used for comments, keywords, breakpoints, executing statements, and debugging	Black, white, and light and dark shades of blue, green, cyan, red, purple, yellow, and gray
Tab stop width	1 to 32
Whether or not to require variable declarations, check syntax, or show and align the grid	Yes or No
Grid height and width	45 to 1,485 twips
Default Save As format	Binary or text
Save project on run	Yes or No

Project Options

From the Options menu, choose Project to modify any of the settings in the Project Options dialog box, shown in Figure 5.5.

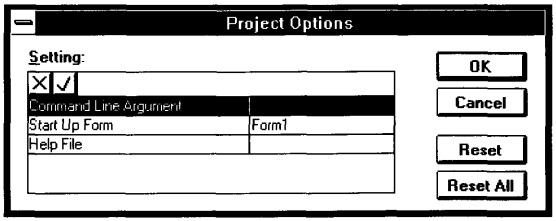


Figure 5.5 The Project Options dialog box

The following table describes the options you can set.

Options	Description
Command Line Argument	The command-line argument that Visual Basic sends to an application (sent only when you choose Start from the Run menu).
Start Up Form	The first form that Visual Basic displays at run time, or **Sub Main ( )**.
Help File	The name of a Help file you want to link to the application.

Editing the AUTOLOAD.MAK File

When you create a new project, Visual Basic automatically adds the files listed in the AUTOLOAD.MAK file. By default, the startup file includes the custom controls GRID.VBX and MSOLE2.VBX; therefore, Visual Basic automatically adds the grid and OLE controls to new projects. In addition to specifying which files Visual Basic adds, the AUTOLOAD.MAK file also specifies how the Visual Basic environment appears when you open new projects.

You can edit the AUTOLOAD.MAK file as you would edit any project file.

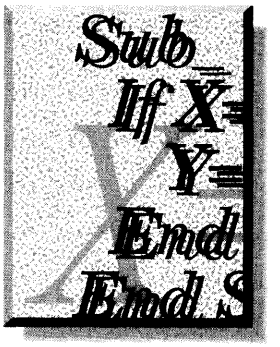
▶ **To edit the AUTOLOAD.MAK file in Visual Basic**

1. From the File menu, choose Open Project.
2. Select AUTOLOAD.MAK.
3. Choose the OK button.
4. For each file you want to remove, select it in the Project window and then choose Remove File from the File menu.
5. From the Options menu, choose Environment to set the options you want to appear when you open new projects.
6. From the File menu, choose Save Project.

Note Visual Basic will read the AUTOLOAD.MAK file only if it appears in the same directory as Visual Basic. If Visual Basic cannot find the AUTOLOAD.MAK file, or if you delete the file, Visual Basic will add a single, new form to new projects.

CHAPTER 6

Programming Fundamentals



Visual Basic is a complete programming language that supports the structured programming constructs found in most other modern programming languages. This chapter introduces the essential components of the Visual Basic language.

Contents

- Structure of a Visual Basic Application
- Procedures
- Code in Procedures
- Controlling Execution

Many of these components are presented with *syntax*, using typographic conventions that illustrate the correct form of the statement or function. The same conventions are used throughout the rest of this manual, in Help, and in the *Language Reference*. For an explanation of these conventions, see "Document Conventions" at the beginning of this manual.

Structure of a Visual Basic Application

Your application can contain form modules (.FRM) and code modules (.BAS), as well as custom controls (.VBX). A form module contains the visual elements of a form, including all the controls on the form and the Basic code associated with that form. A code module contains only Basic code. You must have at least one code or form module. By default, your project contains a single form module, but you can add additional form and code modules. For more information on creating new form and code modules, and adding existing form and code modules to a new project, see Chapter 5, "Managing Projects."

How an Event-Driven Application Works

An *event* is an action recognized by a form or control. Event-driven applications execute Basic code in response to an event. Each form and control in Visual Basic has a predefined set of events. If one of these events occurs, Visual Basic invokes the code in the associated event procedure.

Although objects in Visual Basic automatically *recognize* a predefined set of events, you determine if and how they *respond* to a particular event. When you want a control to respond to an event, you write code called an *event procedure* for that event.

Many objects recognize the same event, although different objects can execute different event procedures when the event occurs. For example, a Click event occurs when a user clicks an object. If a user clicks a form, the Form_Click event procedure executes; if a user clicks a command button named Command1, the Command1_Click event procedure executes.

Here's what happens in a typical event-driven application:

1. The application starts and automatically loads and displays the startup form.
2. A form or control receives an event. The event can be caused by the user (for example, a keystroke), by the system (for example, a timer event), or indirectly by your code (for example, a Load event when your code loads a form).
3. If there is an event procedure corresponding to that event, it executes.
4. The application waits for the next event.

Note Many events occur in conjunction with other events. For example, when the DblClick event occurs, the MouseDown, MouseUp, and Click events also occur.

Event-Driven vs. Traditional Programming

In a traditional or "procedural" application, the application itself rather than an event controls the portions of code that execute. Execution starts with the first line of executable code and follows a defined pathway through the application, calling procedures as needed.

In event-driven programs, a user action or system event executes an event procedure. Thus, the order in which your code executes depends on which events occur, which in turn depends on what the user does. This is the essence of graphical user interfaces and event-driven programming: The user is in charge, and your code responds.

Because you can't predict what the user will do, your code must make a few assumptions about "the state of the world" when it executes. When you must make assumptions (for example, that a text box has text in it before a command button is pressed), you should try to structure your application so those assumptions are always valid (for example, disabling the command button and enabling it only in the Change event for the text box).

Your code may trigger additional events as it performs certain operations. For example, loading a form causes the form's Load event to occur, and changing the Text property of a text box causes that text box's Change event to occur.

Note Avoid performing an operation in an event procedure that causes the same event to occur again; this causes a *cascading event*. For example, if your code in the Change event procedure for a text box sets the Text property of that text box, the Change event procedure is invoked again, in which your code sets the Text property again, causing the Change event to occur again, and so on until Visual Basic generates the error `"Out of Stack Space."`

Code That Executes at Startup

By default, the first form in your application is designated as the *startup form*. When your application starts running, this form is displayed (so the first code to execute is the code in the Form_Load event for that form). If you want a different form to be displayed when your application starts, you must change the startup form.

▶ **To change the startup form**

1. From the Options menu, choose Project.

   The Project Options dialog box is displayed.

2. Select the Start Up Form option.
3. In the Start Up Form list box, select the form you want to be the new startup form.

Sometimes you might want your application to start without any form initially loaded. For example, you might want to execute code that loads a data file and then displays one of several different forms depending on what was in the data file. You can do this by creating a **Sub** procedure called Main in a code module. This procedure *must* be a **Sub** procedure, and it cannot be in a form module. You can then select Sub Main from the Start Up Form option in the Project Options dialog.

Ending an Application

An event-driven application stops running when all its forms are closed and no code is executing. If there is a hidden form when the last visible form is closed, your application will appear to have ended (because no forms are visible) but will in fact continue to run until all the hidden forms are closed. This situation often arises because any access to an unloaded form's properties or controls implicitly loads that form without displaying it.

The easiest way to avoid this problem and end your program is to use the **End** statement, which stops the execution of all code in your application and closes any files you may have opened. For example, you could have a command button named cmdQuit that lets a user exit the program. The Click event procedure could be as simple as this:

```
Sub cmdQuit_Click ()
    End
End Sub
```

The **End** statement ends an application immediately; no code after the **End** statement is executed, and no further events occur. However, sometimes you want the Unload and QueryUnload events to occur for each form before the application ends. You can do this by explicitly unloading each form with the **Unload** statement before executing the **End** statement.

In addition to the **End** statement, the **Stop** statement halts an application. However, you should use the **Stop** statement only while debugging.

For More Information For information on the **Stop** statement, see Chapter 9, "Debugging," or search Help for *Stop*.

Modules

Simple applications can consist of just a single form, and all of the code in the application resides in that form module. As your applications get larger and more sophisticated, you add additional forms. Eventually you find that there is common code you want to execute in several forms. You cannot invoke procedures in one form from another, and you don't want to duplicate the code in both forms. So you create a separate code module containing a procedure that implements the common code, and then invoke the procedure from each form module. Over time, you can build up a library of code modules containing useful, common procedures.

Each code and form module can contain:

- **Declarations**. You can place constant, type, variable, and DLL procedure declarations at the module level of form or code modules. However, you cannot place any executable code (anything other than declarations) at the module level.
- **Event procedures**. These are **Sub** procedures that are executed in response to a user or system event. Event procedures occur only in form modules.
- **General procedures**. These are procedures that are not directly associated with an event. General procedures in a form module are local to that module; they can't be invoked from other modules. All the procedures in a code module are general procedures, and they can be invoked from any code or form module in the application. General procedures can be either **Sub** procedures (procedures that do not return a value) or **Function** procedures (procedures that return a value).

Sub and **Function** procedures are discussed in more detail in the next section, "Procedures." Module-level declarations are explained in Chapter 7, "Variables, Constants, and Data Types."

Event Procedures

When an object in Visual Basic recognizes that an event has occurred, it automatically invokes the event procedure with the name that corresponds to the event. Because the name establishes an association between the object and the code, event procedures are said to be attached to forms and controls.

An event procedure for a control combines the control's actual name (specified in the Name property), an underscore (_), and the event name. For instance, if you want a command button named MyButton to invoke an event procedure when it is clicked, use the procedure MyButton_Click.

An event procedure for a form combines the word "Form," an underscore, and the event name. If you want a form to invoke an event procedure when it is clicked, use the procedure Form_Click. (Like controls, forms do have unique names, but they are not used in the names of event procedures.) If you are using the MDIForm, the event procedure combines the word "MDIForm," an underscore, and the event name.

Note Although you can write event procedures from scratch, it's better to use the code templates provided by Visual Basic, which automatically include the correct procedure names.

All event procedures use the same general syntax.

Syntax for control event	Syntax for form event
Sub *controlname_eventname* ( ) *statementblock* **End Sub**	**Sub Form_***eventname* ( ) *statementblock* **End Sub**

The words **Sub** and **End Sub** mark the beginning and end of the procedure. The words following **Sub** are the procedure's name, and *statementblock* is the code you want executed when the event occurs.

Important It's a good idea to change the names of your controls *before* you start writing event procedures for them. If you change the name of a control *after* attaching a procedure to it, you also must change the name of the procedure to match the new name of the control. Otherwise, Visual Basic won't be able to match the control to the procedure. When a procedure name does not match a control name, it becomes a general procedure: In the Code window, select "(general)" from the Object box, and then select the procedure name from the Procedure box.

For More Information Visual Basic recognizes a variety of events for each kind of form and control. For explanations of all events, see the *Language Reference*, or search Help for *Events*.

General Procedures

Event procedures are invoked when a particular event occurs on a form or control. Those event procedures can in turn invoke other procedures. A procedure that is not invoked when an event occurs is called a *general procedure*. A general procedure is not invoked until another part of the application explicitly calls it.

Why create general procedures? One reason is that several different event procedures may need the same actions performed. A good programming strategy is to put common statements in a separate procedure (a general procedure) and have event procedures call it. This eliminates the need to duplicate code and also makes the application easier to maintain. For example, the scroll bars example in Chapter 3 used a general procedure called by the scroll events for three different scroll bars. Figure 6.1 illustrates the use of a general procedure.

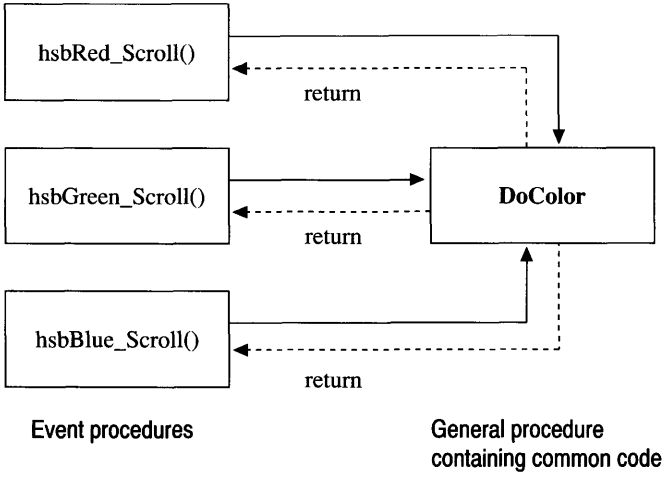


Figure 6.1 How general procedures are called by event procedures

Once you have defined a general procedure in a form module, you can call it from any of the event procedures in that form. If you want a general procedure that can be called from anywhere in your application, you must place it in a code module.

Working with Procedures

To create a new general procedure, make a Code window the active window. Then choose New Procedure from the View menu. Another technique is to type a procedure heading in the Code window and press ENTER. The procedure heading can be as simple as **Sub** or **Function** followed by a name. For example, you can enter either of the following:

```
Sub UpdateForm ()

Function GetCoord ()
```

Visual Basic responds by clearing the Code window and displaying a template for the new procedure.

To edit an existing general procedure, select "(general)" from the Object box in the Code window, and then select the procedure in the Procedure box.

The programming environment provides several other ways to work with procedures, as summarized in the following table.

Task	Menu command
List all procedures in your project.	From the Window menu, choose Procedures.
Edit a procedure in another module.	From the Window menu, choose Procedures. Select the module and the name of a procedure to view, and then choose OK.
Delete a procedure from a module.	Select the entire procedure in the Code window, and then choose Cut from the Edit menu.
Move or copy a procedure from one module to another.	Select the entire procedure in the Code window, and then choose Cut or Copy from the Edit menu. Click the destination module, and then choose Paste from the Edit menu.
Undo a change to code.	From the Edit menu, choose Undo.

Procedures

Procedures can be either **Sub** procedures or **Function** procedures. **Sub** procedures do not return a value, so a call to a **Sub** procedure is a complete statement. **Function** procedures return a value, so a call to a **Function** procedure is part of an expression.

Note Event procedures are always **Sub** procedures, never **Function** procedures.

Sub Procedures

The syntax for a **Sub** procedure is:

Sub *procedurename* (*arglist*)
    statements
End Sub

The *arglist* is a list of argument names, separated by commas if there is more than one. Each argument looks like a variable declaration and acts like a variable in the procedure. The syntax of each argument is:

[**ByVal**] *variablename* [( )] [**As** *type*]

If you do not provide a type, the argument is given the **Variant** type. The **Variant** type is very flexible, so often you will not need to specify a type for the argument. However, the *type* can be any of the fundamental data types (**Integer**, **Long**, **Single**, **Double**, **Currency**, or **String**), a user-defined type, or an object type. Parentheses after *variablename* indicate that the argument is an array. See Chapter 7, "Variables, Constants, and Data Types," for more information on data types, arrays, and the **ByVal** keyword. Object types are discussed in Chapter 8, "Objects and Instances."

Each time the procedure is called, the *statements* between **Sub** and **End Sub** are executed. Visual Basic substitutes each reference to an item in the argument list with the corresponding argument. When Visual Basic calls your event procedures, it supplies a value for each argument (if the event procedure has any). When you call your own general procedures, you must supply values for each argument. For example, suppose you have defined the MultiBeep general procedure as:

```
Sub MultiBeep (NBeeps)
Dim I
    For I = 1 To NBeeps
        Beep
    Next I
End Sub
```

The following statement calls MultiBeep with the argument *3*:

```
MultiBeep 3
```

The procedure substitutes *3* for *NBeeps* (the name that appears in the argument list). The procedure therefore beeps three times.

Normally, calls to a Visual Basic **Sub** procedure do not use parentheses around the argument list. If you call a **Sub** procedure with the optional **Call** statement, however, you must surround the argument list with parentheses:

```
Call MultiBeep(3)
```

For More Information For additional details about the **Sub** procedure, search Help for *Sub procedures*. For information about the **Call** statement, search Help for *Call*.

Function Procedures

The syntax for a **Function** procedure is:

Function *procedurename* (*arguments*) [**As** *type*]
    statements
End Function

The arguments for a **Function** procedure work in exactly the same way as the arguments for a **Sub** procedure. Aside from the **Function** keyword, there are three differences between **Sub** and **Function** procedures:

- You always use parentheses with each **Function** call (you cannot use the **Call** statement to call a **Function** procedure).
- **Function** procedures have data types, just as variables do. This determines the type of the return value. (In the absence of an **As** clause, the type is the default **Variant** type unless the *procedurename* has a type-declaration character.)
- You return a value by assigning it to the *procedurename* itself. When the **Function** procedure returns a value, this value is then used as part of a larger expression.

For example, you could write a function that calculates the third side, or hypotenuse, of a right triangle given the other two sides:

```
Function Hypotenuse (A, B)
    Hypotenuse = Sqr(A ^ 2 + B ^ 2)
End Function
```

You call a **Function** procedure the same way you call any of the built-in functions in Visual Basic:

```
Label1.Caption = Hypotenuse (Val(Text1.Text), Val(Text2.Text))

X = Hypotenuse(Width, Height)
```

For More Information For additional details about the **Function** procedure, search Help for *Function procedures*.

Public and Private Procedures

The procedures in a form are *private* to that form; you can call them from within the form, but the code in other form or code modules cannot call them. The procedures in a code module are *public*; they can be called from anywhere else in your application.

Because the procedures in a form are private, you can have procedures with the same name in different forms. This is why each form can have form event procedures with the same name (such as Form_Load). The public procedures in code modules must be unique across all code modules.

You can create a procedure in a form module with the same name as a private procedure in another module. In this situation, Visual Basic uses two rules to decide which procedure to invoke:

- Visual Basic first looks in the current form or code module.
- If the procedure name is not found, Visual Basic scans all modules (but not forms) for the procedure.

Consequently, calls within the module where the private procedure is defined will go to that procedure; calls outside that module will go to the public procedure.

Private Procedures in Modules

Sometimes you may want to create procedures in a module that are private to that module. For example, you may have "helper" routines in a module that should not be called from outside the module and should be available only to other procedures within that module. Or you may have many modules, perhaps written by several individuals, and you want to minimize the conflicts between names in the modules by making some of the procedures private. You create such private procedures by declaring them with the **Private** keyword:

```
Private Sub SomeSub ()
    .
    .
    .
End Sub

Private Function SomeFunction ()
    .
    .
    .
End Function
```

You can declare procedures in a form with the **Private** keyword, but this has no effect because the procedures in a form are always private anyway. There is no way to make the procedures in a form public. If you want to call a general procedure in a form, you should move it to a code module.

For More Information For additional details about **Private** procedures, search Help for *Private*.

Code in Procedures

As you read the previous chapters, you saw (and perhaps typed in and tried out) code in procedures. Maybe the code seemed obvious; maybe you were puzzled by some of it. The rest of this chapter describes the fundamentals of the code you've seen so far and prepares you for the code you will see in the rest of this manual.

Some Mechanics (Comments, Numbers, and Statements)

As you read through the examples in this manual, you'll often come across the comment symbol ('). This symbol tells Visual Basic not to do anything with the words that follow it. Such words are remarks placed in the code for the benefit of you, the developer, and other programmers who might examine the code later. For example:

```
' This is a comment beginning at the left edge of the screen.
Text1.Text = "Hi!"          ' Place friendly greeting in text box.
```

You will also see a lot of numbers. Most numbers in this manual are decimal (base 10). But occasionally it's convenient to use hexadecimal numbers (base 16). Visual Basic represents numbers in hexadecimal with the prefix &H and octal (base 8) with &O. The following table shows the same numbers in decimal, octal, and hexadecimal.

Decimal	Octal	Hexadecimal
9	&O11	&H9
15	&O17	&HF
16	&O20	&H10
20	&O24	&H14
255	&O377	&HFF

Although some important values in the CONSTANT.TXT file are defined in hexadecimal, you generally don't have to learn the hexadecimal or octal number system yourself.

Note If you want to use a hexadecimal or octal number in situations that involve large numbers, append an ampersand (&) to the number. This causes the value to be stored correctly as a long integer.

Visual Basic statements are normally one to a line, and there is no statement terminator. However, you can place more than one statement on a line if you use a colon (:) to separate them:

```
Text1.Text = "Hello" : Red = 255 : Text1.BackColor = Red
```

Naming Conventions in Visual Basic

While you are writing your Visual Basic code, you declare and name many elements (**Sub** and **Function** procedures, variables and constants, and so on). The names of the procedures, variables, and constants that you declare in your Visual Basic code must follow the same rules as the names you give to your forms and controls:

- They must begin with a letter.
- They must contain only letters, numbers, and the underscore character; punctuation characters and spaces are not allowed.
- They must be no longer than 40 characters.

In addition, there is one more rule:

- The names of items you declare in your code cannot be reserved words.

A *reserved word* is a word that Visual Basic uses as part of its language. This includes predefined statements (such as **If** and **Loop**), functions (such as **Len** and **Abs**), methods (such as **Show** and **Move**), and operators (such as **Or** and **Mod**). In Visual Basic 1.0, property names (such as Visible and Top) were considered reserved words; this is no longer true.

For More Information For a complete list of reserved words, see Appendix F of the *Language Reference*, or search Help for *reserved words*.

Your forms and controls can have the same name as a reserved word. For example, you can have a control named Loop. In your code you cannot refer to that control in the usual way, however, since Visual Basic assumes you mean the **Loop** keyword. For example, this code causes an error:

```
Loop.Visible = True              ' Causes an error.
```

To refer to a form or control that has the same name as a reserved word, you must surround it with square brackets: [ ]. For example, this code does not cause an error:

```
[Loop].Visible = True            ' Square brackets work.
```

You can use square brackets in this way only when referring to forms and controls that you have named with a reserved word. You cannot use square brackets to declare a variable or define a procedure with the same name as a reserved word.

Note Because typing the square brackets can get tedious, you may want to refrain from using reserved words as the name of forms and controls. However, you can use this technique if a future version of Visual Basic defines a new keyword that conflicts with an existing form or control name when you update your code to work with the new version.

Setting and Retrieving Properties

Probably the most common statement in Visual Basic is the *assignment statement*, which assigns a value to a variable or to a property reference by copying data from one place to another. The syntax uses an equal sign (=):

destination = *source*

The statement tells the application, "Copy information from *source* to *destination*." The *destination* must be a variable or reference to a property. The *source* can be any valid Visual Basic expression and may involve calculations.

Assignment statements in Visual Basic generally do one of three things:

- Set the value of a property.
- Retrieve the value of a property.
- Store or retrieve data in a variable.

Setting the Value of a Property

To set the value of a property at run time, place a reference to the property (*object.property*) on the left side of an assignment statement. For example:

```
Text1.Text = "Your name here."
Text1.BackColor = 0
```

When you are referring to a form property in the code for that form, you can omit the name of the form:

```
Sub Form_Click ()
' Set the Caption property of the form.
    Caption = "You clicked me!"
End Sub
```

The Text property is a string of characters that specifies the contents of a text box, list box, or combo box. While you generally assign text strings to text properties, you can assign numbers or mix numbers with text:

```
Text1.Text = 60
Text2.Text = "The amount is " & 60
```

In these statements, as a result of the **Variant** data type, Visual Basic automatically converts the number into a string before assigning it to the Text property. For more information on the **Variant** data type, see Chapter 7, "Variables, Constants, and Data Types."

Retrieving the Value of a Property

To retrieve the value of a property and store it in a variable or another property, place a reference to the property on the right side of an assignment statement. For example:

```
Text2.Text = Text1.Text
MyString = Text1.Text
```

Properties on Other Forms

When you have several forms or code modules in your application, you'll find you need to refer to the properties and controls on one form from the code in another form or code module. The syntax for doing this is an extension of the syntax you've already used. You must specify the form when you are referring to a form property from another form or code module:

```
Form1.Caption = "A new caption for the form"
Form1.BackColor = 0
```

Controls on Other Forms

When you are referring to a control on one form from code in another form or code module, you must specify the form that contains the control. You separate the form and control references with the **!** operator:

```
Form1!Text1.Text = "Your name here"
Form1!Text1.BackColor = 0
Form1!Text1.Move 0, 0
```

Note Visual Basic allows you to use the period qualifying operator (.) in place of the **!** operator when referring to controls on forms. This is to ease compatibility with code written for Microsoft Visual Basic 1.0, which used the period qualifying operator to refer to controls on forms. Nevertheless, using the **!** operator to reference controls on forms is recommended to avoid confusion between controls and form properties that have the same name. If a control has the same name as a form property, references to that name using the period qualifying operator always refer to the property and never to the control. References using the **!** operator always refer to the control.

Using the Value of a Control

All controls have a property that you can use for storing or retrieving values just by referring to the control, without using the property name. This is called the *value* of the control and is usually the most important or most commonly used property for that kind of control. Table 6.1 lists the property that is considered to be the value for each kind of control.

Table 6.1 Controls and the Properties That Are Their Values

Control	Property
Check box	Value
Combo box	Text
Command button	Value
Common dialog	Action
Data	Caption
Directory list box	Path
Drive list box	Drive
File list box	FileName
Frame	Caption
Grid	Text
Horizontal scroll bar	Value
Image	Picture
Label	Caption
Line	Visible
List box	Text
Menu	Enabled
Option button	Value
Picture box	Picture
Shape	Shape
Text box	Text
Timer	Enabled
Vertical scroll bar	Value

Whenever you want to refer to a property on a control that happens to be the value of that control, you can do so without specifying the property name in your code. For example, this line sets the value of the Text property of a text box control:

```
Text1 = "This text is assigned to the Text property of Text1"
```

In this example, the Caption property of Label1 is set to the FileName property of File1 whenever the user clicks a file in the file list box:

```
Sub File1_Click ()
    Label1 = File1
End Sub
```

Note Using the value of a control instead of the equivalent property causes your code to run slightly faster. Because it also makes your code somewhat less readable, however, the examples in this manual do not use it but instead refer explicitly to the properties on all controls. You may want to try writing your code both ways, and decide to use the value of controls in your code if you have no trouble reading it.

Variables

You often need to store values temporarily when performing calculations with Visual Basic. For example, you may want to calculate several values, compare them, and perform different operations on them, depending on the result of the comparison. You need to retain the values if you want to compare them but not store them in a property.

Visual Basic, like most programming languages, uses *variables* for storing values. Variables have a *name* (the word you use to refer to the value the variable contains) and a *data type* (which determines the kind of data the variable can store).

Declaring Variables

You don't have to explicitly declare or create variables in Visual Basic. If you use a name in your code that's not the name of any existing variable, Visual Basic creates a new variable with that name. However, it's good programming practice to explicitly declare your variables. Also, you can use the Environment Options dialog box to specify that variable declarations are required, in which case you must declare your variables.

You declare a variable with the **Dim** statement:

Dim *variablename*

Variable names follow the same rules as everything else you name in the Visual Basic language. For example, you create a variable called `Result` with this line:

```
Dim Result
```

Variables declared with the **Dim** statement within a procedure exist only as long as the procedure is executing. When the procedure finishes, the value of the variable disappears. In addition, the value of a variable in a procedure is *local* to that procedure: You can't access a variable in one procedure from another procedure. These characteristics allow you to use the same variable names in different procedures without worrying about conflicts or accidental changes.

By declaring a variable in a different way, however, you can preserve its value even when a procedure ends, or you can make it available to several procedures. For more information on requiring variable declaration or changing the lifetime or scope of variables, see Chapter 7, "Variables, Constants, and Data Types."

Storing and Retrieving Data in Variables

You use assignment statements to perform calculations and assign the result to a variable:

```
Ergs = 10
Ergs = Ergs + 1
```

Data Types

By default, Visual Basic variables are of the **Variant** data type. The **Variant** data type can store numeric, date/time, or string data. You don't have to convert between these types of data when assigning them to a **Variant** variable; Visual Basic automatically performs any necessary conversion. If you know that a variable will always store data of a particular type, however, Visual Basic can handle that data more efficiently if you declare a variable of that type. For information about declaring variables with data types other than **Variant**, see Chapter 7, "Variables, Constants, and Data Types."

Controlling Execution

The statements that control decisions and loops in Visual Basic are called *control structures*. Visual Basic control structures are similar to those control structures found in C and Pascal. The most commonly used control structures in Visual Basic are:

- **If_Then** blocks
- **If_Then_Else** blocks
- **Select Case** statements
- **Do** loops
- **For** loops

The first three items in the preceding list are decision structures. You use them to define groups of statements that may or may not be executed, depending on run-time conditions. The last two items are loop structures. You use them to define groups of statements that Visual Basic executes repeatedly.

Decision Structures

Like macros, Visual Basic procedures can test conditions and then, depending on the results of that test, perform different operations. The decision structures that Visual Basic supports include:

- **If...Then**
- **If...Then...Else**
- **Select Case**

If...Then

Use an **If...Then** block to execute one or more statements conditionally. You can use either a single-line syntax or a multiple-line "block" syntax:

If *condition* **Then** *statement*

If *condition* **Then**
    statements
End If

The *condition* is usually a comparison, but it can be any expression that evaluates to a numeric value. Visual Basic interprets this value as **True** or **False**; a zero numeric value is **False**, and any nonzero numeric value is considered **True**. If *condition* is **True**, Visual Basic executes all the *statements* following the **Then** keyword. You can use either single-line or multiple-line syntax to execute just one statement conditionally (these two examples are equivalent):

```
If anyDate < Now Then anyDate = Now

If anyDate < Now Then
    anyDate = Now
End If
```

Notice that the single-line form of **If...Then** does not use an **End If** statement. If you want to execute more than one line of code when *condition* is **True**, you must use the multiple-line block **If...Then...End If** syntax.

```
If anyDate < Now Then
    anyDate = Now
    Timer1.Enabled = False          ' Disable timer control.
End If
```

If...Then...Else

Use an **If...Then...Else** block to define several blocks of statements, one of which gets executed:

If *condition1* **Then**
  [*statementblock-1*]
[**ElseIf** *condition2* **Then**
  [*statementblock-2*]] ...
[**Else**
  [*statementblock-n*]]
End If

Visual Basic first tests *condition1*. If it's **False**, Visual Basic proceeds to test *condition2*, and so on, until it finds a **True** condition. When it finds a **True** condition, Visual Basic executes the corresponding statement block and then executes the code following the **End If**. As an option, you can include an **Else** statement block, which Visual Basic executes if none of the conditions are **True**.

If...Then is really just a special case of **If...Then...Else**. Notice that you can have any number of **ElseIf** clauses, or none at all. You can include an **Else** clause whether or not you have **ElseIf** clauses.

For example, your application could perform different actions depending on which control in a menu control array was clicked:

```
Sub mnuCut_Click (Index As Integer)
    If Index = 0 Then               ' Cut command.
        CopyActiveControl           ' Call general procedures.
        ClearActiveControl
    ElseIf Index = 1 Then           ' Copy command.
        CopyActiveControl
    ElseIf Index = 2 Then           ' Clear command.
        ClearActiveControl
    Else                            ' Paste command.
        PasteActiveControl
    End If
End Sub
```

Notice that you can always add more **ElseIf** parts to your **If...Then** structure. However, this syntax can get tedious to write when each **ElseIf** compares the same expression to a different value. For this situation, you can use a **Select Case** decision structure.

For More Information For additional details about **If...Then...Else**, see the *Language Reference*, or search Help for *If*.

Select Case

Visual Basic provides the **Select Case** structure as an alternative to **If...Then...ElseIf** for selectively executing one block of statements from among multiple blocks of statements. A **Select Case** statement provides capability similar to the **If...Then...Else** statement, but it makes code more efficient and readable.

A **Select Case** structure works with a single test expression that is evaluated once, at the top of the structure. Visual Basic then compares the result of this expression with the values for each **Case** in the structure. If there is a match, it executes the block of statements associated with that **Case**:

Select Case *testexpression*
  [**Case** *expressionlist1*
    [*statementblock-1*]]
  [**Case** *expressionlist2*
    [*statementblock-2*]] ...
  [**Case Else**
    [*statementblock-n*]]
End Select

Each *expressionlist* is a list of one or more values. If there is more than one value in a single list, the values are separated by commas. Each *statementblock* contains zero or more statements. If more than one **Case** matches the test expression, only the statement block associated with the first matching **Case** is executed. Visual Basic executes statements in the **Case Else** clause (which is optional) if none of the values in the expression lists matches the test expression.

For example, suppose you added another command to the Edit menu in the **If..Then..ElseIf** example. You could add another **ElseIf** clause, or you could write the function with **Select Case**:

```
Sub mnuCut_Click (Index As Integer)
    Select Case Index
        Case 0                          ' Cut command.
            CopyActiveControl           ' Call general procedures.
            ClearActiveControl
        Case 1                          ' Copy command.
            CopyActiveControl
        Case 2 Then                     ' Clear command.
            ClearActiveControl
        Case 3                          ' Paste command.
            PasteActiveControl
        Case Else
            frmFind.Show                ' Show Find dialog.
    End Select
End Sub
```

Notice that the **Select Case** structure evaluates an expression once at the top of the structure. In contrast, the **If...Then...ElseIf** structure can evaluate a different expression for each **ElseIf** statement. You can replace an **If...Then...ElseIf** structure with a **Select Case** structure only if each **ElseIf** statement evaluates the same expression.

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively. The loop structures that Visual Basic supports include:

- **Do...Loop**
- **For...Next**

Do...Loop

Use a **Do** loop to execute a block of statements an indefinite number of times. There are several variations of the **Do...Loop** statement, but each evaluates a numeric condition to determine whether to continue execution. As with **If...Then**, the *condition* must be a value or expression that evaluates to **False** (zero) or to **True** (nonzero).

In the following **Do...Loop**, the *statements* are executed as long as the *condition* is **True**:

Do While *condition*
    statements
Loop

When Visual Basic executes this **Do** loop, it first tests *condition*. If *condition* is **False** (zero), it skips past all the statements. If it's **True** (nonzero), Visual Basic executes the statements and then goes back to the **Do While** statement and tests the condition again.

Consequently, the loop can be executed any number of times, as long as *condition* is nonzero or **True**. The statements are never executed if *condition* is initially **False**. For example, this procedure counts the occurrences of a target string within another string by looping as long as the target string is found:

```
Function CountStrings (longstring, target)
Dim position, count
    position = 1
    Do While InStr(position, longstring, target)
        position = InStr(position, longstring, target) + 1
        count = count + 1
    Loop
    CountStrings = count
End Function
```

If the target string doesn't occur in the other string, then **InStr** returns 0 and the loop isn't executed.

Another variation of the **Do...Loop** statement executes the statements first and then tests *condition* after each execution. This variation guarantees at least one execution of *statements*:

Do
    statements
Loop While *condition*

Two other variations are analogous to the previous two, except that they loop as long as *condition* is **False** rather than **True**.

Loop zero or more times	Loop at least once
Do Until *condition* *statements* **Loop**	**Do** *statements* **Loop Until** *condition*

Notice that **Do Until** *condition* is exactly equivalent to **Do While Not** *condition*.

For...Next

Do loops work well when you don't know how many times you need to execute the statements in the loop. When you know you must execute the statements a specific number of times, however, your code is more efficient if you use a **For** loop. Unlike a **Do** loop, a **For** loop uses a *counter* variable that increases or decreases in value during each repetition of the loop. The syntax is:

For *counter* = *start* **To** *end* [**Step** *increment*]
  statements
Next [*counter*]

The arguments *counter*, *start*, *end*, and *increment* are all numeric.

Note The argument *increment* can be either positive or negative. If *increment* is positive, *start* must be less than or equal to *end* or the statements in the loop won't be executed. If *increment* is negative, *start* must be greater than or equal to *end* for the body of the loop to be executed. If **Step** isn't set, then *increment* defaults to 1.

In executing the **For** loop, Visual Basic:

1. Sets *counter* equal to *start*.
2. Tests to see if *counter* is greater than *end*. If so, Visual Basic exits the loop.
   (If *increment* is negative, Visual Basic tests to see if *counter* is less than *end*.)
3. Executes the *statements*.
4. Increments *counter* by 1—or by *increment* if it's specified.
5. Repeats steps 2 through 4.

This code prints the names of all the available Screen fonts:

```
Sub Form_Click ()
Dim i
    For i = 0 To Screen.FontCount - 1
        Print Screen.Fonts(i)
    Next
End Sub
```

Another example in Chapter 3 used a **For...Next** loop to step through the entries in the Selected property of a multiple-column list box:

```
Sub cmdTransfer_Click ()
    For n = 0 To (lstTop.ListCount - 1)
        If lstTop.Selected(n) = True Then           ' If selected,
            lstBottom.AddItem lstTop.List(n)        ' add to list.
        End If
    Next
End Sub
```

Nested Control Structures

As the previous example demonstrates, you can place control structures inside other control structures (such as an **If...Then** block within a **For...Next** loop). A control structure placed inside another control structure is said to be *nested.*

Control structures in Visual Basic can be nested to as many levels as you want. It's common practice to make nested decision structures and loop structures more readable by indenting the body of the decision structure or loop.

For example, this procedure prints all the font names that are common to both the Printer and Screen:

```
Sub Form_Click ()
Dim SFont, PFont
    For SFont = 0 To Screen.FontCount - 1
        For PFont = 0 To Printer.FontCount - 1
            If Screen.Fonts(SFont) = Printer.Fonts(PFont) Then
                Print Screen.Fonts(SFont)
            End If
        Next PFont
    Next SFont
End Sub
```

Notice that the first **Next** closes the inner **For** loop and the last **For** closes the outer **For** loop. Likewise, in nested **If** statements, the **End If** statements automatically apply to the nearest prior **If** statement. Nested **Do...Loop** structures work in a similar fashion, with the innermost **Loop** statement matching the innermost **Do** statement.

Exiting a Control Structure

The **Exit** statement allows you to exit directly from a **For** loop, **Do** loop, **Sub** procedure, or **Function** procedure. Syntactically, the **Exit** statement is simple: **Exit For** can appear as many times as needed inside a **For** loop, and **Exit Do** can appear as many times as needed inside a **Do** loop:

For *counter* = *start* **To** *end* [**Step** *increment*]
  [*statementblock*]
  [**Exit For**]
  [*statementblock*]
Next [*counter*[, *counter*] [,...]]

Do [{**While** | **Until**} *condition*]
  [*statementblock*]
  [**Exit Do**]
  [*statementblock*]
Loop

The **Exit Do** statement works with all versions of the **Do** loop syntax.

Exit For and **Exit Do** are useful because sometimes it's appropriate to quit a loop immediately, without performing any further iterations or statements within the loop. For example, in the previous example that printed the fonts common to both the Screen and Printer, the code continues to compare Printer fonts against a given Screen font even when a match has already been found with an earlier Printer font. A more efficient version of the function would exit the loop as soon as a match is found:

```
Sub Form_Click ()
Dim SFont, PFont
    For SFont = 0 To Screen.FontCount - 1
        For PFont = 0 To Printer.FontCount - 1
            If Screen.Fonts(SFont) = Printer.Fonts(PFont) Then
                Print Screen.Fonts(SFont)
                Exit For                ' Exit inner loop.
            End If
        Next PFont
    Next SFont
End Sub
```

As this example illustrates, an **Exit** statement almost always appears inside an **If** statement or **Select Case** statement nested inside the loop.

Exiting a Sub or Function Procedure

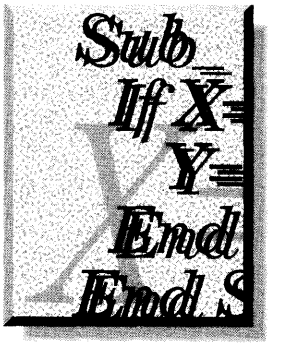
The syntax of **Exit Sub** and **Exit Function** is similar to that of **Exit For** and **Exit Do** in the previous section, "Exiting a Control Structure." **Exit Sub** can appear as many times as needed, anywhere within the body of a **Sub** procedure. **Exit Function** can appear as many times as needed, anywhere within the body of a **Function** procedure.

Exit Sub and **Exit Function** are useful when the procedure has done everything it needs to do and can return immediately. For example, if you want to change the previous example so it prints only the first common Printer and Screen font it finds, you would use **Exit Sub**:

```
Sub Form_Click ()
Dim SFont, PFont
    For SFont = 0 To Screen.FontCount - 1
        For PFont = 0 To Printer.FontCount - 1
            If Screen.Fonts(SFont) = Printer.Fonts(PFont) Then
                Print Screen.Fonts(SFont)
                Exit Sub            ' Exit the procedure.
            End If
        Next PFont
    Next SFont
End Sub
```

CHAPTER 7

Variables, Constants, and Data Types



So far, most of the **Function** procedures, arguments, and variables you've seen have had the **Variant** data type. The **Variant** data type is very flexible and can accommodate many of your needs. However, Visual Basic supplies several other data types that allow you to optimize your code for speed and size when you don't need the flexibility that **Variant** provides. You can declare constants and create arrays of any of these variable types.

Contents

- Declaring Variables
- Scope and Lifetime of Variables
- Fundamental Variable Data Types
- Arrays
- User-Defined Types (Structures)
- Symbolic Constants

Declaring Variables

As you saw in Chapter 6, "Programming Fundamentals," you use variables to store values while your Visual Basic code is executing. Within a procedure, you declare a variable with the **Dim** statement, supplying a name for the variable:

Dim *variablename* [**As** *type*]

Variable names follow the same rules as everything else you create and name in Visual Basic. A variable name:

- Must begin with a letter.
- Must contain only letters, numbers, and the underscore character (_); punctuation characters and spaces are not allowed.
- Must not exceed 40 characters.
- Cannot be a reserved word.

The optional **As** *type* clause in the **Dim** statement allows you to define the data type of the variable you are declaring. If you omit it, Visual Basic makes the variable the default data type, which is **Variant** (unless you change it with the **Def***type* statement). Data types, including **Variant**, are discussed later in this chapter.

Note In Microsoft Visual Basic version 1.0, the default data type was **Single**, not **Variant**. Because the **Variant** type can store the same range of values as the **Single** type, this change should not affect any calculations your code performs on variables of the default data type. If you wrote code in Visual Basic 1.0 that performed **Get** and **Put** statements on variables that were not explicitly declared with a data type, you must change this code to explicitly declare these variables with the **Single** data type. For more information, see Appendix C, "Compatibility with Other Versions," or search Help for *Get* and *Put*.

Implicit Declaration

You don't have to declare a variable before using it. For example, you could write a function like this:

```
Function SafeSqr(num)
    TempVal = Abs(num)
    SafeSqr = Sqr(TempVal)
End Function
```

You don't have to declare TempVal before using it in the function. Visual Basic automatically creates a variable with that name, so you can use it as if you had explicitly declared it. While this is convenient, it can lead to subtle errors in your code if you misspell a variable name. For example, suppose that this was the function you wrote:

```
Function SafeSqr(num)
    TempVal = Abs(num)
    SafeSqr = Sqr(TemVal)
End Function
```

At first glance, this looks the same. But because the TempVal variable was misspelled on the next-to-last line, this function will always return zero. When Visual Basic encounters a new name, it can't determine whether you actually meant that name to be a new variable or you just mistyped an existing variable name, so it creates a new variable with that name.

Explicit Declaration

To avoid the problem of misnaming variables, you can stipulate that Visual Basic always generate an error message whenever it encounters a name not previously declared explicitly as a variable. To do this, place this statement in the Declarations section of a form or code module:

```
Option Explicit
```

Had this statement been in effect for the form or code module containing the SafeSqr function, Visual Basic would have recognized TempVal and TemVal as undeclared variables and generated errors for both of them. You could then explicitly declare TempVal:

```
Function SafeSqr(num)
    Dim TempVal
    TempVal = Abs(num)
    SafeSqr = Sqr(TemVal)
End Function
```

Now you'd understand the problem immediately because Visual Basic would have displayed an error message for the incorrectly spelled TemVal.

Because the **Option Explicit** statement helps you catch these kinds of errors, you may want to use it with all your code. By setting an environment option, you can have Visual Basic add **Option Explicit** to every form and code module you create.

▶ **To have Option Explicit automatically inserted in all new forms and modules:**

1. From the Options menu, choose Environment.
2. In the Environment Options dialog box, select the Require Variable Declaration option.
3. In the Settings box, type **yes**.

   –or–

   Select Yes in the Settings list box.

Note The **Option Explicit** statement operates on a per-module basis; it must be placed in the Declarations section of every form and code module for which you want Visual Basic to enforce explicit variable declarations. If you set Require Variable Declaration to Yes, Visual Basic inserts **Option Explicit** in all new form and code modules but does not add it to existing code. You must manually add **Option Explicit** to existing form and code modules.

Scope and Lifetime of Variables

When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable; it has a *scope* that is local to that procedure. Sometimes, however, you need to use a variable with a broader scope, such as one whose value is available to all the procedures within the same form or code module, or even to all the procedures in your entire application. Visual Basic allows you to specify the scope of a variable when you declare it.

Scoping Variables

Depending on how it is declared, a variable is scoped one of three ways.

Scope	Declaration
Local	**Dim**, **Static**, or **ReDim** (within a procedure)
Module	**Dim** (in Declarations section of a form or code module)
Global	**Global** (in Declarations section of a module)

Local Variables

A *local variable* is recognized only within the procedure in which it appears. It's a good choice for any kind of temporary calculation. A dozen procedures can all have a variable called `Temp`, for instance, but as long as each `Temp` is local, each procedure has its own variable. A procedure can alter its local `Temp` variable without affecting `Temp` variables in other procedures.

A procedure can even call itself (a technique called *recursion*), and each invocation of the procedure gets its own copy of the local variables in that procedure. Local variables are declared within procedures with the **Dim** statement. Local variables declared with **Dim** remain in existence only as long as the procedure is executing. Local variables declared with **Static** (described later in this section) remain in existence for the lifetime of the application.

Note Implicitly declared variables (if allowed because **Option Explicit** is not in effect) always have local scope.

Module-Level Variables

A *module-level variable* shares information among all the procedures in a code or form module. It's available to all the procedures in that module, but not to code in other form or code modules. You create module-level variables by declaring them with the **Dim** statement in the Declarations section of the module. Module-level variables remain in existence for the lifetime of the application, and module-level variables in forms preserve their values even when the form is unloaded.

Global Variables

A *global variable* has the broadest scope of all. The values in global variables are available in every procedure in every form and code module in your application. Like module-level variables, global variables are declared in the Declarations section of a module. However, you declare global variables with the **Global** statement instead of with the **Dim** statement.

Note You must declare global variables in the Declarations section of a code module; you cannot declare global variables in a procedure, and you cannot declare global variables anywhere in form modules.

You can declare your global variables in any module. For the purposes of organizing your code, however, it's often a good idea to keep your global variables confined to a limited number of modules so they are easy to find. This also helps you avoid declaring the same global variable twice in two different modules, something that causes an error when you try to compile your code.

Global variables exist, and retain their values, for the lifetime of the application.

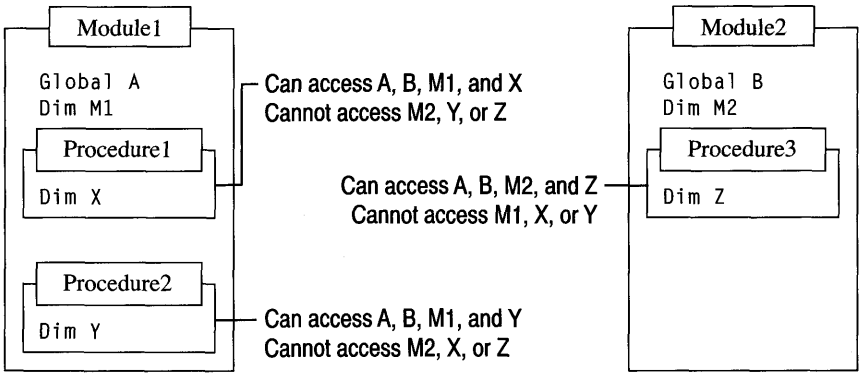


Figure 7.1 Visibility of variables with different scopes

Name Conflicts and Shadowing

A variable cannot change scope while your code is running. However, you can have a variable with the same name at a different scope. For example, you could have a global variable called `Temp` and then, within a procedure, declare a local variable called `Temp`. References to the name `Temp` within the procedure would access the local variable; references to `Temp` outside the procedure would access the global variable.

In general, when variables have the same name but different scope (see Figure 7.1), the more local variable always *shadows* (is accessed in preference to) less local variables. So if you also had a module-level variable named `Temp`, it would shadow the global variable `Temp` within that module (and the local `Temp` would shadow the module-level `Temp` within that procedure).

You can also declare a variable with the same name as a form property. Within the code in a form module, a variable with the same name as a form property shadows the property. If you want to access a form property that is shadowed by a variable, you must qualify the property name with a reference to the form name or the **Me** keyword. Within form code, similarly, variables with the same names as controls on the form shadow the controls. You must qualify the control with a reference to the form or **Me** to set or get its value or any of its properties. For example:

```
Sub Form_Click ()
    Dim Text1, BackColor
```

```
    ' Assume there is also a control on the form called Text1.
    Text1 = "Variable"      ' Variable shadows control.
    Me!Text1 = "Control"    ' Must qualify with Me to get control.
    Text1.Top = 0           ' This causes an error!
    Me!Text1.Top = 0        ' Must qualify with Me to get control.
    BackColor = 0           ' Variable shadows property.
    Me!BackColor = 0        ' Must qualify with Me to get form property.
End Sub
```

For more information about **Me** and the **!** operator, see Chapter 8, "Objects and Instances."

While these shadowing rules are not complex, shadowing can confuse you and lead to subtle bugs in your code, so it's considered good programming practice to keep the names of your variables distinct from each other and, in form modules, different from the names of properties and controls on the form.

The names of your module-level and global variables can also conflict with the names of your procedures. Because the names of procedures in modules have global scope—unless a procedure is in a form or is declared **Private**, it's visible to and can be called by all other procedures—a global variable can't have the same name as any public procedure in any code module in the application. **Private** procedures have module-level scope for the module in which they are defined and therefore cannot have the same name as a module variable in that module. Procedures in form modules have module-level scope within the form and therefore cannot have the same name as module-level variables in the form.

Static Variables

In addition to scope, variables also have a lifetime. The values in module and global variables are preserved for the lifetime of your application. However, local variables declared with **Dim** exist only while the procedure in which they are declared is executing. Normally, when a procedure is finished executing, the values of its local variables are not preserved and the memory used by the local variables is reclaimed. The next time the procedure is executed, all its local variables are reinitialized.

However, you can make Visual Basic *preserve* the value of a local variable by making the variable *static*. Use the **Static** keyword to declare one or more variables inside a procedure, exactly as you would with the **Dim** statement:

```
Static Depth
```

For example, the following function calculates a running total by adding a new value to the total of previous values stored in the static variable `Accumulate`:

```
Function RunningTotal(num)
    Static Accumulate
    Accumulate = Accumulate + num
    RunningTotal = Accumulate
End Function
```

If `Accumulate` was declared with **Dim** instead of **Static**, the previous accumulated values would not be preserved across calls to the function, and the function would simply return the same value with which it was called.

You could produce the same result by declaring `Accumulate` in the Declarations section of the module, making it a module variable. Once you change the scope of a variable this way, however, the procedure no longer has exclusive access to it. Since other procedures can access the value of the variable and change it, the running totals might be unreliable and the code would be more difficult to maintain.

Because event procedures, like all procedures, reinitialize their local variables each time they are invoked, you often use static variables in event procedures to record or "flag" certain conditions from one event to the next. For example, if you want to display a message the first time a form is loaded but not thereafter, you can write code like this:

```
Sub Form_Load ()
    Static LoadedOnceAlready
    If Not LoadedOnceAlready Then
        LoadedOnceAlready = True
        MsgBox "Welcome to my Visual Basic application"
    End If
End Sub
```

Notice that this code works because static variables in forms, like module-level variables in forms, continue to preserve their values even when the form is unloaded.

Declaring All Local Variables as Static

To make *all* local variables in a procedure static, place the **Static** keyword at the beginning of a procedure heading. For example:

```
Static Function RunningTotal(num)
```

This makes all the local variables in the procedure static, regardless of whether they are declared with **Static** or **Dim** or declared implicitly. You can place **Static** in front of any **Sub** or **Function** procedure heading, including event procedures and those that are also declared as **Private**.

Fundamental Variable Data Types

When you declare a variable, you can also supply a data type for it. All variables have a data type that determines what kind of data they can store. By default, if you don't supply a data type, the variable is given the **Variant** data type.

The Variant Data Type

The **Variant** data type can store many kinds of data. Like a text box control on a form, a **Variant** variable is equally capable of storing numbers, strings of text, or dates and times. You don't have to convert between these types of data when assigning them to a **Variant** variable; Visual Basic automatically performs any necessary conversion. For example:

```
Dim SomeValue       ' Variant by default.
SomeValue = "17"    ' SomeValue contains "17" (a two-character string).
SomeValue = SomeValue - 15
' SomeValue now contains the numeric value 2.
SomeValue = "U" & SomeValue
' SomeValue now contains "U2" (a two-character string).
```

While you can perform operations on **Variant** variables without much concern for what kind of data they actually contain, there are some traps you must avoid. If you perform arithmetic operations or functions on a **Variant**, it must contain something that is a number. For example, you cannot perform any arithmetic operations on the value "U2" even though it contains a numeric character, because the entire value is not a valid number. Likewise, you cannot perform any calculations on the value "1040EZ," but you can perform calculations on the values "+10" or "–1.7E6" because they are valid numbers.

Another trap involves the + operator. Because the + operator can be used for both addition and string concatenation, you can get unexpected results when you use it with two **Variant** variables; depending on what the variables contain, Visual Basic will either concatenate or add the values. The **&** operator performs string concatenation of two variables regardless of what the variables contain, so use it instead of the + operator when you want to concatenate two **Variant** variables. The **&** operator is discussed in more detail in the section "Strings Stored in **Variants**" later in this chapter.

Internal Representation of Values in Variants

Variant variables maintain an internal representation of the values that they store. This internal representation determines how Visual Basic treats these values when performing comparisons and other operations. When you assign a value to a **Variant** variable, Visual Basic uses the most compact representation that accurately records the value. Later operations may cause Visual Basic to change the representation it is using for a particular variable. (A **Variant** variable is not an untyped variable; rather, it is a variable that can freely change its type.) These internal representations correspond to the explicit data types discussed later in this chapter.

Most of the time you don't have to worry about what internal representation Visual Basic is using for a particular variable; Visual Basic handles conversions automatically. If you want to know, however, you can use the **VarType** function to determine what representation a variable is using. Table 7.1 lists the return values of the **VarType** function.

Table 7.1 VarType Return Values

Return value	Internal representation
0	Empty
1	**Null**
2	**Integer**
3	**Long**
4	**Single**
5	**Double**
6	**Currency**
7	Date/Time
8	**String**

For example, if you store values with decimal fractions in a **Variant** variable, Visual Basic always uses the **Double** internal representation. If you know that your application does not need the high accuracy (and slower speed) that a **Double** supplies, you can speed your calculations by converting the values to **Single**, or even **Currency**:

```
If VarType(X) = 5 Then X = CSng(X)          ' Convert to Single.
```

Note Future versions of Visual Basic may add additional **Variant** representations, so any code you write that makes decisions based on the return value of the **VarType** function should gracefully handle return values greater than 8. **Variant** variables may grow in size in future versions as well, so you should not make any assumptions in your code regarding their size.

For More Information For information about the **VarType** function, see the *Language Reference*, or search Help for *VarType*.

Numeric Values Stored in Variants

When you store numbers in a **Variant** variable, Visual Basic uses the most compact representation possible. So if you store a small number without a decimal fraction, the **Variant** uses an **Integer** representation for the value. If you then assign a larger number, Visual Basic will use a **Long** or (if it is very large or has a fractional component) **Double** representation.

Sometimes you want to use a specific representation for a number. For example, you might want a **Variant** variable to store a numeric value as **Currency** to avoid round-off errors in later calculations. Visual Basic provides several conversion functions that you can use to convert values into a specific type. To convert a value to **Currency**, for example, you use the **CCur** function:

```
PayPerWeek = CCur(hours * hourlyPay)
```

An error occurs if you attempt to perform a mathematical operation or function on a **Variant** that does not contain a number or something that can be interpreted as a number (such as a date/time value or a string containing a number). For this reason, you often want to determine if a **Variant** variable contains a value that can be used successfully as a number. The **IsNumeric** function performs this task:

```
Do
    anyNumber = InputBox("Enter a number")
Loop Until IsNumeric(anyNumber)
MsgBox "The square root is: " & Sqr(anyNumber)
```

When Visual Basic converts a representation that is not numeric (such as a string containing a number) to a numeric value, it uses the International settings (specified in the Windows Control Panel) to interpret the thousands separator, decimal separator, and currency symbol. Thus, if the country setting in the Control Panel is set to United States, Canada, or Australia, these two examples would return **True**:

```
IsNumeric("$100")
IsNumeric("1,560.50")
```

While these two would return **False**:

```
IsNumeric("DM100")
IsNumeric("1.560,50")
```

However, the reverse would be the case—the first two would return **False** and the second two **True**—if the country setting in the Control Panel was set to Germany.

If you assign a **Variant** containing a number to a string variable or property, Visual Basic converts the representation of the number to a string automatically. If you want to explicitly convert a number to a string, use the **CStr** function. You can also use the **Format** function to convert a number to a string that includes formatting such as currency, thousands separator, and decimal separator symbols. The **Format** function automatically uses the appropriate symbols according to the International settings dialog box in the Windows Control Panel.

For More Information For information on converting values, see the *Language Reference*, or search Help for *conversion*. For information on **Format**, see the *Language Reference*, or search Help for *Format*.

Strings Stored in Variants

Generally, storing and using strings in **Variant** variables poses few problems. As mentioned earlier, however, sometimes the result of the + operator can be ambiguous when used with two **Variant** values. If both of the **Variants** contain numbers, then the + operator performs addition. If both of the **Variants** contain strings, then the + operator performs string concatenation. But if one of the values is represented as a number and the other is represented as a string, the situation is more complicated. Visual Basic first attempts to convert the string into a number. If the conversion is successful, then the + operator adds the two values; if unsuccessful, it concatenates the number with the string to produce a new string.

When you want to be sure that concatenation occurs regardless of the representation of the value in the variables, use the & operator. For example, this code:

```
Sub Form_Click ()
    Dim X, Y

    X = "6"
    Y = "7"
    Print X + Y, X & Y
    X = 6
    Print X + Y, X & Y
End Sub
```

produces this result on the form:

```
67        67
13        67
```

Tip When typing your code, be careful to leave a space between any variable name and the & operator. If you do not leave a space, Visual Basic assumes you intended to use the & as the type-definition character for the variable name. Type-definition characters are discussed in the section "Other Fundamental Data Types" later in this chapter.

Date/Time Values Stored in Variants

In addition to strings and numbers, **Variant** variables can also contain date/time values. Several functions return date/time values. For example:

```
Sub Form_Click ()
    Dim rightnow, daysleft, hoursleft, minutesleft

    rightnow = Now                    ' Now returns the current date/time.
    daysleft = Int(DateSerial(Year(rightnow) + 1, 1, 1) - rightnow)
    hoursleft = 24 - Hour(rightnow)
    minutesleft = 60 - Minute(rightnow)
    Print daysleft & " days left in the year."
    Print hoursleft & " hours left in the day."
    Print minutesleft & " minutes left in the hour."
End Sub
```

You can perform math on date/time values as well. Adding or subtracting integers adds or subtracts days; adding or subtracting fractions adds or subtracts time. Thus, adding 20 adds 20 days, while subtracting 1/24 subtracts one hour, and so on.

The range for dates stored in **Variant** variables is January 1, 0000, to December 31, 9999. Calculations on dates don't take into account the calendar revisions prior to the switch to the Gregorian calendar, however, so calculations producing date values earlier than the year in which the Gregorian calendar was adopted (1752 in Britain and its colonies at that time; earlier or later in other countries) will be incorrect.

You can use date/time literals in your code by enclosing them with the number sign (#), in the same way you enclose string literals with double quotation marks (""). For example, you can compare a **Variant** containing a date/time value with a literal date:

```
If SomeDate > #3/6/93# Then
```

Similarly, you can compare a date/time value with a complete date/time literal:

```
If SomeDate > #3/6/93 1:20pm# Then
```

If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to midnight (the start of the day). If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to December 30, 1899.

Visual Basic accepts a wide variety of date and time formats in literals. These are all valid date/time values:

```
SomeDate = #3-6-93 13:20#
SomeDate = #March 27, 1993 1:20am#
SomeDate = #Apr-2-93#
SomeDate = #4 April 1993#
```

Note Notice that the interpretation of day and month is dependent on the International settings in the Windows Control Panel when the code actually runs. So if the Country option in the Control Panel is set to United States, the first example equates to March 6, but if the Country option is set to Australia, the first example equates to June 3. For this reason, you should avoid using numbers for both the day and month in date/time literals if your application may be used in different countries. Instead, use the name of the month, as in the latter three of the preceding examples.

In the same way that you can use the **IsNumeric** function to determine if a **Variant** variable contains a value that can be considered a valid numeric value, you can use the **IsDate** function to determine if a **Variant** contains a value that can be considered a valid date/time value. You can then use the **CVDate** function to convert the value into a date/time value. For example, the following code tests the Text property of a text box with **IsDate**. If the property contains text that can be considered a valid date, Visual Basic converts the text into a date and computes the days left until the end of the year:

```
Dim SomeDate, daysleft
If IsDate(Text1.Text) Then
    SomeDate = CVDate(Text1.Text)
    daysleft = DateSerial(Year(SomeDate) + 1, 1, 1) - SomeDate
    Text2.Text = daysleft & " days left in the year."
Else
    MsgBox Text1.Text & " is not a valid date."
End If
```

For More Information For information about the various date/time functions, see the *Language Reference*, or search Help for *Date*.

The Empty Value

Sometimes you need to know if a value has ever been assigned to a variable since the variable was created. A **Variant** variable has the Empty value before it is assigned a value. The Empty value is a special value different from 0, a zero-length string (""), or the null value. You can test for the Empty value with the **IsEmpty** function:

```
If IsEmpty(Z) Then Z = 0
```

When a **Variant** contains the Empty value, you can use it in expressions; it is treated as either 0 or a zero-length string depending on the expression.

The Empty value disappears as soon as any value is assigned to a **Variant** variable (including the value of 0, the zero-length string, and the null value). You can set a **Variant** variable back to the Empty value by assigning another Empty **Variant** variable to it.

The Null Value

The **Variant** data type can contain one other special value: **Null**. **Null** is commonly used in database applications to indicate unknown or missing data. Because of the way in which it is used in databases, **Null** has some unique characteristics:

- Expressions involving **Null** always result in **Null**. Thus, **Null** is said to "propagate" through expressions; if any part of the expression evaluates to **Null**, the entire expression evaluates to **Null**.
- Passing **Null**, a **Variant** containing **Null**, or an expression that evaluates to **Null** as an argument to most functions causes the function to return **Null**.
- **Null** values propagate through intrinsic functions that return **Variant** data types.

You can also assign **Null** with the **Null** keyword:

```
Z = Null
```

You can use the **IsNull** function to test if a **Variant** variable contains **Null**:

```
If IsNull(X) And IsNull(Y) Then
    Z = Null
Else
    Z = 0
End If
```

If you assign **Null** to a variable of any type other than **Variant**, a trappable error occurs. Assigning **Null** to a **Variant** variable doesn't cause an error, and **Null** will propagate through expressions involving **Variant** variables (though **Null** does not propagate through certain functions). You can return **Null** from any **Function** procedure with a **Variant** return value.

Tip The propagation of **Null** makes it useful as an error value. If you write **Function** procedures that return **Null** when an error occurs, and then combine these functions in expressions, you can use **IsNull** to test the final result of the expression to see if an error occurred. Because **Null** propagates, the final result will be **Null** if an error occurred in any of the functions; you don't have to test the result of each function individually.

Variables are not set to **Null** unless you explicitly assign **Null** to them, so if you don't use **Null** in your application, you don't have to write code that tests for and handles it.

For More Information For information on how to use **Null** in expressions, search Help for *Null*.

Other Fundamental Data Types

The **Variant** data type handles all types of fundamental data and converts between them automatically. If you want to create concise, fast code, however, you may want to use other data types where appropriate. For example, if a variable will always contain small integer values, you can save several bytes (and speed up your code slightly when performing arithmetic operations on the variable) by declaring that variable to be **Integer** instead of **Variant**.

There are seven fundamental data types in Visual Basic, including **Variant**, as shown in Table 7.2.

Table 7.2 Fundamental Visual Basic Data Types

Type name	Description	Type-declaration character	Range
Integer	2-byte integer	%	– 32,768 to 32,767
Long	4-byte integer	&	– 2,147,483,648 to 2,147,483,647
Single	4-byte floating-point number	!	–3.402823E38 to –1.401298E-45 (negative values)
			1.401298E-45 to 3.402823E38 (positive values)
Double	8-byte floating-point number	#	–1.79769313486232D308 to –4.94065645841247D-324 (negative values)
			4.94065645841247D-324 to 1.79769313486232D308 (positive values)
Currency	8-byte number with fixed decimal point	@	–922337203685477.5808 to 922337203685477.5807
String	String of characters	$	0 to approximately 65,500 characters
Variant	Date/time, floating-point number, or string	(None)	Date values: January 1, 0000 to December 31, 9999; numeric values: same range as **Double**; string values: same range as **String**

You can also declare arrays of any of these fundamental types. (For more information, see the section "Arrays" later in this chapter.)

Before using a non-**Variant** variable, you must use the **Dim, Global,** or **Static** statement to declare it **As** *datatype*. For example, the following statements declare an **Integer**, **Double**, **String**, and **Currency** type, respectively:

```
Dim I As Integer
Dim Amt As Double
Static YourName As String
Global BillsPaid As Currency
```

The **Dim** statement can combine multiple declarations, as in these statements:

```
Dim I As Integer, Amt As Double
Dim YourName As String, BillsPaid As Currency
Dim Test, J As Integer, Amount
```

Note If you do not supply a data type, the variable is given the default type. In the preceding example, the variables `Test` and `Amount` are of the **Variant** data type. This may surprise you if your experience with other programming languages leads you to expect all variables to have the same specified type (in this case, **Integer**).

Instead of declaring a type with **As**, you can identify the data type of a variable by appending a type-declaration character to the end of the variable name (see the preceding table). For example, `Count%` has type **Integer**, `X#` has type **Double**, and `MyName$` has type **String**. This method of specifying the type of a variable is available for compatibility with other versions of Basic.

If you don't use either method to declare a type, Visual Basic assumes the variable has the **Variant** data type—whatever type you specified as the default type with the **Def***type* statement.

For More Information For information on the **Def***type* statement, search Help for *Deftype*.

Numeric Types

If you know that a variable always stores whole numbers (such as 12) rather than numbers with a fractional amount (such as 3.57), declare it as an **Integer** or **Long** type. Operations are faster with integers, and these types consume less memory than a **Variant**. They are especially useful as the counter variables in **For...Next** loops.

If the variable contains a fraction, declare it as a **Single**, **Double**, or **Currency** variable. The **Currency** data type supports up to four digits to the right of the decimal point and 15 to the left; it is a fast and accurate fixed-point data type suitable for monetary calculations. Floating-point (**Single** and **Double**) numbers have much larger ranges than **Currency**, but can be subject to small rounding errors.

Note Floating-point values can be expressed as *mmm***E***eee* or *mmm***D***eee*, in which *mmm* is the mantissa and *eee* is the exponent (a power of 10). The highest positive value of a **Single** data type is 3.402823E+38, or 3.4 times 10 to the 38th power; the highest positive value of a **Double** data type is 1.79769313486232D+308, or about 1.8 times 10 to the 308th power. Using **D** to separate the mantissa and exponent in a numeric literal causes the value to be treated as a **Double** data type.

All numeric variables can be assigned to each other and to variables of the **Variant** type. Visual Basic rounds off (not truncates) the fractional part of a floating-point number before assigning it to an integer.

String Types

If you have a variable that will always contain a string and never a numeric value, you can declare it to be of type **String**:

```
Dim S As String
```

You can then assign strings to this variable and manipulate it using string functions:

```
S = "Database"
S = Left(S, 4)
```

By default, a string variable or argument is a *variable-length string*; the string grows or shrinks as you assign new data to it. You can also declare strings that have a fixed length. You specify a *fixed-length string* with this syntax:

String * *size*

For example, to declare a string that is always 50 characters long:

```
Dim EmpName As String * 50
```

If you assign a string of fewer than 50 characters, `EmpName` is padded with enough trailing spaces to total 50 characters. If you assign a string that is too long for the fixed-length string, Visual Basic simply truncates the characters. For example:

```
Dim String4 As String * 4, S As String
S = "Database"
Debug.Print S
String4 = S
Debug.Print String4
```

These two code blocks produce this output in the Immediate window:

```
Database
Data
```

Because fixed-length strings are padded with trailing spaces, you may find the **Trim** and **RTrim** functions useful when working with them.

For More Information For information on **Trim** or **RTrim**, search Help for *Trim* or *RTrim*.

Note The kind of **Option Compare** statement you use with a module—**Option Compare Binary** or **Option Compare Text**—determines the results of comparisons between strings, between strings and **Variant** variables, and between **Variant** variables containing strings. By default, Visual Basic uses **Option Compare Binary**, so string comparisons are case-sensitive. However, you can use the **Option Compare Text** statement to make Visual Basic perform case-insensitive comparisons. For more information, search Help for *Option Compare*.

Argument Data Types

The arguments for procedures you write have the **Variant** data type by default. However, you can declare other data types for arguments. For example, the following function accepts a string and an integer:

```
Function Reverse (S As String, ByVal n As Integer)
    ' Reverses the first n characters in S.
    Dim Temp As String, i As Integer

    If n > Len(S) Then n = Len(S)
    For i = n To 1 Step -1
        Temp = Temp & Mid(S, i, 1)
    Next
    Reverse = Temp & Right(S, Len(S) - n)
End Function
```

There are a couple of points to note about this. The first is that if you specify a data type for an argument, you must pass a value of that type for the argument. You cannot pass a **Variant** to a string argument, for example. This code will produce an error:

```
Dim V As Variant
V = "Testing"
Debug.Print Reverse(V, 4)       ' Error: argument type mismatch.
```

However, you can work around this by passing an expression, rather than a data type, for an argument. Visual Basic evaluates an expression and passes it as the required type if it can. The simplest way to turn a variable into an expression is to enclose it in parentheses:

```
Debug.Print Reverse((V), 4)     ' Turns the variable into an expression.
```

Another way to enable a **Variant** to be passed to procedures with arguments of specific types is to declare the arguments with the **ByVal** keyword, as illustrated by the second argument, *ByVal n As Integer*, in the preceding Reverse function example. Thus, you could pass a **Variant** as the second argument to this function:

```
Dim S As String, V As Variant
S = "Testing"
V = "2"
Debug.Print Reverse(S, V)           ' Works!
```

Passing Arguments by Value

The **ByVal** keyword does more than simply enable other data types to be passed to an argument of a particular data type. Its primary purpose is to specify that values are passed to that argument *by value* rather than *by reference*.

A variable's value can be changed by a procedure when it's passed to that procedure by reference. By contrast, only a *copy* of the variable is passed to the procedure when the variable is passed by value. If the procedure changes that value, the change affects only the copy and not the variable itself. This is important in the Reverse procedure; if the second argument is not declared by **ByVal**, bugs could appear in the code. For example, suppose the second argument in Reverse wasn't declared **ByVal** and you called it this way:

```
Dim I As Integer, S As String
I = 10
S = "Testing"
R = Reverse(S, I)
' Now I = 7 (the length of S).
```

You don't normally expect a function to modify its arguments, as happens here. To avoid this kind of side effect in any procedures that modify their arguments, declare those arguments with **ByVal**.

Function Data Types

Because Visual Basic functions return values, they, like variables, have a data type. Also like variables, functions have the **Variant** type by default. If you don't set a return value for the function (by assigning a value to the name of the function), the function returns a **Variant** containing the Empty value. However, a function can be declared to return a particular type. For example, the Reverse function could be declared to return a string:

```
Function Reverse (S As String, ByVal n As Integer) As String
```

In this example, if you don't assign a return value, this function will return a zero-length string (""). If a function is declared to return a numeric type (such as **Integer** or **Double**), it returns zero if you don't explicitly assign a return value.

As with variables, you usually don't have to declare a data type for a function because the default **Variant** data type handles most types of data automatically. As with variables, however, Visual Basic can work more efficiently with functions if you explicitly declare a data type for them.

Arrays

If you have programmed in other languages, you're probably familiar with the concept of arrays. *Arrays* allow you to refer to a series of variables by the same name and to use a number (an index) to tell them apart. This helps you create smaller and simpler code in many situations, because you can set up loops that deal efficiently with any number of cases by using the index number. Arrays have both upper and lower bounds, and the elements of the array are contiguous within those bounds. Because Visual Basic allocates space for each index number, avoid declaring an array larger than you need it to be.

Note The arrays discussed in this section are arrays of variables, declared in code. They are different from the control arrays you specify by setting the Index property of controls at design time. Arrays of variables are always contiguous; unlike control arrays, you cannot load and unload elements from the middle of the array. Conversely, you cannot use functions that work with arrays of variables, such as **UBound** and **LBound**, on control arrays.

All the elements in an array have the same data type. Of course, when the data type is **Variant**, the individual elements may contain different kinds of data (strings, numbers, or date/time values). You can declare an array of any of the fundamental data types, including user-defined types (described later in this chapter) and object variables (described in Chapter 8, "Objects and Instances").

There are three ways to declare a fixed-size array, depending on the scope you want the array to have:

- To create a *global array*, use the **Global** statement in the Declarations section of a code module to declare the array.
- To create a *module-level array*, use the **Dim** statement in the Declarations section of a code or form module to declare the array.
- To create a *local array*, use the **Static** statement in a procedure to declare the array (although you can use **Dim** if the whole procedure is declared **Static**). Local fixed-size arrays must be static. See the section "Static Variables" earlier in this chapter.

There are additional rules when you create a *dynamic array* (an array whose size can change at run time). Later in this chapter, the section "Dynamic Arrays" discusses these rules.

When declaring an array, follow the array name by the upper bound in parentheses. The upper bound must be an integer (in the range –32,768 to 32,767). For example, these array declarations can appear in the Declarations section of a code module:

```
Dim Counters(14) As Integer
Dim Sums(20) As Double
```

To create a global array, you simply use **Global** in place of **Dim**:

```
Global Counters(14) As Integer
Global Sums(20) As Double
```

The same declarations within a procedure use **Static**:

```
Static Counters(14) As Integer
Static Sums(20) As Double
```

The first declaration creates an array with 15 elements, with index numbers running from 0 to 14. The second creates an array with 21 elements, with index numbers running from 0 to 20. The default lower bound is 0. However, you can change the default lower bound to 1 by placing an **Option Base** statement in the Declarations section of a module:

```
Option Base 1
```

Another way to specify lower bounds is to provide it explicitly (as an integer, in the range –32,768 to 32,767) using the **To** keyword:

```
Dim Counters(1 To 15) As Integer
Dim Sums(100 To 120) As String
```

In the preceding declarations, the index numbers of `Counters` run from 1 to 15, and the index numbers of `Sums` run from 100 to 120.

Note Many other versions of Basic allow you to use an array without first declaring it. Visual Basic does not allow this; you must declare an array before using it.

Loops often provide an efficient way to manipulate arrays. For example, the following **For** loop initializes all elements in the array to 5:

```
Static Counters(1 To 15) As Integer
Dim I As Integer
For I = 1 To 15
    Counters(I) = 5
Next
```

Multidimensional Arrays

With Visual Basic, you can declare arrays of up to 60 dimensions. For example, the following statement declares a two-dimensional 10-by-10 array within a procedure:

```
Static MatrixA(9, 9) As Double
```

Either or both dimensions can be declared with explicit lower bounds:

```
Static MatrixA(1 To 10, 1 To 10) As Double
```

You can efficiently process a multidimensional array by using nested **For** loops. For example, these statements initialize every element in `MatrixA` to a value based on its location in the array:

```
Dim I As Integer, J As Integer
Static MatrixA(1 To 10, 1 To 10) As Double
For I = 1 To 10
    For J = 1 To 10
        MatrixA(I, J) = I * 10 + J
    Next J
Next I
```

You can extend this to more than two dimensions. For example:

```
Dim MultiD(3, 1 To 10, 1 To 15)
```

This declaration creates an array that has three dimensions with sizes 4 by 10 by 15. The total number of elements is the product of these three dimensions, or 600.

Note When you start adding dimensions to an array, the total storage needed by the array increases dramatically, so use multidimensional arrays with care. Be especially careful with **Variant** arrays, because **Variant** arrays are larger than other data types.

Dynamic Arrays

Sometimes you may not know exactly how large to make an array. You may want to have the capability of changing the size of the array at run time.

A dynamic array can be resized at any time. Dynamic arrays are among the most flexible and convenient features in Visual Basic, and they help you to manage memory efficiently. For example, you can use a large array for a short time and then free up memory to the system when you're no longer using the array.

The alternative is to declare an array with the largest possible size and then ignore array elements you don't need. However, this approach, if overused, might cause the operating environment to run low on memory.

▶ **To create a dynamic array**

1. Declare the array with a **Global** statement (if you want the array to be global) or **Dim** statement at the module level (if you want the array to be module level), or a **Static** or **Dim** statement in a procedure (if you want the array to be local). You declare the array as dynamic by giving it an empty dimension list. For example:

   ```
   Dim DynArray()
   ```

2. Allocate the actual number of elements with a **ReDim** statement. For example:

   ```
   ReDim DynArray(X + 1)
   ```

The **ReDim** statement can appear only in a procedure. Unlike the **Dim** and **Static** statements, **ReDim** is an executable statement—it makes the application carry out an action at run time.

The **ReDim** statement supports the same syntax described in the last few sections for fixed arrays. Each **ReDim** can change the number of elements as well as the lower and upper bounds for each dimension. However, the number of dimensions in the array cannot change.

For example, the dynamic array `Matrix1` is created by first declaring it at the module level:

```
Dim Matrix1() As Integer
```

A procedure then allocates space for the array:

```
Sub CalcValuesNow ()
    .
    .
    .
    ReDim Matrix1(19, 29)
End Sub
```

The **ReDim** statement allocates a matrix of 20 by 30 integers (at a total size of 600 elements). Alternatively, the bounds of a dynamic array can be set using variables:

```
ReDim Matrix1(X, Y)
```

When you create a dynamic array that is local to a procedure, declaring the array with a **Dim** or **Static** statement is recommended but not required. Doing so limits the number of dimensions to 8 (this is always the limit on the number of dimensions for module-level and global dynamic arrays). If you need more dimensions, avoid using **Dim** and create the array directly with **ReDim**:

```
ReDim BigArr(A, B, C, D, E, F, G, H, I) As Integer
```

Using this technique, you can have up to 60 dimensions in your dynamic array. However, the array is local to the procedure.

Preserving the Contents of Dynamic Arrays

Each time you execute the **ReDim** statement, all the values currently stored in the array are lost. Visual Basic resets the values to the Empty value (for **Variant** arrays), to zero (for numeric arrays), to a zero-length string (for string arrays), or to **Nothing** (for arrays of objects—for more information, see Chapter 8, "Objects and Instances").

This is useful when you want to prepare the array for new data, or when you want to shrink the array to take up minimal memory. However, sometimes you may want to change the size of the array without losing the data in the array. You can do this by using **ReDim** with the **Preserve** keyword. For example, you can enlarge an array by one element without losing the values of the existing elements with code like this:

```
ReDim Preserve MyArray(UBound(MyArray) + 1)
```

Only the upper bound of the last dimension in a multidimensional array can be changed when you use the **Preserve** keyword; if you change any of the other dimensions, or the lower bound of the last dimension, a run-time error occurs. Thus, you can do this:

```
ReDim Preserve Matrix(10, UBound(Matrix, 2) + 1)
```

But you cannot do this:

```
ReDim Preserve Matrix(UBound(Matrix, 1) + 1, 10)
```

For More Information For information about dynamic arrays, see the *Language Reference*, or search Help for *ReDim*.

Huge Arrays

Arrays whose total size (total number of elements times size of each element) exceeds 64K are known as *huge arrays*. You don't have to do anything special to create a huge array; Visual Basic creates a huge array automatically if you attempt to create an array whose size exceeds 64K. However, there are certain restrictions on huge arrays:

- You cannot create huge arrays of object variables.
- You cannot create huge arrays of variable-length strings (huge arrays of fixed-length strings are allowed, however).
- Huge arrays take up some additional memory over and above the memory required for their elements.

The maximum size for a huge array is 64 MB in enhanced-mode Windows and 1 MB in standard-mode Windows. Remember that you must use an integer to specify each dimension in the array, so the range for each dimension is –32,768 to 32,767. For more information about calculating the size of huge arrays, see Appendix D, "Specifications and Limitations."

User-Defined Types (Structures)

You can combine variables of several different types to create user-defined types (also known as *structs* in the C programming language and *records* in Pascal). User-defined types are useful when you want to create a single variable that records several related pieces of information.

You create a user-defined type with the **Type** statement, which must be placed in the Declarations section of a code module. User-defined types are always global, so you cannot use the **Type** statement in a form module. The variables that you declare as a user-defined type can be global, module-level, or local in form or code modules. For example, you could create a user-defined type that records information about a computer system:

```
' Declarations (of a code module)
Type SystemInfo
    CPU As Variant
    Memory As Long
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

You can declare local, module-level, or global variables of that type:

```
Dim MySystem As SystemInfo, YourSystem As SystemInfo
```

Assigning and retrieving values from the elements of this variable is similar to setting and getting properties:

```
MySystem.CPU = "486"
If MySystem.PurchaseDate > #1/1/92# Then
```

You can also assign one variable to another if they are both of the same user-defined type. This assigns all the elements of one variable to the same elements in the other variable:

```
YourSystem = MySystem
```

A user-defined type can contain an ordinary (fixed-size) array, but not a dynamic array. For example:

```
Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives(25) As String
    VideoColors As Integer
    Cost As Currency
    PurchaseDate As Variant
End Type
```

You can access the values in an array within a user-defined type in the same way that you access the values in a property array:

```
Dim MySystem As SystemInfo
MySystem.DiskDrives(0) = "1.44 MB"
```

You can also declare an array of user-defined types:

```
Dim AllSystems(100) As SystemInfo
```

Follow the same rules to access the components of this data structure:

```
AllSystems(5).CPU = "386SX"
AllSystems(X).DiskDrives(2) = "100M SCSI"
```

This nesting of data structures can get as complex as you like:

```
Type DriveInfo
    Type As String
    Size As Long
End Type

Type SystemInfo
    CPU As Variant
    Memory As Long
    DiskDrives(26) As DriveInfo
    Cost As Currency
    PurchaseDate As Variant
End Type

Dim AllSystems(100) As SystemInfo
AllSystems(1).DiskDrives(0).Type = "Floppy"
```

You can declare procedure arguments with a user-defined type:

```
Sub FillSystem (SomeSystem As SystemInfo)
    SomeSystem.CPU = lstCPU.Text
    SomeSystem.Memory = txtMemory.Text
    SomeSystem.Cost = txtCost.Text
    SomeSystem.PurchaseDate = Now
End Sub
```

However, you can't specify a user-defined type as the return type of a **Function** procedure. If you want a procedure to return a user-defined type, you must pass it as one of the arguments. User-defined types are always passed by reference, so the procedure can modify the argument and return it to the calling procedure, as illustrated in the previous example.

Tip Because the **Variant** data type can store many different types of data, a **Variant** array can be used in many situations where you might expect to use a user-defined type. A **Variant** array is actually more flexible than a user-defined type, since you can change the type of data you store in each element at any time, and you can make the array dynamic so you can change its size as necessary. However, a **Variant** array always uses more memory than an equivalent user-defined type.

Symbolic Constants

Often, you'll find that your code contains constant values that reappear over and over. Or you may find that the code depends on certain numbers that are difficult to remember—numbers that, in and of themselves, have no obvious meaning.

In these cases you can greatly improve the readability of your code—and make it easier to maintain—by using the **Const** statement. This statement allows you to assign a meaningful name, called a *symbolic constant*, in place of a number. While a constant definition somewhat resembles a variable, you can't modify a constant or assign a new value to it as you can to a variable.

The syntax for a **Const** statement is:

[Global] Const *constantname* = *expression*

The argument *constantname* is a valid symbolic name (the rules are the same as the rules for creating variable names), and *expression* is composed of numeric or string constants and operators (you can't use function calls in *expression*).

A **Const** statement can represent a mathematical or date/time quantity:

```
Const PI = 3.14159265
Global Const MAX_PLANETS = 9
Const RELEASE_DATE = #1/1/93#
```

The **Const** statement can also be used to define string constants:

```
Global Const VERSION = "07.10.A"
Const CODE_NAME = "Enigma"
```

Note The CONSTANT.TXT file included with Visual Basic provides many of the constants you may want to use in your code.

You can place more than one constant declaration on a single line, if you separate them with commas:

```
Global Const PI = 3.14, MAX_PLANETS = 9, WORLD_POP = 6E+09
```

The expression on the right side of the equal sign (=) is often a number or literal string, but it can also be an expression that results in a number or string (although that expression can't contain calls to functions). You can even define constants in terms of previously defined constants:

```
Const PI2 = PI * 2
```

Once you define constants, you can place them in your code to make it much more readable. For example:

```
Static SolarSystem(1 To MAX_PLANETS)
If numPeople > WORLD_POP Then Exit Sub
```

A **Const** statement has scope just as a variable declaration does, and the same rules apply:

- To create a constant that exists only within a procedure, declare it within that procedure.
- To create a constant available to all procedures within a module or form but not to any code outside that module or form, declare the constant in the Declarations section of the form or module.
- To create a constant available throughout the application, declare the constant in the Declarations section of a module and place the keyword **Global** before **Const**. Global constants, like global variables, cannot be declared in a form.

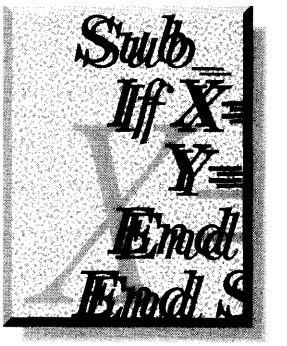
There is one complication to be aware of when you're using global constants. Because constants can be defined in terms of other constants, you must be careful not to set up a *cycle*, or circular reference between two or more constants. A cycle occurs when you have two or more global constants, each of which is defined in terms of the other. For example:

```
' In Module 1:
Global Const A = B * 2

' In Module 2:
Global Const B = A / 2
```

When a cycle occurs, Visual Basic generates an error when you attempt to run your application. You cannot run your code until you resolve the circular reference. To avoid problems like this, it is considered good programming practice to restrict all your global constants to a single module or, at most, a small number of modules.

CHAPTER 8

Objects and Instances



When you create an application in Visual Basic, you work with objects; you create form objects and draw control objects on those forms. The Visual Basic language extends your command over these objects by allowing you to declare and use variables in your code that represent objects. These *object variables* allow you to manipulate a form or control as easily as you manipulate an **Integer** or **String** variable. Using object variables, you can create multiple instances (copies) of forms at run time, write procedures that modify any form or control, and construct useful data structures, such as arrays of forms.

Contents

- An Introduction to Object Variables
- Declaring an Object Variable
- Using Object Variables
- Multiple Instances and **Me**
- The Forms and Controls Collections
- System Objects

MULTINST.MAK and OBJECTS.MAK

Many of the code examples in this chapter are taken from the MULTINST.MAK and OBJECTS.MAK sample applications. If you installed the sample applications, you will find these applications in the \OBJECTS subdirectory of the main Visual Basic directory (\VB\SAMPLES\OBJECTS).

An Introduction to Object Variables

Object variables provide a number of benefits; one of the most significant is that they allow you to create additional instances of forms in your application. An *instance* of a form is a copy that has an existence independent of the original. In the same way that you can run multiple instances of an application, your application can have multiple instances of its forms.

You can get a quick example of this by creating a form with a command button on it. In the Click event for the command button, add this code:

```
Sub Command1_Click ()
Dim F As New Form1      ' Declare form variable.
    F.Show              ' Load and display new instance.

    ' Move and color new instance.
    F.Move Left + (Width \ 10), Top + (Height \ 10)
    F.BackColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
End Sub
```

This code assumes the form containing the command button has the name `Form1`. The first line in the procedure declares a object variable of type `Form1`. The **New** keyword in the declaration specifies that Visual Basic will automatically create a new instance of the form `Form1` when the variable is used. The next line uses the variable with the **Show** method, so Visual Basic loads a new instance of the form and displays it. The rest of the code moves and changes the BackColor property of the new instance so it will be more visible.

Initially, the new instance is an exact copy of the original form as it existed at design time, so it appears in the same location as the original (and directly on top of the original if you haven't moved the original). The **Move** method moves the new instance so that the previous instances are visible. The Left, Top, Width, and Height properties used in the line of code with the **Move** method are the properties of the current instance (the one in which this procedure is executing), but the **Move** method applies to the new instance (the one referred to by the form variable). Thus the **Move** method places the new instance in a position relative to the current instance.

Figure 8.1 shows multiple instances of a form.

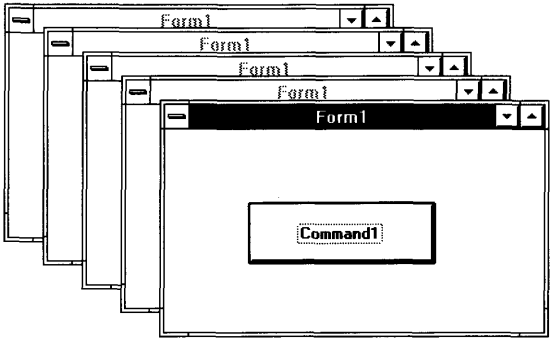


Figure 8.1 Creating multiple instances of a form

Declaring an Object Variable

You declare an object variable in the same way you declare other variables, with **Dim**, **ReDim**, **Static**, or **Global**. The only differences are the optional **New** keyword and the *objecttype*; both of these are discussed later in this chapter. The syntax is:

{Dim | ReDim | Static | Global} *variablename* **As [New]** *objecttype*

For example, you can declare an object variable that refers to a form in your application called frmMain:

```
Dim FormVar As New frmMain  ' frmMain must exist in your application.
```

You can also declare an object variable that can refer to any form in your application:

```
Dim anyForm As Form         ' Generic form variable
```

Similarly, you can declare an object variable that can refer to any text box in your application:

```
Dim anyText As TextBox      ' Can refer to any text box
                            ' (but only a text box).
```

You can also declare an object variable that can refer to a control of any type:

```
Dim anyControl As Control   ' Generic control variable
```

Notice that you can declare a form variable that refers to a specific form in your application, but you cannot declare a control variable that refers to a particular control. You can declare a control variable that can refer to a specific *type* of control (such as `TextBox` or `ListBox`), but not to one particular control of that type (such as `txtEntry` or `List1`). Object types are discussed in more detail later in this chapter.

Note You cannot include objects in a user-defined type.

Scope and Lifetime of Object Variables

The scope and lifetime of object variables, like those of other variables, depends on where in your code you declare them and the Visual Basic keywords you use when declaring them.

Declaration	Scope	Lifetime of object variable
Dim (within a procedure)	Local	While procedure is executing.
Static (within a procedure)	Local	Lifetime of the application (if declared in a code module); lifetime of the form instance (if declared in a form module).
Dim (in the Declarations section of a form or code module)	Module level	Lifetime of the application (if declared in a code module); lifetime of the form instance (if declared in a form module).
Global (in the Declarations section of a code module)	Global	Lifetime of the application.

If you make an entire procedure static by placing the **Static** keyword in the procedure heading, all local object variables in that procedure are static, even if they are declared with **Dim**.

For More Information For information on the scope and lifetime of variables, see Chapter 7, "Variables, Constants, and Data Types"; the *Language Reference*; or search Help for *Dim*.

Object Types

Just as other (data) variables have a data type, object variables have an object type. Unlike other variables, however, object variables can have either a generic or a specific type.

Generic and Specific Object Types

Generic object variables can refer to one of many specific types of objects. A generic form variable, for example, can refer to any form in your application; a generic control variable can refer to any control on any form in your application.

Specific object variables must refer to one specific type of object. A specific form variable can refer to only one form in your application (though it can refer to one of many instances of that form). Similarly, a specific control variable can refer to only one particular type of control in your application, such as `TextBox` or `ListBox`.

There are only three generic object types in Visual Basic. These generic object types, and the objects they can refer to, are listed in Table 8.1.

Table 8.1 Generic Object Types

Object type	Object referenced
Form	Any form in your application (including MDI children and the MDI form).
Control	Any control in your application.
MDIForm	The MDI form in your application (if your application has one).

Generic object variables are useful when you don't know in advance the specific type of object the variable will refer to at run time. For example, if you want to write code that can operate on any form in your application, you must use a generic form variable.

Note Since there can be only one MDI form in your application, there is no need to use the generic MDIForm type. Instead, you can use the specific MDIForm type (`MDIForm1`, or whatever you specified for the Name property of the MDI form) whenever you need to declare a form variable that refers to the MDI form. In fact, because Visual Basic can resolve references to properties and methods of specific form types before you run your application, you should always use the specific MDIForm type.

The generic MDIForm type is provided only for completeness; should a future version of Visual Basic allow multiple MDI forms in a single application, it might become useful.

Specific Form Object Types

In addition to the generic Form and MDIForm types, Visual Basic defines a specific form type for each form you create at design time. The names for these specific form types are determined by the Name property of the forms. So, for example, if you create a form at design time and give it the name frmMain, you can declare a variable of that type in your code:

```
Dim formReference As frmMain            ' frmMain must be a form
ReDim mainArray(10) As New frmMain      ' in your project.
```

Effectively, each form you create at design time becomes a form type you can use to declare specific form variables in your code.

If you want to set the Name property for a form to a name that is a reserved word in Visual Basic, you can enclose the name in square brackets ([ ]) when you declare a variable that refers to that form type. For example, if you give a form the name True, you must use square brackets to declare a variable of that type:

```
Dim formVar As New [True]        ' Form is named True, a reserved word.
```

For more information about using square brackets, see the section "Naming Conventions in Visual Basic" in Chapter 6, "Programming Fundamentals."

Specific Control Object Types

There is a specific control type for each control in Visual Basic. You can declare variables with any of these object types:

CheckBox	Grid	OptionButton
ComboBox	Frame	OLE
CommandButton	HScrollBar	PictureBox
CommonDialog	Image	Shape
Data	Label	TextBox
DirListBox	Line	Timer
DriveListBox	ListBox	VScrollBar
File List Box	Menu	

For example, you can declare a variable that can refer to any horizontal scroll bar control (but only horizontal scroll bar controls):

```
Dim hsbControl As HScrollBar
```

In addition, each custom control defines its own control type. For example, the OLE control defines its control type as OLE. You can determine the control type for a custom control by selecting the control and examining it in the Properties window. Visual Basic displays the control type for a custom control next to the name of the control in the Object box, as shown in Figure 8.2.

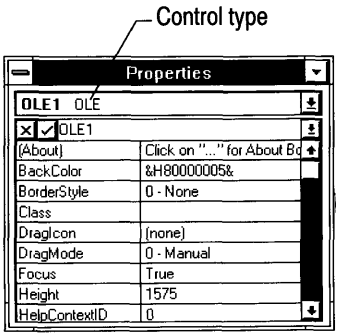


Figure 8.2 The Properties window displays the control type in the Object box.

You also use the control type when testing the type of a control variable with the **If...Type Of** statement. This is discussed in the section "Determining the Type of an Object Variable" later in this chapter.

Choosing Between Generic and Specific Object Variables

Whenever possible, you should declare your object variables with specific object types. When you use specific object types for variables and arguments, Visual Basic can resolve references to the properties and methods of those objects when you attempt to run the application. If you use generic object types, Visual Basic must resolve references to their properties and methods every time it encounters them at run time—a significantly slower process. Moreover, for specific object variables, Visual Basic detects errors in the use of properties and methods as soon as you attempt to run your application. For generic object variables, Visual Basic cannot detect many of these errors until it encounters them at run time.

However, there are some situations where you must use a generic object variable. If a form variable might refer to two or more different forms in your application, you must declare it as a generic form variable. If a control variable might refer to more than one type of control, you must declare it as a generic control variable. In addition, if you are going to assign a generic object reference (such as ActiveForm or ActiveControl) to an object variable, you must declare it as a generic object variable.

For More Information The Visual Basic Professional Edition defines additional object types you can use in your code. To use these object types, you must have version 2.0 (or later) of the Professional Edition. For more information, see the *Professional Features* manual.

The New Keyword

While declaring object variables is similar to declaring variables of other types, there is one key difference: You can use the **New** keyword when you declare a variable of a specific form type.

When you declare a form variable with **New**, you specify that Visual Basic should always ensure that an instance of the form exists when you use the form variable. (This is sometimes called "auto-instantiation.") Visual Basic automatically creates a new instance of the form whenever the form variable doesn't already refer to one.

In the example at the beginning of this chapter, a local form variable was declared with the **New** keyword:

```
Sub Command1_Click ()
Dim F As New Form1      ' Declare form variable.
    F.Show              ' Load and display new instance.
    .
    .
    .
End Sub
```

Because the form variable is declared with the **New** keyword, Visual Basic loads a new instance of the specific form type (in this case, `Form1`) the first time a property, method, or control on the form is referenced. This occurs in the third line of the preceding code, when the **Show** method is executed.

The form variable is a local variable, so it ceases to exist when the procedure ends. The loaded form instance it refers to still exists, and continues to exist until it is closed or unloaded. The next time the procedure is executed, the local form variable comes into existence again—but it is newly initialized at this point and does not refer to any form. In the third line, when the form variable is used with the **Show** method, Visual Basic automatically creates another new instance of the form.

If you do not declare the variable with **New**, any attempt to refer to a property, method, or control on the form causes an error. Such a variable is still useful, however, because you can use the **Set** statement to refer to a form instance that already exists. The **Set** statement is discussed in the section "Assigning Object Variables" later in this chapter.

Note You can declare only specific form variables with **New**. You cannot declare generic form variables with **New**, and you cannot use **New** to declare generic or specific control variables.

Implicit Form Variable Declaration

Visual Basic automatically declares a global form variable for each form in your application. These implicitly declared variables have the same name as the form. For example, if you add a form called frmMain to your application at design time, Visual Basic effectively declares a variable with the same name when you run your application:

```
Dim frmMain As New frmMain      ' Implicit form variable declaration
                                ' performed by Visual Basic; you never
                                ' see this.
```

These variables are what you've been using when you use the methods and manipulate the properties of a form by using its name in code. You might have thought that code like this operated directly on the form:

```
frmMain.Show 1
If frmMain!txtFileName.Text = "" Then End
Unload frmMain
```

But in fact, this code is using the implicitly declared variable with the same name as the form. This fact may be only an interesting detail that has no impact on the code you write, but it can lead to bugs in your code if you forget to refer to one of your form variables and refer to this implicit form variable instead. This is an easy mistake to make. Because your form variables are of type `frmMain`, for example, it is easy to forget the name of your variable and refer to the form type instead, resulting in code like this:

```
Dim F As frmMain
    F.Caption = "This is a new instance"
    ' Intervening code makes you forgetful.
    frmMain.Show
```

Perhaps what you really meant to write was this:

```
Dim F As frmMain
    F.Caption = "This is a new instance"
    ' Intervening code makes you forgetful.
    F.Show
```

Until you discover the problem, the code in the first case may leave you wondering why the caption on the displayed form instance has not changed.

You must also keep this detail in mind if you are adding multiple instances to code you wrote in Visual Basic 1.0. In Visual Basic 1.0, your code could refer *only* to the implicitly declared form variables. As you add form variables, you must remember to change your code to use them instead of the implicit form variables.

Note Visual Basic always creates these implicit form variables; **Option Explicit** has no effect on this behavior.

Arrays of Objects

You can declare and use arrays of an object type just as you declare and use an array of any data type. These arrays can be fixed-size or dynamic, but you cannot create huge arrays of objects (huge arrays contain more than 64K of data).

Arrays of Form Variables

You can declare an array of forms with **Dim**, **ReDim**, **Static**, or **Global** in the same way you declare an array of any other type. If you declare the array with the **New** keyword, Visual Basic automatically creates a new instance of the form for each element in the array as you use the elements in the array. For example, you might declare a global array of forms of a specific type, or a module-level array of generic forms:

```
Global Document() As New frmNotePad      ' Global dynamic form array
Dim WorkingForms(10) As Form             ' Generic form array
```

You then use the elements of the form array the same way you use any form variable:

```
Dim i As Integer
    For i = 0 to UBound(Document)
        Document(i).BackColor = QBColor(i Mod 16)
        Document(i).Show
    Next i
```

For a more extensive example using an array of form variables, see Chapter 14, "Multiple-Document Interface (MDI) Applications."

Arrays of Control Variables

You can declare an array of controls with **Dim**, **ReDim**, **Static**, or **Global** in the same way you declare an array of any other type. Unlike form arrays, however, control arrays cannot be declared with the **New** keyword. You can declare an array to be a specific control type:

```
ReDim ActiveImages(10) As Image
```

When you declare an array to be a particular control type, you can assign only controls of that type to the array. In the case of the preceding declaration, you can assign only image controls to the array. However, you can assign image controls on different forms to the array.

Alternatively, you can declare an array of generic control variables. For example, you might want to keep track of every control that was dropped onto a particular control, and not allow a control to be dropped onto it if one has been dropped onto it once already. You can do this by maintaining a dynamic array of control variables that contains references to each control that has been dropped:

```
Sub List1_DragDrop (Source As Control, X As Single, Y As Single)
Static newIndex As Variant, droppedCtrls() As Control
Dim i As Integer
    ' The first time, the array is uninitialized and newIndex is empty.
    If IsEmpty(newIndex) Then
        newIndex = 0
    Else
        For i = LBound(droppedCtrls) To UBound(droppedCtrls)
            If droppedCtrls(i) Is Source Then
            ' Control is in array, so it has already been dropped here.
                Beep
                Exit Sub
            End If
        Next i
        newIndex = UBound(droppedCtrls) + 1
    End If
    ' Enlarge array.
    ReDim Preserve droppedCtrls(newIndex)
    ' Set new element to the control that was dropped.
    Set droppedCtrls(newIndex) = Source
    ' Add value of the control to the list in the list box.
    List1.AddItem Source
End Sub
```

This example uses the **Is** operator to compare the variables in the control array with the control argument and the **Set** statement to assign the argument to an element in the array. Both of these keywords are explained in the section "Using Object Variables" later in this chapter.

Arrays of Control Variables Compared to Control Arrays

You might be confused about the difference between arrays of control variables and control arrays. Remember that you create a control array by setting the Index property of a control at design time; you can then use **Load** and **Unload** to add or remove any element in a control array, sometimes leaving "gaps" in the sequence of index values.

In contrast, you create arrays of control variables the same way you create arrays of any other kind of variable—with **Dim**, **ReDim**, **Global**, or **Static**—and all the elements in an array of control variables must be contiguous (there can be no "gaps"). Table 8.2 summarizes these differences.

Table 8.2 Differences Between Arrays of Control Variables and Control Arrays

Action	Array of control variables	Control array
Create array elements	**Dim**, **ReDim**, **Global**, or **Static**	Set the Index property at design time
Increase or decrease number of elements	**ReDim Preserve**	Load or unload controls at run time
Calculate largest and smallest element	**UBound** and **LBound**	Must track separately
Make elements contiguous	Yes	No
Range of index values	–32,768 to 32,767	0 to 32,767

Using Object Variables

Once you have declared an object variable, you can use it as you would an equivalent object: set and retrieve properties, execute methods, and so on. In addition, there are other operations you can perform on object variables. You can pass object variables to procedures, make object variables refer to other objects, compare two object variables, and test the type of an object variable.

Passing Objects to Procedures

You can pass objects as arguments to procedures. For example, the first argument in the DragDrop and DragOver event procedures is a generic control argument:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
```

You can define any of the arguments in your procedures to be generic or specific object types. Objects are always passed by reference, so you can change the properties of the object. For example, this procedure changes the visibility of any control passed to it (except controls that do not have a Visible property).

```
Sub FlipVisibility (C As Control)
    C.Visible = Not C.Visible
End Sub
```

The return value of a **Function** procedure can't be an object; it must be a fundamental data type, such as **Integer** or **Variant**.

Passing Generic and Specific Object Types

When an argument is a generic object type (Form or Control), you can pass a generic or a specific object variable for this argument. For example, you can pass any specific form variable for an argument declared **As Form**. Likewise, you can pass any control reference or specific control variable for an argument declared **As Control**.

On the other hand, if an argument is declared with a specific object type, you can pass only object variables or references of that specific type for that argument. For example, you can pass only list box variables to this procedure:

```
Sub ResizeList (lbAny As ListBox)
```

Likewise, you can pass only specific object variables of type `frmMain` to this procedure:

```
Sub UpdateForm (f As frmMain)
```

Notice that it is the object type specified when the variable is declared, and not the object that the variable refers to, that determines whether you can pass a variable to an argument declared with a specific type. In the following code, you can pass the first variable to the UpdateForm procedure, but not the second:

```
Dim frm As New frmMain, anyFrm As Form
    frm.Show
    Set anyFrm = frm
    UpdateForm frm          ' This works.
    UpdateForm anyFrm       ' This causes a type-mismatch error.
```

If you pass `Screen.ActiveForm` and `MDIForm.ActiveForm` to a procedure, the argument must be declared **As Form**. Even if you know that you'll always pass a specific form type to the procedure, Visual Basic requires that you declare the argument with the generic form type. Likewise, if you pass `Screen.ActiveControl` or `MDIForm.ActiveControl` to a procedure, the argument must be declared **As Control**.

Determining the Type of An Object Variable

To determine the type of an object variable, you use an **If...TypeOf** and **ElseIf...TypeOf** statement, which has this syntax:

{**If** | **ElseIf**} **TypeOf** *object* **Is** *objecttype* **Then**

The *object* can be any object variable, and *objecttype* can be any generic or specific object type. For example, you can test for the type of control that was dropped on a picture box:

```
Sub PicTarget_DragDrop (Source As Control, X As Single, Y As Single)
    If TypeOf Source Is Image Then
        picTarget.Picture = Source.Picture
    ElseIf TypeOf Source Is TextBox Then
        picTarget.CurrentX = picTarget.ScaleLeft
        picTarget.CurrentY = picTarget.ScaleTop
        picTarget.Print Source.Text
    End If
End Sub
```

Note You can use the **If...TypeOf** statement to determine the control type of a custom control. Use the syntax **If TypeOf** *object* **Is** *objecttype*. The identifier you use for *object* is the class name of the custom control. See the section "Specific Control Object Types" earlier in this chapter for more information.

You can test for only one object type at a time. In the previous example, if you want to handle additional object types for `Source`, you must add additional **ElseIf TypeOf** statements, even if you want to perform the same action. You cannot write code like this:

```
If TypeOf Source Is Image Or TypeOf Source Is PictureBox Then
```

Instead, you must add additional **ElseIf TypeOf** statements:

```
Sub PicTarget_DragDrop (Source As Control, X As Single, Y As Single)
    If TypeOf Source Is Image Then
        picTarget.Picture = Source.Picture
    ElseIf TypeOf Source Is PictureBox Then
        picTarget.Picture = Source.Picture
    ElseIf TypeOf Source Is TextBox Then
        picTarget.CurrentX = picTarget.ScaleLeft
        picTarget.CurrentY = picTarget.ScaleTop
        picTarget.Print Source.Text
    End If
End Sub
```

Likewise, you cannot combine the **TypeOf** syntax with other conditions on the same line. However, an **If TypeOf** or **ElseIf TypeOf** statement can be part of a larger **If** block that tests for other conditions. For example:

```
If X = 3 Then
    Print X
ElseIf TypeOf Source Is Timer Then
    ThisControl.Interval = 1000
End If
```

Sometimes you may want to take an action if a control is not a particular type. You can't combine the **Not** keyword with **TypeOf**, but you can produce the same effect by including an empty statement block combined with an **Else** clause. For example, the following code sets the Visible property to **False** if `ThisControl` is not a timer:

```
If TypeOf ThisControl Is Timer Then
Else
    ThisControl.Visible = False
End If
```

Alternatively, you can simply exit the procedure if a timer is detected:

```
If TypeOf ThisControl Is Timer Then Exit Sub
ThisControl.Visible = False
```

You cannot use **TypeOf** with **Select Case**.

Assigning an Object Variable

You assign a different object to an object variable with the **Set** statement:

Set *objectvar* = *objectreference*

Use the **Set** statement whenever you want to make an object variable refer to a different object. For example, your application might have several instances of a form that contains a timer. You want to keep track of the instance that has the enabled timer (your application allows only one instance at a time to have an enabled timer). You can do this by creating a global form variable:

```
Global frmEnabledTimer As frmMain       ' Tracks instance of frmMain
                                        ' with enabled timer.
```

Then, in the code that enables the timer control, you use the **Set** statement to make this form variable refer to the instance that has the enabled timer:

```
Set frmEnabledTimer = Me
```

Now, whenever you want to refer to the form that has the enabled timer, you can use this variable. For example, you might minimize this form instance:

```
frmEnabledTimer.WindowState = 1
```

Before you can use a generic object variable, you must use the **Set** statement to make it refer to a specific object. For example:

```
Dim anyForm As Form
    Set anyForm = Screen.ActiveForm
```

Sometimes you may use object variables, and particularly control variables, simply to shorten the code you have to type. For example, you might have code like this:

```
If frmAccountDisplay!txtAccountBalance.Text < 0 Then
    frmAccountDisplay!txtAccountBalance.BackColor = 0
    frmAccountDisplay!txtAccountBalance.ForeColor = 255
End If
```

You can shorten this code significantly if you use a control variable:

```
Dim Bal As TextBox
Set Bal = frmAccountDisplay!txtAccountBalance
If Bal.Text < 0 Then
    Bal.BackColor = 0
    Bal.ForeColor = 255
End If
```

Assigning Nothing

When an object variable does not refer to any object, it has the special value **Nothing**. When you declare an object variable, it is **Nothing** initially. If you declare a form variable with **New**, it is **Nothing** until you refer to a property, method, or control on the form; at that point, Visual Basic creates a new form instance, and the form variable refers to this instance instead of **Nothing**.

You assign **Nothing** to any object variable with the **Set** statement:

```
Set frmEnabledTimer = Nothing
```

It is good programming practice to assign **Nothing** to global, module-level, or static local object variables when you are no longer using them. Each object variable uses memory and system resources; these are freed when you assign **Nothing** to the variable. Local object variables (other than static variables) are set to **Nothing** automatically when the procedure ends.

Understanding Object Variable Assignment

There is a key difference between using the **Set** statement with an object variable and assigning a value to a simple variable (such as an **Integer** or **Variant** variable). Object variables are *references* to objects, rather than copies of those objects. So if you use **Set** to make several object variables refer to the same object, the variables don't contain independent copies of that object; instead, they are all *references* to the *same* object. The following code illustrates this difference:

```
Dim X, Y
X = "Visual"
Y = X
X = "Basic"
```

In this code, `X` and `Y` are **Variant** variables. Each gets a separate copy of the value "Visual." When `X` is changed, you would not expect the value of `Y` to be affected, and indeed, it is not. However, consider this similar code involving object variables:

```
Dim X As Control, Y As Control
Set X = frmMain!txtEdit
Set Y = X
X.Text = "Basic"
```

If you check the Text property of `Y`, you will find it has changed as well. `X` and `Y` are not independent copies of the control; both are simply references to the actual control. Changing the value of the control or any of its properties with an object variable changes the actual object. The change is reflected in all other object variables that refer to that control. This characteristic of object variables ensures that your code works with the most up-to-date values in the object, regardless of whether they've been changed by other code or by the user.

Tip If you have programmed in C or some other programming languages, you may find it helpful to think of object variables as pointers. While there are some differences, object variables are similar to pointers in that several object variables can refer or "point to" the same object, and changes to the object are reflected in all the object variables that refer to it. Just as C pointers can be NULL, object variables can be set to **Nothing** (and have that value initially).

C and most other languages do not have an equivalent to an object variable declared with the **New** keyword. An object variable declared with **New** is like a pointer: It ensures that the object it points to always exists—by loading a new instance of the object, if necessary—when you try to use the variable. This protects you from errors caused by NULL pointers, but it also means you can't test the variable for the **Nothing** value.

Creating New Instances with Set

You can use **New** with the **Set** statement to create a new instance of a form. The syntax for this version of the **Set** statement is:

Set *objectvar* = **New** *objecttype*

For example, you might declare a global generic form variable:

```
Global anyForm As Form
```

Elsewhere in your code you use the **New** keyword with the **Set** statement to make Visual Basic ensure that this variable now always refers to an instance of a specific form:

```
Set anyForm = New frmMain       ' Set variable to auto-instantiate.
anyForm.Show                    ' Display new instance.
```

Notice that declaring a form variable with **New**, like this:

```
Dim frmInstance As New frmMain
```

is equivalent to declaring the variable without **New** and then using **New** with the **Set** statement:

```
Dim frmInstance As Form
    Set frmInstance = New frmMain
```

Using **New** with the **Set** statement is not exactly the same as declaring a variable with **New**, however. If you declare a variable with **New**, Visual Basic always ensures that the variable refers to a loaded form instance. Even if you assign **Nothing** to the variable, Visual Basic loads a new form instance the next time you attempt to use the properties, methods, or controls on the form.

By contrast, if you declare a form variable without **New** and then use **New** with the **Set** statement to create a new form instance, you can assign **Nothing** to the variable, and Visual Basic won't attempt to create another form instance the next time you attempt to use the variable.

```
Dim frmAuto As New frmMain
    frmAuto.Show
    Set frmAuto = Nothing
    If frmAuto Is Nothing Then          ' Never true
    .
    .
    .
```

```
Dim frmOther As Form
    Set frmOther = New frmMain
    frmOther.Show
    Set frmOther = Nothing
    If frmOther Is Nothing Then         ' Always true
    .
    .
    .
```

Because the variable in the first example is declared with **New**, you can never test it for **Nothing** (even immediately after setting it to **Nothing**). The variable in the second example can be tested for **Nothing** because it is not declared with **New**.

Testing Object References with Is

Sometimes you need to compare two object references to see if they both refer to the same object. You use the **Is** operator to compare two object variables. For example, you might want to test a global form variable to see if it refers to the form that is being unloaded:

```
Sub Form_QueryUnload (Cancel As Integer, UnloadMode As Integer)
    If frmEnabledTimer Is Me Then Set frmEnabledTimer = Nothing
End Sub
```

You can test control variables in the same way. For example, you can write code in a DragDrop event procedure to compare the *Source* argument with the control where the drop is taking place, and then generate an error message if they both refer to the same control:

```
Sub imgTool_DragDrop (Source As Control, X As Single, Y As Single)
    If Source Is imgTool Then
        MsgBox "Cannot drop control on itself."
    Else
        ' Do something with dropped control.
    End If
End Sub
```

Sometimes you may want to take an action if two object variables do *not* refer to the same object. You do this as you would expect, by combining the **Not** keyword with the comparison:

```
Sub imgTool_DragDrop (Source As Control, X As Single, Y As Single)
    If Not (Source Is imgTool) Then
        If TypeOf Source Is Image Then
            imgTool.Picture = Source.Picture
        End If
    End If
End Sub
```

Testing for Nothing

You can also use **Is** to test for the special value **Nothing**. When an object variable does not refer to any object, it **Is Nothing**. For example, this code compares a form variable with **Nothing**:

```
Sub Form_Load ()
    If frmMostRecent Is Nothing Then
        Set frmMostRecent = Me
    End If
End Sub
```

Form variables declared with **New** cannot be tested for **Nothing**. When you declare a form variable with **New**, you are specifying that Visual Basic should ensure that the form variable always refers to a loaded form instance whenever you use that form variable. So even if you set this form variable to **Nothing**, Visual Basic will load a new form instance when you use it in the comparison with **Nothing**:

```
Dim frmMostRecent As New frmMain
    If frmMostRecent Is Nothing Then
        ' This code is never executed because a form declared
        ' with New never Is Nothing.
    End If
```

If you want to be able to test a form variable for **Nothing**, declare it without **New**. You can then use the **Set** statement to make the variable refer to a specific form. (Use **New** with the **Set** statement if you want Visual Basic to create a new instance when you use the form variable.)

Sometimes you may want to take an action if an object variable is *not* **Nothing**. You do this by combining the **Not** keyword with the test for **Nothing**:

```
If Not(frmMostRecent Is Nothing) Then
    ' frmMostRecent is not Nothing.
    frmMostRecent.Show
End If
```

Multiple Instances and Me

When you use the **New** keyword (either when you declare a form variable or when you assign a form variable with the **Set** statement), you create an *auto-instantiating* form variable. The first time you attempt to use the variable, Visual Basic creates a new instance of the form. Each instance of a form has a separate existence from all other instances. Each instance begins its existence with its properties set to the values the original form type had at design time, but once you create an instance, you can change its properties independent of any other instance. You can think of the original form at design time as a "prototype" for all the instances of that form; the original form is the cookie cutter, and the instances are the cookies.

All instances of a particular form type share the same code (the event procedures and general procedures in the form module), but each instance has its own copy of the data for the form (properties, controls, and module-level and static local variables). Because each instance gets its own copy of data, you cannot use module-level variables to share data between the instances of a form type. Instead, you must use global variables.

When an event occurs on a form instance, the code in the appropriate event procedure executes *in the context* of that instance, using the data for that instance. Most of the time the code does not need to distinguish the instance in which it is running. Sometimes, however, when running in the context of a particular instance, the code for the form needs to refer to that instance explicitly. For example, the **Unload** statement requires a form reference. If an event procedure on the form closes and unloads the form, it must refer to the correct instance in the **Unload** statement. It can do this using the **Me** keyword.

For example, the code to close and unload the form might look like this:

```
Sub cmdClose_Click ()
    Unload Me       ' Unload this instance.
End Sub
```

When multiple instances of a form exist, **Me** always refers to the instance in which the code is currently running. This is usually the same as `Screen.ActiveForm`, because the form must be the active form for most events to occur. But **Me** is not always the same as `Screen.ActiveForm`. For example, a timer event can occur for a timer control, even when the timer is on a form that is not the active form. Thus in timer events (and link events when performing DDE), a reference to **Me** may refer to a different form than `Screen.ActiveForm`. In these cases, **Me** always refers to the instance of the form in which the code is executing.

You use **Me** just as you would use a specific form variable. You can compare it with the **Is** operator and assign it to other form variables with the **Set** statement:

```
If Not (frmTrack Is Me) Then
    Set frmTrack = Me
End If
```

You can also pass **Me** to procedures that accept a form as an argument. **Me** is "read-only," however; you cannot assign other form variables to **Me**.

Lifetime of Form Instances

Notice that the variable you use to refer to an instance of a form type and the instance itself have separate existences. If the form variable is local to a procedure (and not static), the variable disappears when the procedure ends, but the loaded instance continues to exist. At this point, you may not have an object variable that refers to the instance, but you can still refer to the instance through `Screen.ActiveForm` or `MDI.ActiveForm` (if the instance is the active form), **Me** (in the code for the instance), or through the Forms collection (described later in this chapter).

Visual Basic keeps a *reference count* for each form instance. Each variable that refers to a particular instance increases the reference count for that instance. As you set one of these variables to **Nothing**, Visual Basic decrements the reference count. The reference count is incremented when the instance is loaded, and decremented when the instance is unloaded. If the reference count reaches 0, the instance no longer exists and Visual Basic reclaims the memory and system resources it used, including the memory used to store the values in static and module-level variables in that instance of the form. Thus, static and module-level variables in form modules don't necessarily remain in existence for the lifetime of the application. If you assign **Nothing** to all the variables that refer to a specific instance, and then unload the instance, the static and module-level variables for that instance cease to exist.

If you have a variable that refers to a form instance, you can unload the form instance and then load it again (either explicitly, with the **Load** statement, or implicitly, by referring to one of the form's properties, methods, or controls). Each time you unload a form instance, all of its properties (and the properties of the controls on that form) are set back to the values that the original form type had at design time.

```
Dim frmNew As New FrmMain
    frmNew.Caption = "Test"     ' Instance is implicitly loaded.
    frmNew.Show                 ' Display instance.
    Unload frmNew               ' Unload instance.
    frmNew.Show                 ' Implicitly reload instance.
' Notice that Caption is reset to design-time value.
```

Although the properties of an unloaded form instance and its controls are reset to their design-time values, the module-level variables in the form are not reinitialized—unless there are no variables referring to the form instance.

When you unload a form instance, you might assume that the instance has ceased to exist and Visual Basic has reclaimed the memory and system resources used by the instance. This is the case only if there are no form variables referring to that instance. If there are variables referring to that instance, it still occupies some memory, even when unloaded. In addition, each object variable uses some memory and system resources, regardless of whether the object to which it refers is visible or loaded. If you no longer need an object variable, set it to **Nothing**.

Warning Avoid assigning **Me** to a static or module-level form variable in the form instance referred to by **Me**. If you do this, and the form instance is later unloaded, the static or module-level form variable becomes "orphaned"; it is still in existence but cannot be accessed.

An orphaned form variable consumes system resources until the application ends. If you find it necessary to create a static or module-level form variable that refers to the form instance that contains it, be sure to assign **Nothing** to the form variable before the form is unloaded (in the QueryUnload event, for example).

The Forms and Controls Collections

Visual Basic provides collections that you can use to access all the forms in your application, or all the controls on a particular form. A *collection* is similar to a zero–based array of objects, except:

- You don't declare a collection (it is "pre-declared" by Visual Basic).
- You can't use **UBound** or **LBound** with a collection.
- You can't pass a collection to a procedure.
- Each collection has a Count property.

There is one global Forms collection in your application. Each form has its own Controls collection.

The Forms Collection

The Forms collection is global; you can refer to it from anywhere in your code. The Forms collection contains only loaded form instances. There may be other unloaded forms in your application, but they do not appear in the Forms collection until they are loaded. Each loaded instance of a form appears as a separate form in the Forms collection.

The syntax for referring to a form array is:

Forms(*index*)

The argument *index* is a numeric expression. The Forms collection is zero–based, and the Forms collection's Count property returns the number of forms in the collection, so *index* must be in the range 0 to `Forms.Count-1` (inclusive).

You can use the Forms collection wherever you can use a form variable, but the Forms collection is particularly useful whenever you want to iterate through all the loaded forms. For example, you might want to fill a list box with the captions of all loaded forms:

```
Dim i As Integer
    ' Refill list (in case an instance was added or removed).
    lstForms.Clear
    For i = 0 To Forms.Count - 1
        lstForms.AddItem Forms(i).Caption
    Next
```

There is no permanent association between a form and its position in the Forms collection. The forms occur in the collection in an unpredictable order, and the order changes as you load and unload form instances. For example, you might try to write code like this to unload all the loaded form instances:

```
Function UnloadAll () As Integer
Dim i As Integer
    For i = 0 To Forms.Count - 1
        Unload Forms(i)
    Next
    Quit = Forms.Count
End Function
```

This code fails, however, because as you unload each form, the others change their position in the collection, so there is no guarantee that this code will reference each form in the collection. Moreover, as you unload forms, the number of forms in the collection decreases, so eventually the code causes an error when it attempts to use an index that is greater than the number of the forms now in the collection.

If you want to maintain a reference to a particular form, you can't rely on the integer value that specifies its position in the Forms collection. Instead, you must use a form variable and use the **Set** statement to assign it to the form.

Note The Forms collection is a predefined name, not a reserved word. You can shadow the Forms collection by declaring a local variable called `Forms` in a procedure. Within that procedure, any reference to `Forms` will access the variable, not the Forms collection, so the Forms collection will be unavailable in that procedure. For this reason, and because you may find it confusing to read code that uses a variable called `Forms`, declaring a variable with this name is not recommended.

The Controls Collection on Each Form

In the same way that the Forms collection allows you to refer to any of the loaded forms, the Controls collection allows you to refer to any of the loaded controls on a form. The syntax for the Controls collection is:

[*formname*].**Controls**(*index*)

The *formname* is optional if you are referring to the Controls collection for a form within that form's code. If you want to refer to the Controls collection from another form or code module, you must specify the *formname*. Of course, the *formname* can be any form variable that refers to a loaded form instance.

All loaded controls on a form are included in the Controls collection. When there are control arrays on the form, each loaded element of each array is included in the Controls collection. The order of the controls in the collection is arbitrary, and it changes as elements of control arrays are loaded and unloaded.

The Controls collection has a Count property, and you use a number between 0 and one less than Count to refer to a particular control. This allows you to iterate through all of the controls on a form. For example, you can change the color of every control on a form with a procedure like this:

```
Sub ChangeColors (frmAny As Form, NewColor)
Dim i As Integer
    frmAny.BackColor = NewColor
    On Error Resume Next    ' Some controls don't have BackColor.
    For i = 0 To frmAny.Controls.Count - 1
        frmAny.Controls(i).BackColor = NewColor
    Next i
End Sub
```

If you specify *formname*, you can actually omit `Controls` and simply specify an index surrounded by parentheses. For example, these two statements are equivalent:

```
frmAny.Controls(i).BackColor = NewColor
frmAny(i).BackColor = NewColor          ' Can omit Controls.
```

However, you may find this confusing because the second statement looks like you are setting the BackColor property for one element in an array of forms. For this reason, explicitly specifying the Controls array is recommended.

As with the Forms collection, there is no permanent association between a particular control and its position in the Controls collection. If you want to keep track of a particular control on a form, use a control variable rather than the position of the control in the Controls collection.

System Objects

Visual Basic provides five special system objects, listed in Table 8.3, that enable your application to manipulate and obtain information about the environment in which it is running.

Table 8.3 System Objects

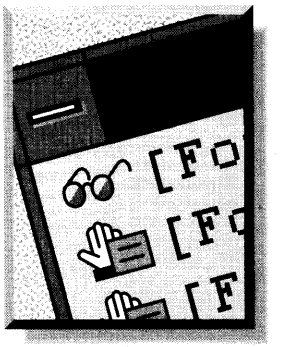
Object	Use
App	Supplies information specific to the application.
Clipboard	Provides access to the operating environment Clipboard.
Debug	Enables printing to the Debug window.
Printer	Enables printing text and graphics to the printer.
Screen	Supplies the current form, control, and other screen-related data.

You cannot declare object variables for any of these system objects, and you cannot pass the system objects to a procedure. Because the system objects are global, however, you can use them in code anywhere in your application.

For More Information For information on the Application object, see the *Language Reference*, or search Help for *applications*. For information on the Clipboard object, see Chapter 17, "Interacting with the Environment." The Debug object is used only with the **Print** method; it allows you to print values to the Debug window when you're debugging code as described in Chapter 9, "Debugging." The Printer object is described in Chapter 16, "Displaying and Printing Information," and the Screen object is discussed in Chapter 15, "Creating Graphics for Applications."

CHAPTER 9

Debugging



Visual Basic provides tools to help analyze how your application operates. These debugging tools are particularly useful in locating the source of errors (bugs). But you can also use them to experiment with changes to your application or to learn how other applications work.

This chapter shows how to use the debugging tools included in Visual Basic.

Contents

- Approaches to Debugging
- Design Time, Run Time, and Break Mode
- Using Break Mode
- Running Selected Portions of Your Application
- Using the Calls Dialog
- Using the Debug Window
- Monitoring Data with Watch Expressions
- Testing Data and Procedures with the Immediate Pane
- Special Considerations in Debugging Events
- Testing and Using Command-Line Arguments
- Avoiding Bugs

Approaches to Debugging

The debugging techniques presented in this chapter are actually a set of analysis tools. Visual Basic cannot diagnose or fix errors for you, but it does provide tools to help you analyze how execution flows from one part of the procedure to another, and how variables and property settings change as statements are executed. Debugging tools are a way of looking inside your application; they can help you determine what happens and why.

Visual Basic debugging support includes breakpoints, break expressions, watch expressions, single stepping, procedure stepping, and displays of the values of variables and properties. Visual Basic also includes special debugging features like edit-and-continue capability, setting the next statement to execute, and procedure testing while the application is halted.

Kinds of Errors

To understand how debugging is useful, consider the three kinds of errors you can encounter:

- Compile errors
- Run-time errors
- Logic errors

Compile Errors

Compile errors result from incorrectly constructed code. You may have incorrectly typed a keyword, omitted some necessary punctuation, or used a **Next** statement without a corresponding **For** statement. Visual Basic detects these errors when you compile the application.

Compile errors include errors in syntax. For example, you could have a statement like this:

```
Left
```

Left is a valid word in the Visual Basic language, but without an object it doesn't meet the syntax requirements for that word. Visual Basic detects syntax errors as soon as you enter them in the Code window if you have set the Syntax Checking option to "Yes." The default for Syntax Checking is "Yes."

Run-Time Errors

Run-time errors occur (and are detected by Visual Basic) when a statement attempts an operation that is impossible to carry out. A good example is division by zero. Suppose you have this statement:

```
Speed = Miles / Hours
```

If the variable `Hours` contains zero, the division is an invalid operation, even though the statement itself is syntactically correct. The application must run before it can detect this error.

You can include code in your application to trap and handle run-time errors when they occur. For information on dealing with run-time errors, see Chapter 10, "Handling Run-Time Errors."

Logic Errors

Logic errors occur when an application doesn't perform the way it was intended. An application can have syntactically valid code, run without performing any invalid operations, and yet produce incorrect results. Only by testing the application and analyzing results can you verify that the application is performing correctly.

How Debugging Tools Help

Debugging tools are designed to help you with:

- Logic and run-time errors.
- Observing the behavior of code that has no errors.

For instance, an incorrect result may be produced at the end of a long series of calculations. In debugging, the task is to determine where something went wrong. Perhaps you forgot to initialize a variable, chose the wrong operator, or used an incorrect formula.

There are no magic tricks to debugging, and there is no fixed sequence of steps that works every time. Basically, debugging helps you understand what's going on while your application runs. Debugging tools give you a snapshot of the current state of your application, including:

- Appearance of the user interface (UI).
- Values of variables, expressions, and properties.
- Active procedure calls.

The better you understand how your application is working, the faster you can find any bugs.

Debugging Tools on the Toolbar

Among its many debugging tools, Visual Basic provides several buttons on the toolbar that are very helpful. Figure 9.1 shows these tools.

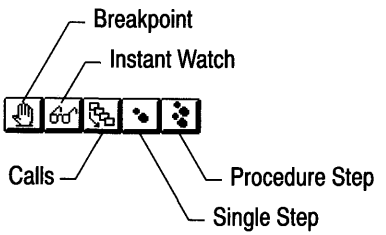


Figure 9.1 Debugging tools on the toolbar

The following table briefly describes the purpose of each tool. The sections in this chapter note instances when each of these tools can help you debug or analyze an application more efficiently.

Debugging tool	Purpose
Breakpoint	Defines a line in the Code window where Visual Basic breaks execution of the application.
Instant Watch	Displays the current value of an expression while the application is in break mode.
Calls	Displays a dialog box while in break mode that shows all procedures that have been called but not yet run to completion.
Single Step	Executes the next executable line of code in the application.
Procedure Step	Executes the next executable line of code in the application without stepping through procedure calls.

Design Time, Run Time, and Break Mode

To test and debug an application, you need to understand which of three modes you are in at any given time. You use Visual Basic at design time to create an application and at run time to run it. This chapter introduces *break mode*, which suspends running of the program so you can examine and alter data.

Identifying the Current Mode

The Visual Basic title bar always shows you the current mode. Figure 9.2 shows the title bar for design time, run time, and break mode.

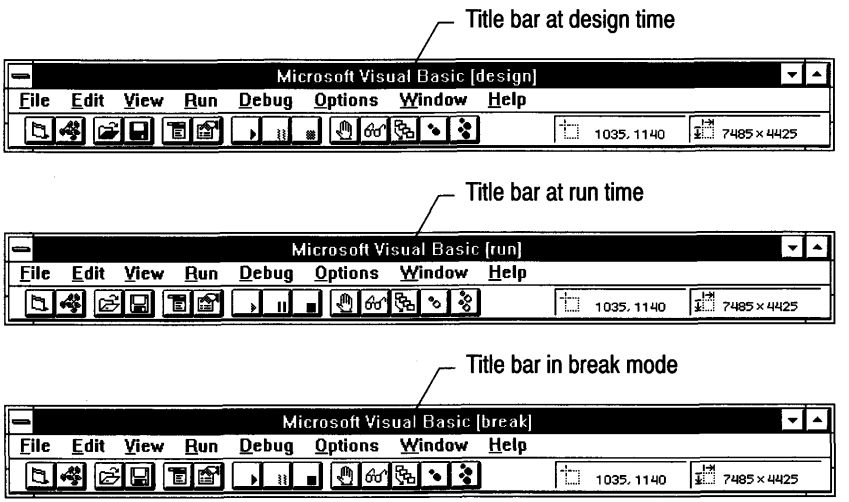


Figure 9.2 Identifying the current mode with the title bar

Notice that the current mode also determines which debugging tools are available; the unavailable buttons appear dimmed on the toolbar.

The characteristics of the three modes are listed below.

Mode	Description
Design time	Most of the work of creating an application is done here. You can design forms, draw controls, write code, and use the Properties window to set or view property settings. You cannot execute code or use debugging tools, except for setting breakpoints and creating watch expressions. Choose Start from the Run menu or click the Run button to switch to run time. If your application contains code that executes when the application starts, choose Single Step from the Debug menu (or press F8) to place the application in break mode at the first executable statement.
Run time	When this application takes control, you interact with the application the same way a user would. You can view code, but you cannot change it. Choose End from the Run menu or click the End button to switch back to design time.
Break mode	Choose Break from the Run menu, click the Break button, or press CTRL+BREAK to switch to break mode. Execution is suspended while running the application. You can view and edit code (by choosing Code from the View menu), examine or modify data, restart the application, end execution, or continue execution from the same point. You can set breakpoints and watch expressions at design time, but other debugging tools work only in break mode.

Using the Toolbar to Change Modes

The toolbar provides three buttons that let you change quickly from one mode to another. These buttons appear in Figure 9.3.

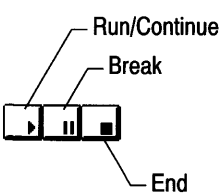


Figure 9.3 Run/Continue, Break, and End buttons on the toolbar

Whether any of these buttons is available depends on whether Visual Basic is in run time, design time, or break mode. The following table lists the buttons available for different modes.

Mode	Toolbar buttons available
Design time	Run/Continue
Run time	Break, End
Break	Run/Continue, End

Using Break Mode

At design time, you can change the design or code of an application, but you cannot see how your changes affect the way the application runs. At run time, you can watch how the application behaves, but you cannot directly change the way it operates.

Break mode halts the operation of an application and gives you a snapshot of its condition at any moment. Variable and property settings are preserved, so you can analyze the current state of the application. You can also make changes that affect how the application runs. When an application is in break mode, you can:

- Observe the condition of the application's interface.
- Determine which active procedures have been called.
- Control which statement the application will run next.
- Watch the values of variables, properties, and statements.
- Change the values of variables and properties.
- Run Visual Basic statements immediately.
- Modify code in the application.
- Manually control the operation of the application.

Entering Break Mode at a Problem Statement

When debugging, you may want the application to halt at the place in the code where the problem probably started. This is one reason Visual Basic provides breakpoints and **Stop** statements. A *breakpoint* defines a statement or set of conditions at which Visual Basic automatically stops execution and puts the application in break mode without running the statement containing the breakpoint.

Entering Break Mode Automatically

Visual Basic enters break mode when any of the following occurs:

- A statement on the line generates an untrapped run-time error.
- Execution reaches a line with a breakpoint.
- Execution reaches a **Stop** statement.
- A break expression defined in the Add Watch dialog box changes or becomes true, depending on how you defined it.

Entering Break Mode Manually

Visual Basic also switches into break mode if you do any of the following while the application is running:

- Press CTRL+BREAK.
- Choose Break from the Run menu.
- Choose the Break button on the toolbar.

It's possible to break execution when the application is idle (when it is between processing of events); when this happens, execution does not stop at a specific line, but Visual Basic switches to break mode anyway.

Displaying Variables in Break Mode

You can display variables only if they have the appropriate scope relative to the current statement. Whenever Visual Basic enters break mode automatically, the Code window appears and identifies:

- The currently active form or module (indicated in the Code window's title bar).
- The currently executing procedure (displayed in the window).
- The next statement to be executed (identified by a rectangular outline).

When you place an application in break mode manually, the Debug window appears.

Fixing a Run-Time Error and Continuing

Some run-time errors result from simple oversights when entering code; these errors are easily fixed. Type mismatches that involve properties are run-time errors—for example, when you try to assign a string value to a variable that expects an integer data type.

Figure 9.4 shows a run-time error message.

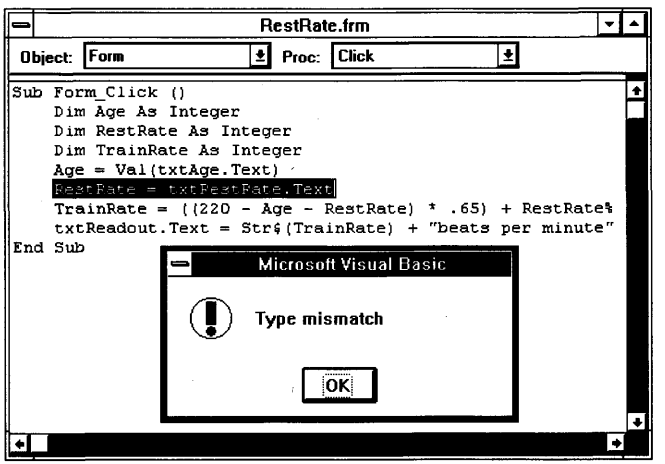


Figure 9.4 Run-time errors such as "Type mismatch" halt execution.

The solution for a type mismatch is to fix the problem statement so that it places the correct type of data (numeric) into the variable `RestRate`:

```
RestRate = Val(txtRateText.Text)
```

Then you can continue program execution with the same line that halted the application, even though you've changed some of the code; simply choose Continue from the Run menu or click the Run button on the toolbar. As you continue running the application, you can verify that the problem is fixed.

Some changes (most commonly, changing variable declarations or adding variables) require you to restart the application. When this happens, Visual Basic displays a message that asks if you want to restart the application.

Using a Breakpoint to Selectively Halt Execution

At run time, a breakpoint tells Visual Basic to halt just before a specific line of code. When Visual Basic is executing a procedure and it encounters a line of code with a breakpoint, it switches to break mode.

You can set or remove a breakpoint in break mode or at design time.

▶ **To set or remove a breakpoint**

1. In the Code window, move the insertion point to the line of code where you want to set or remove a breakpoint.
2. From the Debug menu, choose Toggle Breakpoint.

   –or–

   Click the Breakpoint button on the toolbar.

   –or–

   Press F9.

When you set a breakpoint, Visual Basic highlights the selected line in bold text, using the colors that you've specified with the Environment command in the Options menu.

For example, Figure 9.5 shows a procedure with a breakpoint on the fifth line. In the Code window, Visual Basic indicates a breakpoint by displaying the text on that line in bold and in the colors specified for a breakpoint.

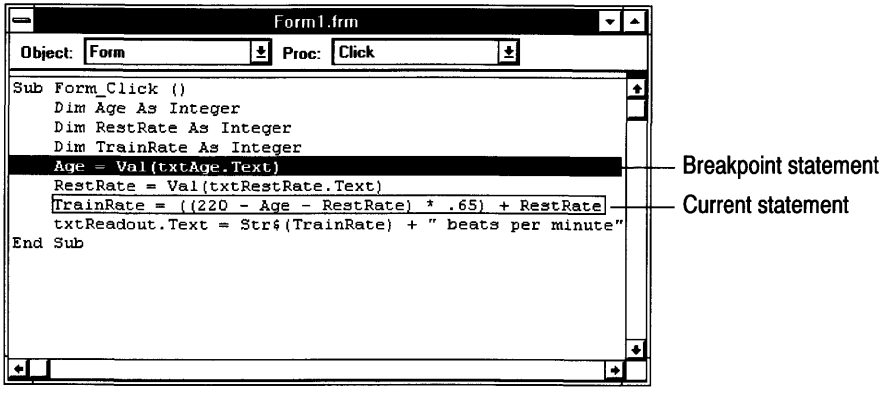


Figure 9.5 A procedure halted by a breakpoint

Identifying the Current Statement

In Figure 9.5, a rectangular outline surrounds the seventh line of code. This outline indicates the *current statement*, or next statement to be executed. When the current statement also contains a breakpoint, only the rectangular outline highlights the line of code. Once the current statement moves to another line, the line with the breakpoint is displayed in bold and in color once again.

You can specify the color of the text and background for the current statement by choosing the Environment command from the Options menu and then selecting the Next Statement Text and Next Statement Background options.

Examining the Application at a Breakpoint

Once you reach a breakpoint and the application is halted, you can examine the application's current state. Checking results of the application is easy, because you can move the focus among the forms of your application, the Code window, and the Debug window.

A breakpoint halts the application just before executing the line that contains the breakpoint. If you want to observe what happens when the line with the breakpoint executes, you must execute at least one more statement. Single stepping, discussed later in this chapter, can accomplish this task.

When you are trying to isolate a problem, remember that a statement may be indirectly at fault because it assigns an incorrect value to a variable. You may want to examine the value of variables and properties while in break mode, using watch expressions or the Debug window. The section "Monitoring Data with Watch Expressions" later in this chapter explains how to define and use watch expressions. The section "Testing Data and Procedures with the Immediate Pane" explains how to use the Debug window.

Using Stop Statements to Enter Break Mode

Placing a **Stop** statement in a procedure is an alternative to setting a breakpoint. Whenever Visual Basic encounters a **Stop** statement, it halts execution and switches to break mode. Although **Stop** statements act like breakpoints, they aren't set or cleared the same way.

Note There are important differences between **Stop** statements and breakpoints. If you leave and reload the current project, all breakpoints are cleared. But **Stop** statements stay in the code until you remove them. **Stop** statements also remain in the application when you make an executable (.EXE) file, where they act just like **End** statements. Breakpoints aren't included in an .EXE file.

Remember that a **Stop** statement does nothing more than temporarily halt execution, while an **End** statement halts execution and returns to design mode. You can always choose Continue from the Run menu to continue running the application.

For More Information For information on the **Stop** statement, see the *Language Reference*, or search Help for *Stop*.

Running Selected Portions of Your Application

If you can identify the statement that caused an error, a single breakpoint may help you locate the problem. More often, however, you know only the general area of the code that caused the error. A breakpoint helps you isolate that problem area. You can then use single stepping and procedure stepping to observe the effect of each statement. If necessary, you can also skip over statements or back up by starting execution at a new line.

Single Stepping Through Statements

Single stepping is the process of executing one statement at a time. After stepping through each statement, you can see its effect by looking at your application's forms or the Debug window.

▶ **To step through code one statement at a time**

- Choose Single Step from the Debug menu.

  –Or–

- Click the Single Step button on the toolbar.

  –Or–

- Press F8.

During a single-step command, Visual Basic temporarily switches to run time, executes the current statement, and advances to the next statement. Then it switches back to break mode.

Note Visual Basic allows you to single step between individual statements, even if they are on the same line. A line of code can contain two or more statements, separated by a colon (:). Visual Basic uses a rectangular outline to indicate which of the statements will execute next. Breakpoints apply only to the first statement of a line.

Procedure Stepping

Procedure stepping is identical to single stepping, except when the current statement contains a call to a procedure. Unlike the Single Step command, which steps into the called procedure, the Procedure Step command executes it as a unit and then steps to the next statement in the *current* procedure. Suppose, for example, that the statement calls the procedure SetAlarmTime:

```
SetAlarmTime 11, 30, 0
```

If you choose the Single Step command, the Code window displays the SetAlarmTime procedure and sets the current statement to the beginning of that procedure. This is the better choice only if you want to analyze the code within SetAlarmTime.

If you use the Procedure Step command, the Code window continues to display the current procedure. Execution advances to the statement immediately after the call to SetAlarmTime, unless SetAlarmTime contains a breakpoint or a **Stop** statement. Do this if you want to stay at the same level of code and don't need to analyze the SetAlarmTime procedure.

Tip If you inadvertently single step into a lengthy procedure call, it can take a long time to get back to the procedure you left. You can simulate a procedure step by placing a breakpoint at the end of the procedure you stepped into and then clicking the Run/Continue button on the toolbar. Once you reach the breakpoint, continue single stepping to return to the original procedure.

You can alternate freely between the Single Step and Procedure Step commands. The command you use depends on which portions of code you want to analyze at any given time.

▶ **To use procedure stepping**

- Choose Procedure Step from the Debug menu.

  –Or–

- Click the Procedure Step button on the toolbar.

  –Or–

- Press SHIFT+F8.

Setting the Next Statement to Be Executed

While debugging or experimenting with an application, you may want to skip a certain section of code—for instance, a section that contains a known bug—so you can continue tracing other problems. Or you may want to return to an earlier statement to retest a part of the application using different values for properties or variables.

Visual Basic lets you set a different line of code to execute next, provided it falls within the same procedure. The effect is similar to single stepping, except single stepping executes only the next line of code in the procedure. By setting the next statement to execute, you choose which line executes next.

▶ To set the next statement to be executed

1. Move the insertion point to the line of code you want to execute next.
2. From the Debug menu, choose Set Next Statement.
3. To resume execution, click the Run/Continue, Single Step, or Procedure Step button on the toolbar.

Note You must be in break mode to set the next statement to be executed. The Set Next Statement command is not available at design time or run time.

Using the Calls Dialog

The Calls dialog displays a list of all active procedure calls. Active procedure calls are the procedures in the application that were started but not completed. Figure 9.6 shows the Calls dialog.

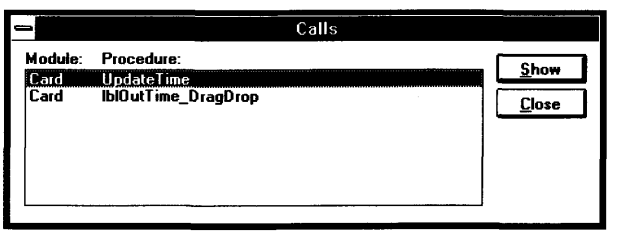


Figure 9.6 The Calls dialog

▶ To display the Calls dialog

1. Put the application in break mode.

   You can display the Calls dialog only when the application is in break mode.
2. Choose the Calls command from the Debug menu.

   –or–

   Click the Calls button on the toolbar.

   –or–

   Press CTRL+L.

The Calls dialog helps you trace the operation of an application as it executes a series of procedures, especially nested procedures. For example, an event procedure can call a second procedure, which can call a third procedure—all before the event procedure that started this chain is completed. Such nested procedure calls can be difficult to follow.

Note If you put the application in break mode during an idle loop, no entries appear in the Calls dialog.

Tracing Nested Procedures

The Calls dialog lists all the active procedure calls in a series of nested calls. It places the earliest active procedure call at the bottom of the list and adds subsequent procedure calls to the top.

The information given for each procedure begins with the file name of the module or the name of the form containing the procedure, followed by the name of the called procedure. Because the Calls dialog doesn't indicate the variable assigned to an instance of a form, it does not distinguish between multiple instances. For more information on multiple instances of a form, see Chapter 8, "Objects and Instances."

You can use the Calls dialog to display the statement in a procedure that passes control of the application to the next procedure in the list.

▶ **To display the statement that calls another procedure in the Calls dialog**

1. In the Calls dialog, choose the procedure call you want to display.
2. Choose the Show button.

   The procedure appears in the Code window.

The cursor location in the Code window indicates the statement that calls the next procedure in the Calls dialog list. If you choose the current procedure in the Calls dialog, the cursor appears at the current statement.

Using the Debug Window

Sometimes you can find the cause of a problem by executing portions of code. But more often you'll have to analyze what's happening to the data as well. You might isolate a problem in a variable or property with an incorrect value. Then you must determine how and why that variable or property was assigned an incorrect value.

The Debug window, which appears only when the application is in break mode, lets you monitor the values of expressions and variables while stepping through the statements in your application. You can also use the Debug window to change the value of variables and properties in break mode to see how different values affect your application. Figure 9.7 shows the Debug window.

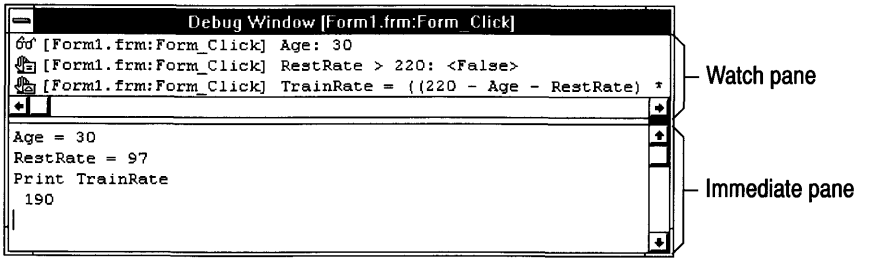


Figure 9.7 The Debug window

The Debug window consists of two panes. The *Watch pane* (above) displays the current watch expressions—expressions whose values you decide to monitor as the application runs. The *Immediate pane* (below) displays information as a result of debugging code in your application or which you request by typing commands directly in the pane. (The Immediate pane acts just like the Immediate window found in Visual Basic version 1.0.)

The title bar of the Debug window indicates the current form or module and the current procedure. So you always know which form or module is current, even if you move the focus between Code windows. If you use the same name for variables with different scope, the scope of variables in the Debug window changes when you move the focus between forms or Code windows.

The current procedure and form (or module) determine which variables can be displayed according to the scoping rules presented in Chapter 7, "Variables, Constants, and Data Types." For example, suppose the Debug window indicates that Form1 is the current form. In this case, you can display any of the form-level variables of Form1. You can also use the **Print** method to examine local variables of the procedure displayed in the Code window. (You can always examine the value of a global variable.)

Monitoring Data with Watch Expressions

As you debug your application, you may find that a problem occurs only when a certain variable or property assumes a particular value or range of values. Or you may learn that a calculation isn't producing the desired result. Many debugging problems aren't immediately traceable to a single statement, so you may need to observe the behavior of a variable or expression throughout a procedure.

Visual Basic monitors *watch expressions*—expressions that you define—for you. When the application enters break mode, these watch expressions appear in the Watch pane of the Debug window, where you can observe their values

You can also direct watch expressions to put the application into break mode whenever the expression's value changes or equals a specified value. For example, instead of single stepping through perhaps tens or hundreds of loops, you can use a watch expression to put the application in break mode when a loop counter reaches a specific value, . Or you may want the application to enter break mode each time a flag in a procedure changes value.

Adding a Watch Expression

You can add a watch expression at design time or in break mode. You use the Add Watch dialog box (shown in Figure 9.8) to add watch expressions.

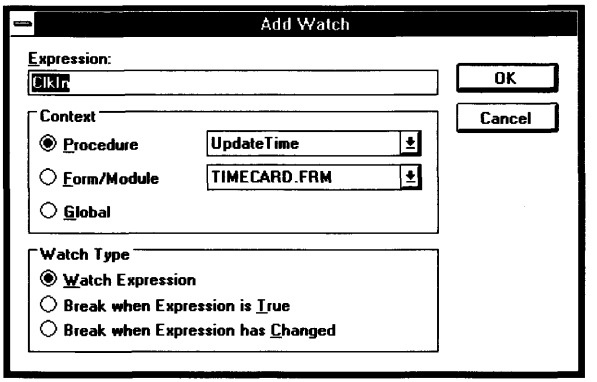


Figure 9.8 The Add Watch dialog box

The following table describes the Add Watch dialog box.

Control	Description
Expression text box	Where you enter the expression that the watch expression evaluates. The expression can be a variable, a property, a function call, or any other valid expression.
Context option group	Sets the scope of variables watched in the expression. Use if you have variables of the same name with different scope. You can also restrict the scope used to watch variables defined for a specific procedure, a specific form or module, or globally in the application. Visual Basic can evaluate a variable in a narrow context more quickly.
Watch Type option group	Sets how Visual Basic responds to the watch expression. Visual Basic can watch the expression and display its value in the Debug window when the application enters break mode. Or you can have the application enter break mode automatically when the expression evaluates to a true (nonzero) statement or each time the value of the expression changes.

▶ To add a watch expression

1. From the Debug menu, choose Add Watch.
2. Enter the expression to evaluate in the Expression text box.
3. If necessary, select an option button in the Context group to set the scope of variables watched.

   If you select the Procedure or Form/Module option, select a procedure, form, or module name from the appropriate list box.
4. If necessary, select an option button in the Watch Type group to determine how you want Visual Basic to respond to the watch expression.
5. Choose OK.

Editing or Deleting a Watch Expression

You can edit or delete any watch expression you select. To select a watch expression, choose the Edit Watch command from the Debug menu. The Watch dialog box shown in Figure 9.9 appears. This dialog box lists all the current watch expressions. You can then select a watch expression from the Current Expressions list to edit or delete.

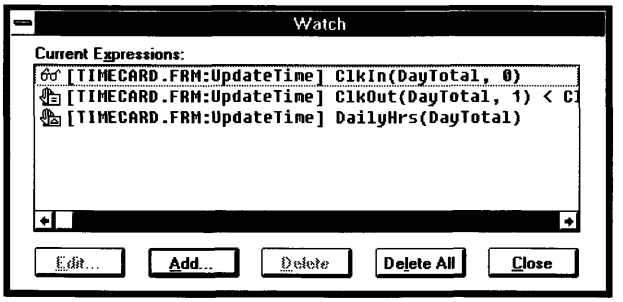


Figure 9.9 The Edit Watch dialog box

▶ To edit a watch expression

1. From the Debug menu, choose the Edit Watch command.
2. From the Current Expressions list, select the watch expression you want to edit.
3. Choose Edit.

   The Edit Watch dialog box appears, identical to the Add Watch dialog box except for the title bar.
4. Make any changes to the expression, the scope for evaluating variables, or the watch type.
5. Choose OK.

Note You can also edit any watch expression displayed in the Debug window by double-clicking the watch expression in the Watch pane. Visual Basic displays the selected watch expression in the Edit Watch dialog box.

▶ **To delete a watch expression**

1. From the Debug menu, choose the Edit Watch command.
2. From the Current Expressions list, select the watch expression you want to delete.
3. Choose Delete to delete a single watch expression.

   –or–

   Choose Delete All to delete all the current watch expressions.

Identifying Watch Types

At the left edge of each watch expression in the Edit Watch dialog and the Watch pane of the Debug window is an icon that identifies the watch type of that expression. Figure 9.10 defines the icon for each of the three watch types.

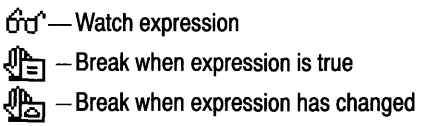


Figure 9.10 Watch type icons

Using Instant Watch

While in break mode, you may want to check the value of an expression for which you have not defined a watch expression. You can check such expressions with the Instant Watch dialog, shown in Figure 9.11.

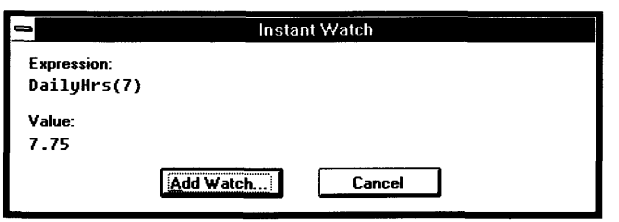


Figure 9.11 Instant Watch dialog

The Instant Watch dialog displays the value of the expression you select from either the Code window or the Immediate pane of the Debug window. To continue watching this expression, click the Add Watch button; the Add Watch dialog, with relevant information from the Instant Watch dialog already entered, is displayed. If the value of the current expression cannot be evaluated, the Add Watch button is not available.

▶ **To display an instant watch**

1. Select an expression in either the Code window or the Immediate pane of the Debug window.
2. Choose the Instant Watch button on the toolbar.

   –or–

   Press SHIFT+F9.

   –or–

   From the Debug menu, choose the Instant Watch command.
3. If you want to add a watch expression based on the expression in the Instant Watch dialog, choose the Add Watch button.

Testing Data and Procedures with the Immediate Pane

Sometimes when you are debugging or experimenting with an application, you may want to test procedures, evaluate expressions, or assign new values to variables or properties. You can use the Immediate pane of the Debug window to accomplish these tasks. You evaluate expressions by printing their values in the Immediate pane.

Printing Information in the Immediate Pane

There are two ways to print to the Immediate pane:

- Include **Print** methods in the application code.
- Enter **Print** methods directly in the Immediate pane.

These **Print** method techniques offer several advantages over watch expressions. First, you don't have to break execution to get feedback on how the application is performing. You can see data or other messages displayed as you run the application. Another advantage is that the feedback is displayed in a separate area (the Immediate pane), so it does not interfere with output that a user sees. Finally, because you can save this code as part of the form, you don't have to redefine these statements the next time you work on the application.

Printing from Application Code

The **Print** method sends output to the Immediate pane whenever you prefix it with the special *Debug object*:

Debug.Print [*items*][;]

For example, the following statement prints the value of `Salary` to the Immediate pane every time it is executed:

```
Debug.Print "Salary = "; Salary
```

This technique works best when there is a particular place in your application code at which the variable (in this case, `Salary`) is known to change. For example, you might put the previous statement in a loop that repeatedly alters `Salary`.

Note Using **Debug.Print** statements has no effect in the executable (.EXE) version of an application. But it is a good idea to remove all **Debug.Print** statements before making an .EXE file.

For More Information For information on the Debug object, see the *Language Reference*, or search Help for *Debug object*.

Printing from Within the Immediate Pane

Once you're in break mode, move the focus to the Debug window to examine data by clicking the Debug window (if visible) or choosing Debug from the Window menu. You then can use the **Print** method without the Debug object. Type or paste a statement in the Immediate pane and then press ENTER. The Immediate pane responds by carrying out the statement, as shown in Figure 9.12.

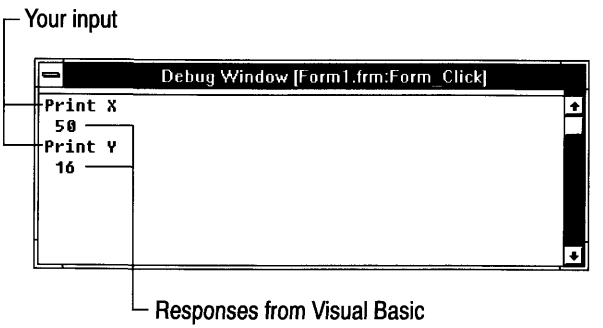


Figure 9.12 Using the Print method to print to the Immediate pane

A question mark (?) is useful shorthand for the **Print** method. The question mark means precisely the same thing as **Print** and can be used in any context in which **Print** is used. For example, the statements in Figure 9.12 could be entered as shown in Figure 9.13.

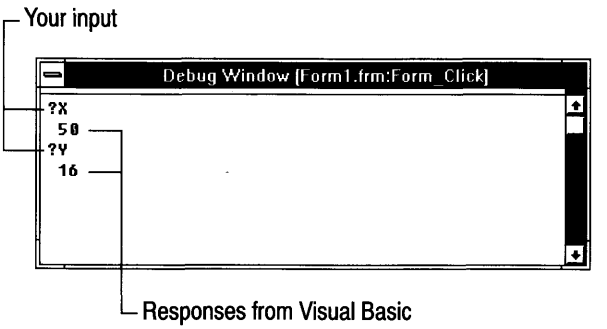


Figure 9.13 Using the Print method with a question mark to print to the Immediate pane

Printing Values of Properties

You can evaluate any valid expression in the Immediate pane of the Debug window, including expressions involving properties. The currently active form or module determines the scope. If the execution halts within code that is attached to a form, you can refer to the properties of that form (or one of its controls) and make the reference to the form implicit with statements such as the following:

```
? BackColor
? Text1.Height
```

Assuming that Text1 is a control on the currently active form, the first statement prints the numeric value of the current form's background color to the Immediate pane. The second statement prints the height of Text1.

If execution is suspended in a module or another form, you must explicitly specify the form name as follows:

```
? Form1.BackColor
? Form1.Text1.Height
```

Note Referencing an unloaded form in the Immediate pane of the Debug window loads that form.

Assigning Values to Variables and Properties

As you close in on the possible cause of an error, you may want to test the effects of particular data values. In break mode, you can set values with statements like these in the Immediate pane of the Debug window:

```
BackColor = 255
VScroll1.Value = 100
MaxRows = 50
```

The first statement alters a property of the currently active form, the second alters a property of the VScroll1 control, and the third assigns a value to a variable.

After you set the values of one or more properties and variables, you can continue execution to see the results. Or you can test the effect on procedures, as described in the next section.

Testing Procedures

The Immediate pane evaluates any valid Visual Basic executable statement, but it doesn't accept data declarations. You can enter calls to **Sub** and **Function** procedures, however, which allows you to test the possible effect of a procedure with any given set of arguments. Simply enter a statement in the Immediate pane just as you would in the Code window. For example:

```
X = Quadratic(2, 8, 8)
DisplayGraph 50, Arr1
Form_MouseDown 1, 0, 100, 100
```

Visual Basic switches to run time long enough to execute the statement, and then returns to break mode. At that point, you can see results and test any possible effects on variables or property values.

If **Option Explicit** is in effect (requiring all variable declarations to be explicit), any variables you enter in the Immediate pane must be declared within the current scope. Scope applies to procedure calls just as it does to variables. You can call any procedure within the currently active form. You can always call a procedure in a module, unless you define the procedure as **Private**, in which case you can call the procedure only while executing in the module.

Note Although all executable statements are supported in the Immediate pane, a control structure is valid only if it can be completely expressed on one line. The following **For** loop is valid in the Immediate pane:

```
For I = 1 To 20 : Print 2 * I : Next I
```

Using the Immediate Pane

Here are some shortcuts you can use in the Immediate pane:

- Once you enter a statement, you can execute it again by moving the insertion point back to that statement and pressing ENTER anywhere on the line.
- Before pressing ENTER, you can edit the current statement and alter its effects.
- You can use the mouse or the arrow keys to move around in the Immediate pane. Don't press ENTER unless you are at a statement you want to execute.
- You can also use the PAGE UP and PAGE DOWN keys to move a page at a time. CTRL+PAGE DOWN always takes you to the end of the Immediate pane, past all of the old statements.
- The HOME and END keys move to the beginning and end of the current line.

When you move the insertion point back to a **Print** statement (or statement with a question mark), remember that Visual Basic overwrites old statements in the Immediate pane as it prints each line of output. This means that unless you append a semicolon (;) to the end of the **Print** statement, it overwrites *two* statements. For example, suppose you move the insertion point to the first of the three lines below:

```
? X
55
? Y * 20
```

Pressing ENTER prints the current value of `X` on the second line and then places the insertion point at the beginning of the third line. The third line is also cleared. To avoid erasing the third line, append a semicolon to the first line:

```
? X ;
55
? Y * 20
```

For More Information For information on the Immediate pane, search Help for *Immediate*.

Special Considerations in Debugging Events

Certain events that are a normal part of using Windows can pose special problems for debugging an application. It's important to be aware of these special problems so they don't confuse or complicate the debugging process.

If you remain aware of how break mode can put events at odds with what your application expects, you can usually find solutions. You may need to use **Debug.Print** statements to monitor values of variables or properties instead of using watch expressions or breakpoints. You may also need to change the values of variables that depend on the sequence of events.

Breaking Execution During MouseDown

If you break execution during a MouseDown event procedure, you may release the mouse button or use the mouse to do any number of tasks. When you continue execution, however, the application assumes that the mouse button is still pressed down. You don't get a MouseUp event until you press the mouse button down again and then release it.

When you press the mouse button down during run time, you break execution in the MouseDown event procedure again, assuming you have a breakpoint there. In this scenario, you never get to the MouseUp event. The solution is usually to remove the breakpoint in the MouseDown procedure.

Breaking Execution During KeyDown

If you break execution during a KeyDown procedure, similar considerations apply. If you retain a breakpoint in a KeyDown procedure, you may never get a KeyUp event. (KeyDown and KeyUp are described in Chapter 17, "Interacting with the Environment.")

Breaking Execution During GotFocus or LostFocus

If you break execution during a GotFocus or LostFocus event procedure, the timing of system messages may cause inconsistent results. Use a **Debug.Print** statement instead of a breakpoint in GotFocus or LostFocus event procedures.

Testing and Using Command-Line Arguments

You can choose to have your application use command-line arguments, which provide data to your application at startup. The user can enter them by choosing the operating environment's Run command and then typing arguments after the application name. You can also use the command-line argument when creating an icon for the application in the Program Manager.

Command-line arguments can also be specified when starting an application with the **Shell** command.

For example, suppose you create an alarm clock application. One of the techniques for setting the alarm time is to let the user type in the desired time directly. The user might enter the following string in the Run command dialog box:

```
Alarm 11:00:00
```

The **Command** function returns all arguments entered after the application name (in this case, Alarm). The Alarm application has only one argument, so in the application code, you can assign this argument directly to the string that stores the desired time:

```
AlarmTime = Command
```

If **Command** returns an empty string, there are no command-line arguments. The application must either prompt for the information directly or select a default action.

To test code that uses **Command**, you can specify sample command-line arguments from within the Visual Basic environment. The application evaluates sample command-line input the same way it does if the user types the argument.

▶ **To set sample command-line arguments**

1. From the Options menu, choose Project.
2. Enter the sample arguments for the Command Line Argument. (Do not type the name of the application itself.)
3. Choose OK.

   –or–

   Press ENTER.
4. Run the application.

For More Information For information on the **Command** function, see the *Language Reference*, or search Help for *Command*.

Avoiding Bugs

There are a number of things you can do to avoid creating bugs in your applications:

- Design your applications carefully by writing down the relevant events and the way your code will respond to each one. Give each event procedure and each general procedure a specific, well-defined purpose.
- Include numerous comments. As you go back and analyze your code, you'll understand it much better if the purpose of each procedure is stated in comments.
- Develop a consistent naming scheme for the variables and objects in your application. One of the most common sources of errors is incorrectly typing a variable name or confusing one control with another.

  You can avoid this error by having Visual Basic require explicit declaration of all variables. For more information on requiring explicit variable declaration, see Chapter 7, "Variables, Constants, and Data Types."

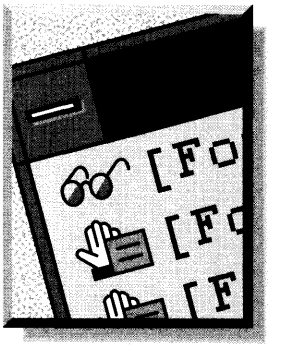
Tips for Debugging

There are a number of things you can do to simplify debugging:

- When your application doesn't produce correct results, browse through the code and try to find statements that may have caused the problem. Set breakpoints at these statements and restart the application. Breakpoints are described in the section "Entering Break Mode at a Problem Statement," earlier in this chapter.
- When the program halts, test the values of important variables and properties. Set watch expressions to monitor these values. Use the Debug window to examine variables and expressions. Watch expressions are described in the section "Monitoring Data with Watch Expressions," earlier in this chapter.
- Single step through your code, using watch expressions to monitor how values change as the code runs.
- If you determine that a variable or property is causing problems in your application, define a break expression that halts execution when the wrong value is assigned to the variable or property.

CHAPTER 10

Handling Run-Time Errors



Ideally, Visual Basic procedures wouldn't need error-handling code at all. Unfortunately, we live in a world where files are mistakenly deleted, disk drives run out of space, and network drives sometimes disconnect unexpectedly. Such eventualities can cause run-time errors in your code. To handle these errors, you need to add error-handling code to your procedures.

Contents

- Why Use Error Handling?
- How to Handle Errors
- Unanticipated Errors
- Generating Your Own Errors
- Centralized Error Handling
- Delayed Error Handling
- Turning Off Error Handling

Why Use Error Handling?

The following **Function** procedure returns **True** if the specified file exists and **False** if it does not, but it doesn't contain error-handling code:

```
Function FileExists (filename)
    FileExists = (Dir(filename) <> "")
End Function
```

The **Dir** function returns the first file matching the specified file name (given with or without wildcard characters, drive name, or path); it returns a zero-length string if no matching file is found. The code appears to cover either of the possible outcomes of the **Dir** call. However, if the drive letter specified in the argument is not a valid drive, the error `Device unavailable` occurs. If the specified drive is a floppy disk drive, this function will work correctly only if a disk is in the drive and the drive door is closed. If not, Visual Basic displays the error `Disk not ready` and halts execution of your code.

Example

To avoid this situation, you can use the error-handling features in Visual Basic to intercept errors and take corrective action. (Intercepting an error is also known as *trapping* an error.) For example, device problems such as an invalid drive or an empty floppy disk drive could be handled by the following code:

```
Function FileExists (filename)
Dim Msg
On Error GoTo CheckError    ' Turn on error trapping so error handler
                            ' responds if any error is detected.
    FileExists = (Dir(filename) <> "")
    Exit Function           ' Avoid executing error handler
                            ' if no error occurs.

CheckError:                 ' Branch here if error occurs.
    ' Define constants to represent Visual Basic error code.
    Const ERR_DISKNOTREADY = 71, ERR_DEVICEUNAVAILABLE = 68
    ' Define constants for message-box types.
    Const MB_EXCLAIM = 48, MB_STOP = 16, MB_OK_CANCEL = 1, BUTTON_OK = 1

    If (Err = ERR_DISKNOTREADY) Then
        Msg = "Put a floppy disk in the drive and close the drive door."
        ' Display message box with an exclamation mark icon and with OK
        ' and Cancel buttons.
        If MsgBox(Msg, MB_EXCLAIM + MB_OK_CANCEL) = BUTTON_OK Then
            Resume
        Else
            Resume Next
        End If
```

```
ElseIf Err = ERR_DEVICEUNAVAILABLE Then
        Msg = "This drive or path does not exist: " + filename
        MsgBox Msg, MB_EXCLAIM
        Resume Next
    Else
        Msg = "Unexpected error #" + Str(Err) + " occurred: " + Error
        ' Display message box with Stop sign icon and OK button.
        MsgBox Msg, MB_STOP
        Stop
    End If
    Resume
End Function
```

In this code, the **Err** function returns the number associated with the run-time error that occurred; the **Error** function returns the message string associated with that error.

When Visual Basic generates the error `Disk not ready`, this code displays a message telling the user to choose one of two buttons—OK or Cancel. If the user chooses OK, the **Resume** statement returns control to the statement at which the error occurred and attempts to re-execute that statement. This succeeds if the user has corrected the problem; otherwise, the program returns to the error handler.

If the user chooses Cancel, the **Resume Next** statement returns control to the statement following the one at which the error occurred (in this case, the **Exit Function** statement).

Should the error `Device unavailable` occur, this code displays a message describing the problem. The **Resume Next** statement then causes the function to continue execution at the statement following the one at which the error occurred.

If an unanticipated error occurs, an alternative message is displayed and the code halts at the **Stop** statement. See the section "Guidelines for Complex Error Handling" for an explanation of how to use the **Stop** statement.

The application you create can correct an error or prompt the user to change the conditions that caused the error. To do this, you use techniques such as those shown in the preceding example. The next section discusses these techniques in detail.

How to Handle Errors

The error-handling code in the preceding example involves three steps:

1. Set, or *enable*, an error trap by telling the application where to branch to (which error-handling routine to execute) when an error occurs.

   The **On Error** statement enables the trap and directs the application to the label `CheckError`.

2. Write an error-handling routine that responds to all errors you can anticipate. If control actually branches into the trap at some point, the trap is then said to be *active*.

   The `CheckError` routine handles the error using an **If...Then...Else** statement that responds to the value returned by the **Err** function. **Err** returns a numeric code corresponding to the error message that Visual Basic generates. In the example, if `Disk not ready` is generated, a message prompts the user to close the drive door. A different message is displayed if the `Device unavailable` error occurs. If any other error is generated, the appropriate message is displayed and the program stops.

3. Exit the error-handling routine.

   In the case of the `Disk not ready` error, the **Resume** statement makes the code branch back to the statement where the error occurred. Visual Basic then tries to re-execute that statement. If the situation has not changed, then another error occurs and execution branches back to the error-handling routine.

   In the case of the `Device unavailable` error, the **Resume Next** statement makes the code branch to the statement following the one at which the error occurred.

Details on how to perform these steps are provided in the following sections. Refer to the FileExists **Function** example in the preceding section as you read.

Setting the Error Trap

An error trap is enabled when Visual Basic executes the **On Error** statement, which specifies an error handler. The error trap remains enabled while the procedure containing it is active—that is, until an **Exit Sub**, **Exit Function**, **End Sub**, or **End Function** statement is executed for that procedure. While only one error trap can be enabled at any one time in any given procedure, you can create several alternative error traps and enable different ones at different times. You can also disable an error trap by using a special case of the **On Error** statement—**On Error GoTo 0**. For more information about disabling error handling, see the section "Turning Off Error Handling," later in this chapter.

Writing an Error-Handling Routine

The first step in writing an error-handling routine is placing the **On Error GoTo** *line* statement, where *line* indicates the label identifying the error-handling code. In the FileExists function, the label is `CheckError`. (Although the colon is part of the label, it isn't used in the **On Error GoTo** *line* statement.) A common convention is to place the error-handling code at the end of the procedure with an **Exit Sub** or **Exit Function** statement immediately before the line label. This allows the procedure to avoid executing the error-handling code if no error occurs.

The **Err** function returns a numeric code representing the most recent run-time error. (As mentioned earlier, a run-time error can occur only when your code is running.) By using **Err** in combination with the **Select Case** or **If...Then...Else** statement, you can take specific action for any error that occurs.

Note The string returned by **Error** explains the error associated with the current error number. The exact wording of the message may vary among different versions of Microsoft Visual Basic.

Exiting an Error-Handling Routine

The FileExists **Function** example used the **Resume** statement within the error handler to re-execute the statement that originally caused the error, and used the **Resume Next** statement to return execution to the statement following the one at which the error occurred. There are other ways to exit an error-handling routine. Depending on the circumstances, you can exit using any of the statements shown in the following table.

Statement	Description
Resume (0)	Program execution resumes with the statement that caused the error or the most recently executed call out of the procedure containing the error-handling routine.
Resume Next	Resumes program execution at the statement immediately following the one that caused the error or with the statement immediately following the most recently executed call out of the procedure containing the error-handling routine.
Resume *line*	Resumes program execution at the label specified by *line*, where *line* is a nonzero line number or line label that must be in the same procedure as the error handler.
Error Err	Triggers a run-time error. When this statement is executed within the error-handling routine, Visual Basic searches along the calls list for another error-handling routine. (The *calls list* is the chain of procedures invoked to arrive at the current point of execution. See the section "Unanticipated Errors," later in this chapter.)

Resume returns to the statement that caused the error. Use it to repeat an operation after correcting the error.

The **Resume Next** statement returns to the statement immediately following the one that caused the error. The difference between **Resume** and **Resume Next** is shown in Figure 10.1.

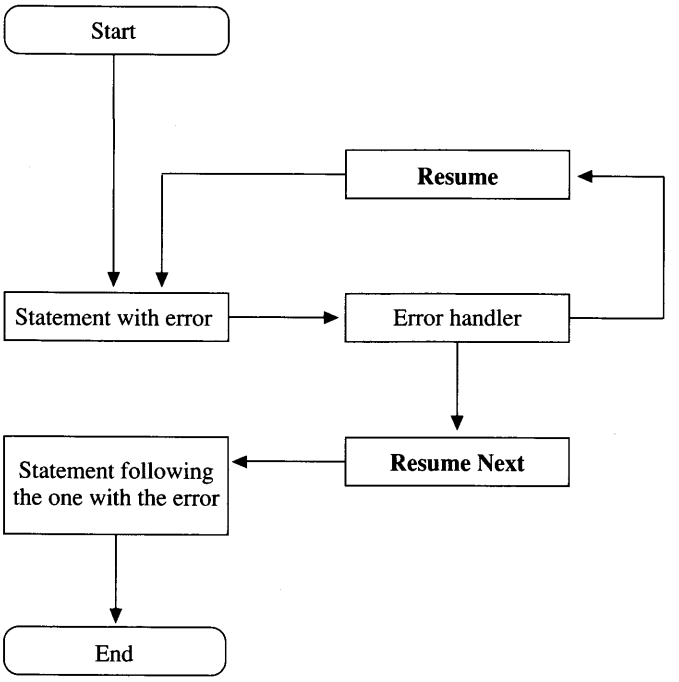


Figure 10.1 Program flow with Resume and Resume Next

Generally, you would use **Resume** whenever the user must make a correction and **Resume Next** whenever the code needs correction. However, your error handler may be written so that the existence of a run-time error is never revealed to the user.

Example

For example, the following **Function** procedure uses error handling to perform "safe" division on its arguments without revealing errors that might occur. The following table lists the errors that can occur when performing division.

Error	Cause
`Division by zero`	Numerator is nonzero, but the denominator is zero.
`Overflow`	Both numerator and denominator are zero (during floating-point division).
`Illegal function call`	Either the numerator or the denominator is a nonnumeric value (or can't be considered a numeric value).

In all three cases, the following **Function** procedure traps these errors and returns **Null** instead:

```
Function Divide (numer, denom)
Const MB_EXCLAIM = 48
Const ERR_DIV0 = 11, ERR_OVERFLOW = 6, ERR_ILLFUNC = 5
On Error GoTo MathHandler
    Divide  = numer / denom
    Exit Function
MathHandler:
    If Err = ERR_DIV0 Or Err = ERR_OVERFLOW Or Err = ERR_ILLFUNC Then
        Divide = Null       ' If error was Division by zero, Overflow,
    Else                    ' or Illegal function call, return Null.
        MsgBox "Unanticipated error " & Err & ": " & Error, MB_EXCLAIM
    End If                  ' In all cases, Resume Next continues
    Resume Next             ' execution at the Exit Function statement.
End Function
```

Resume Next can also be used if you anticipate an error in a loop, and you need to restart the operation. Or, you can use **Resume** *line*, which returns control to a specified line label.

The following example illustrates the use of the **Resume** *line* statement. A variation on the FileExists example shown earlier, this function allows the user to enter a file specification that the function returns if the file exists:

```
Function VerifyFile ()
Const ERR_BADFILENAME = 52, ERR_DRIVEDOOROPEN = 71
Const ERR_DEVICEUNAVAILABLE = 68, ERR_INVALIDFILENAME = 64
Dim NL, Prompt, Msg, FileSpec
    NL = Chr(13) + Chr(10)      ' Carriage return-linefeed combination.
    Prompt = "Enter file specification to check:"
StartHere:
    FileSpec = "*.*"            ' Start with a default specification.
    Msg = Msg + NL + Prompt
    'Let the user modify the default.
    FileSpec = InputBox(Msg, "File Search", FileSpec, 100, 100)
    If FileSpec = "" Then Exit Function      ' Exit if nothing entered.
    On Error GoTo Handler
    VerifyFile = Dir(FileSpec)
    Exit Function
```

```
Handler:
    Select Case Err              ' Analyze error code and load message.
        Case ERR_INVALIDFILENAME, ERR_BADFILENAME
            Msg = "Your file specification was invalid; try another."
        Case ERR_DRIVEDOOROPEN
            Msg = "Close the disk drive door and try again."
        Case ERR_DEVICEUNAVAILABLE
            Msg = "The drive you specified was not found. Try again."
        Case Else
            Error Err
    End Select
    Resume StartHere             ' This jumps back to label so user can
End Function                     ' try another file name.
```

If a file matching the specification is found, the function returns the file name. If no matching file is found, the function returns a zero-length string. If one of the anticipated errors occurs, a message is assigned to `Msg` and execution jumps back to the label `StartHere`. This gives the user another chance to enter a valid path and file specification.

Note Although using **Resume** *line* is a legitimate way to perform this operation, jumps to labels are often considered throwbacks to a less structured style of programming. Their proliferation can make code difficult to understand and debug. Moreover, because line labels must be unique within a module, you have to use different labels in every procedure within a given module.

Unanticipated Errors

As previously noted, an *enabled* error handler is one that was activated by executing an **On Error** statement and hasn't yet been turned off—either by an **On Error GoTo 0** statement or by exiting the procedure where it was enabled. An *active* error handler is one in which execution is currently taking place. To be active, an error handler must first be enabled, but not all enabled error handlers are active.

When an error occurs within a procedure lacking an enabled error-handling routine, or within an active error-handling routine, Visual Basic searches the calls list for another enabled error-handling routine. The calls list is the sequence of calls that leads to the currently executing procedure; it is displayed in the Calls dialog.

Suppose the following sequence of calls occurs, as shown in Figure 10.2:

1. An event procedure calls Procedure A.
2. Procedure A calls Procedure B.
3. Procedure B calls Procedure C.

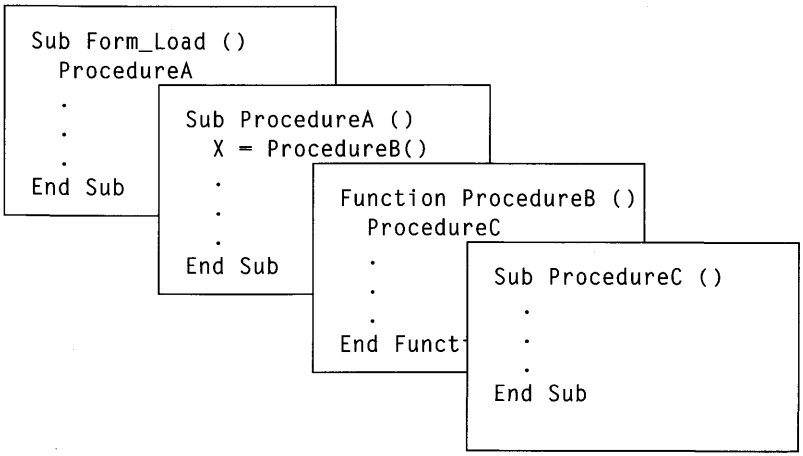


Figure 10.2 A sequence of calls

While Procedure C is executing, the other procedures are pending, as shown in the calls list in the Calls dialog. Figure 10.3 shows the calls list.

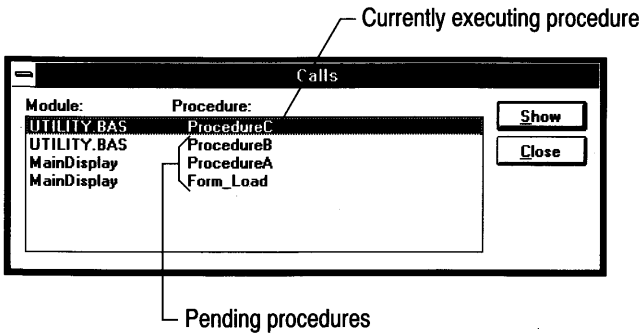


Figure 10.3 The calls list when procedures are pending

If an error occurs in Procedure C and this procedure doesn't have an enabled error handler, Visual Basic searches backward through the pending procedures in the calls list—first Procedure B, then Procedure A, then the initial event procedure (but no farther)—and executes the first enabled error handler it finds. If it doesn't encounter an enabled error handler anywhere in the calls list, it displays the appropriate message and halts execution.

If Visual Basic finds an enabled error-handling routine, execution continues in that error-handling routine as if the error had occurred in the procedure that contains the error-handling routine. If a **Resume** or a **Resume Next** statement is executed in the error-handling routine, execution continues as shown in the following table.

Statement	Result
Resume	The call to the procedure that Visual Basic just searched is re-executed. In the calls list given earlier, if Procedure A has an enabled error handler that includes a **Resume** statement, Visual Basic re-executes the call to Procedure B.
Resume Next	Execution returns to the statement following the last statement executed in that procedure. This is the statement after the call to the procedure that Visual Basic just searched back through. In the calls list given earlier, if Procedure A has an enabled error handler that includes a **Resume Next** statement, execution returns to the statement after the call to Procedure B.

Notice that the statement executed is in the procedure *where the error-handling procedure is found*, not necessarily in the procedure where the error occurred. If you don't take this into account, your code may perform in ways you don't intend.

If the range of errors doesn't include the error that actually occurred, an unanticipated error can occur within a procedure having an enabled error handler. In such a case, the procedure could execute endlessly, especially if the handler executes a **Resume** statement. To prevent such situations, use the **Error** statement in a **Case Else** clause. This actually generates an error within the error handler, forcing Visual Basic to search through the calls list for a handler that can deal with the error.

In the preceding example, the VerifyFile **Function** procedure, the number originally returned by **Err** was given as an argument to the **Error** statement in a **Case Else** clause, thereby generating an error. When such an error occurs within an active error handler, the search back through the calls list begins.

Note The effect of the search back through the calls list is hard to predict, because it depends on whether **Resume** or **Resume Next** is executed in the handler that processes the error successfully. **Resume** returns control to the most recently executed call out of the procedure containing the error handler. **Resume Next** returns control to whatever statement immediately follows the most recently executed call out of the procedure containing the error handler. For example, in the Calls list shown in Figure 10.3, if Procedure A has an enabled error handler and Procedure B and C don't, an error occurring in Procedure C will be handled by Procedure A's error handler. If that error handler uses a **Resume** statement, upon exit, the program continues with a call to Procedure B. However, if Procedure A's error handler uses a **Resume Next** statement, upon exit, the program will continue with whatever statement in Procedure A follows the call to Procedure B. In neither case does the error handler return directly to either the procedure or the statement where the error originally occurred.

Guidelines for Complex Error Handling

When you write large Visual Basic applications that use multiple modules, the error-handling code can get quite complex. Keep these guidelines in mind:

- While you are debugging your code, put an **Error Err** statement in all error-handling routines for cases where no code in the error-handling routine deals with the specific error. This allows your application to try to correct the error in other error-handling routines along the calls list. It also ensures that Visual Basic will display an error message if an error occurs that your code doesn't handle. When you test your code, this technique helps you uncover the errors you aren't handling adequately. However, in a stand-alone .EXE file, you should be cautious: If you execute **Error Err** and no other procedure traps the error, your application will terminate execution immediately, without any QueryUnload or Unload events occurring.
- If you don't want another procedure in the calls list to trap the error, use the **Stop** statement to force your code to terminate. Using **Stop** allows you to examine the context of the error while refining your code in the development environment. However, remember to remove the **Stop** statements before you create an .EXE file. If a stand-alone Visual Basic application (.EXE) encounters a **Stop** statement, it treats it as an **End** statement and terminates execution immediately, without any QueryUnload or Unload events occurring.
- Write a fail-safe error-handling procedure that all your error handlers can call as a last resort for errors they cannot handle themselves. This fail-safe procedure can perform an orderly termination of your application by unloading forms and saving data.

Generating Your Own Errors

You can generate an error in your code with the **Error** statement:

Error *errorcode*

You've already seen this statement used to regenerate the current error in an error handler:

```
Error Err
```

However, you can simulate any Visual Basic run-time error by supplying the error code for that error in the **Error** statement:

```
Error 71    ' Simulate "Disk Not Ready" error.
```

This is useful when you are testing your applications, or when you want to treat a particular condition as being equivalent to a run-time error. For example, you might be writing a module that calls routines in a dynamic link library (DLL) and want error values returned from the DLL routines to be handled as actual Visual Basic errors by the rest of your application.

You can also use the **Error** statement to generate your own, user-defined errors by supplying an error code that does not correspond to a Visual Basic run-time error. Of course, this is useful only if you also write code that handles the new errors that you define.

Note Microsoft Visual Basic doesn't use all available numbers for its own errors. As new errors are defined in future versions of Microsoft Visual Basic, the internal error numbers increase. If you want to generate and trap your own errors, begin your numbering scheme with 32,767 and work your way down. This will ensure that your error numbers will not conflict with future versions of Visual Basic or other Microsoft Basic products.

Custom controls may also define error numbers in this range. To avoid conflicts with these, consult the documentation for any custom controls you may be using in your application.

Centralized Error Handling

When you add error-handling code to your applications, you'll quickly discover that you're handling the same errors over and over. With careful planning, you can reduce code size and save effort by writing a few procedures that your error-handling code can call to handle common error situations.

Example

The FileErrors **Function** procedure that follows displays a message appropriate to the error that occurred and, where possible, allows the user to choose a button to specify what action the program should take next. It then returns a code to the procedure that called it. The value of the code indicates which action in a group of actions the program should take. Note that constants such as ERR_DEVICEUNAVAILABLE must be defined somewhere (globally, or at the module level of the module containing the procedure, or within the procedure itself). The constant MB_ICONEXCLAMATION is defined in the file CONSTANT.TXT.

```
Function FileErrors (errVal As Integer) As Integer
Dim msgType As Integer, Msg As String, response As Integer
' Return Value      Meaning          Return Value     Meaning
' 0                 Resume               2            Unrecoverable error
' 1                 Resume Next          3            Unrecognized error
    msgType = MB_ICONEXCLAMATION        ' Defined in CONSTANT.TXT.
```

```
    Select Case errVal
        Case ERR_DEVICEUNAVAILABLE                          ' Error #68
            Msg = "That device appears unavailable."
            msgType = MB_ICONEXCLAMATION + 4
        Case ERR_DISKNOTREADY                               ' Error #71
            Msg = "Insert a disk in the drive and close the door."
        Case ERR_DEVICEIO                                   ' Error #57
            Msg = "Internal disk error."
            msgType = MB_ICONEXCLAMATION + 4
        Case ERR_DISKFULL                                   ' Error #61
            Msg = "Disk is full. Continue?"
            msgType = MB_ICONEXCLAMATION + 3
        Case ERR_BADFILENAME, ERR_BADFILENAMEORNUMBER  ' Errors #64 & 52
            Msg = "That file name is illegal."
        Case ERR_PATHDOESNOTEXIST                           ' Error #76
            Msg = "That path doesn't exist."
        Case ERR_BADFILEMODE                                ' Error #54
            Msg = "Can't open your file for that type of access."
        Case ERR_FILEALREADYOPEN                            ' Error #55
            Msg = "This file is already open."
        Case ERR_INPUTPASTENDOFFILE                         ' Error #62
            Msg = "This file has a nonstandard end-of-file marker, "
            Msg = Msg + "or an attempt was made to read beyond "
            Msg = Msg + "the end-of-file marker."
        Case Else
            FileErrors = 3
            Exit Function
    End Select
    response = MsgBox (Msg, msgType, "Disk Error")
    Select Case response
        Case 1, 4       ' OK, Retry buttons.
            FileErrors = 0
        Case 5          ' Ignore button.
            FileErrors = 1
        Case 2, 3       ' Cancel, Abort buttons.
            FileErrors = 2
        Case Else
            FileErrors = 3
    End Select
End Function
```

This **Function** procedure handles common file- and disk-related errors. If the error is not related to disk I/O, it returns the value 3. The procedure that calls this **Function** procedure should then either handle the error itself or call another procedure to handle it.

Note As you write larger and larger applications, you'll find that you are using the same constants in several procedures in various forms and modules. Making those constants global and declaring them in a single module may better organize your code and save you from typing the same declarations over and over.

You can simplify error handling by calling the FileErrors procedure wherever you have a procedure that reads or writes to disk. For example, you've probably used applications that warn you if you attempt to overwrite an existing disk file. Conversely, when you try to open a nonexistent file, many applications warn you that the file does not exist and ask if you want to create it. In both instances, errors can occur when the application passes the file name to the operating system.

Example

The following checking routine uses the value returned by the FileErrors procedure to decide what action to take in the event of a disk-related error.

```
Function ConfirmFile (FName As String, Operation As Integer) As Integer
' Parameters:
' FName     File spec to be checked for and confirmed.
' Operation Code for sequential file access mode (Output, Input, etc.)
'       Note procedure works for binary and random because messages are
'       conditioned on Operation being <> to certain sequential modes.
' Returns:
' 1     Confirms operation will not cause a problem.
' 0     User decided not to go through with operation.

Const SAVEFILE = 1, LOADFILE = 2
Const REPLACEFILE = 1, READFILE = 2
Const ADDTOFILE = 3, RANDOMFILE = 4, BINARYFILE = 5
Dim Confirmation As Integer, Action As Integer
Dim NL As String, Msg As Variant
    NL = Chr(13) + Chr(10)                      ' Carriage return-linefeed.
    On Error GoTo ConfirmFileError              ' Turn on the error trap.
    FName = Dir(FName)                          ' See if the file exists.
    On Error GoTo 0                             ' Turn error trap off.
    ' If user is saving text to a file that already exists...
    If FName <> "" And Operation = REPLACEFILE Then
        Msg = "The file " + FName + "already exists on disk." + NL
        Msg = Msg + "Saving the text box contents to that file " + NL
        Msg = Msg + "will destroy the file's current contents, " + NL
        Msg = Msg + "replacing them with the text from the text box."
        Msg = Msg + NL + NL
        Msg = Msg + "Choose OK to replace file, Cancel to stop." + NL
        Confirmation = MsgBox(Msg, 65, "File Message")
    ' If user wants to load text from nonexistent file...
    ElseIf FName = "" And Operation = READFILE Then
        Msg = "The file " + FName + " doesn't exist." + NL
        Msg = Msg + "Would you like to create and then edit it?" + NL
        Msg = Msg + NL + "Choose OK to create the file, Cancel to stop."
```

```
        Confirmation = MsgBox(Msg, 65, "File Message")
    ' If FName doesn't exist, force procedure to return 0 by setting
    ' Confirmation = 2.
    ElseIf FName = "" Then
        If Operation = RANDOMFILE Or Operation = BINARYFILE Then
            Confirmation = 2
        End If
    ' If the file exists & operation isn't successful,
    ' Confirmation = 0 & procedure returns 1.
    End If
    ' If no box was displayed, Confirmation = 0; if user chose OK
    ' in either case, Confirmation = 1 and ConfirmFile should return 1,
    ' to confirm that the intended operation is OK. If Confirmation > 1,
    ' ConfirmFile should return 0, because user doesn't want to go
    ' through with the operation...
    If Confirmation > 1 Then
        ConfirmFile = 0
    Else
        ConfirmFile = 1
        If Confirmation = 1 Then
            If Operation = LOADFILE Then      'User wants to create file.
                Operation = REPLACEFILE       'Assign REPLACEFILE so
            End If  ' caller will understand action to be taken.
        End If      ' Return code confirming desire to either overwrite
                    ' existing file or create new file.
    End If
Exit Function
ConfirmFileError:
Action = FileErrors(Err)
    Select Case Action
        Case 0
            Resume
        Case 1
            Resume Next
        Case 2
            Exit Function
        Case Else
            Error Err
    End Select
End Function
```

The ConfirmFile **Function** procedure receives a specification for the file whose existence will be confirmed, plus information about which access mode will be used when an attempt is made to actually open the file. If a sequential file is to be saved (REPLACEFILE), and a file is found that already has that name (and will therefore be overwritten), the user is prompted to confirm that overwriting the file is acceptable.

If a sequential file is to be opened (READFILE) and the file is not found, the user is prompted to confirm the desire to create the file. If the file is being opened for random or binary access, its existence or nonexistence is either confirmed (return value 1) or refuted (return value 0). If an error occurs in the call to **Dir**, the FileErrors **Function** procedure is called to analyze the error and prompt the user for a reasonable course of action. Like FileErrors, ConfirmFile is used in the applications presented in Chapter 19, "Processing Files."

Delayed Error Handling

You may be used to programming in a language that doesn't interrupt your code when errors occur but instead records errors for you to check later. The C programming language works this way, and you may sometimes find it convenient to follow this practice in your Visual Basic code. You do this by including the **On Error Resume Next** statement to tell Visual Basic to set the value of **Err** when an error occurs, but without interrupting the application.

Follow the **On Error Resume Next** statement with error-handling code; the code can check the value returned by the **Err** function to tell if an error occurred. If **Err** doesn't return zero, an error has occurred, and then the error-handling code can take the appropriate action based on the value of **Err**:

```
On Error Resume Next    ' Don't let an error disrupt the application.
Condition = False
    Do Until Condition = True
    ' A long calculation loop that exits when Condition is True.
        .
        .
        .
    Loop
    Select Case Err         ' Now see the last error that occurred.
        Case 0              ' No error: Continue with application.
            .
            .
            .
        Case Err_Overflow           ' Deal with overflow error.
            .
            .
            .
        Case Err_DivisionByZero     ' Deal with division by zero.
            .
            .
            .
        Case Else                   ' Deal with unexpected errors.
            Error Err
    End Select
    ' Program continues here after checking for errors.
```

Keep these things in mind when using delayed error handling:

- The code that detects and deals with an error differs from the error-handling routines discussed earlier in this chapter—it does not contain any labels or **Resume** statements.
- The error number returned by **Err** is the number of the most recent error. Errors that occur earlier in the preceding code are lost.
- If a **Resume**, **Resume Next**, or **On Error** statement is executed, or if a procedure call is made, Visual Basic resets **Err** to 0. If you're going to use the value returned by **Err**, store it in a variable before executing any of these statements. This is especially important if your code calls procedures that may themselves have errors and error-handling code.
- Once you have handled an error, set **Err** to 0. This ensures that you can detect and handle subsequent errors correctly.
- Finally, if errors occur and handling is deferred, logic errors in the intervening code could complicate things significantly.

Sometimes you use **On Error Resume Next** because you want your code to retry a statement until it executes without causing an error:

```
On Error Resume Next
Do
    ' Some statement that causes an error.
Loop Until Err = 0
```

However, this causes an infinite loop, because **Err** is set when an error occurs but does not get reset when a statement executes without causing an error. The solution is to explicitly reset **Err** in the loop:

```
On Error Resume Next
Do
    ' Some statement that causes an error.
    If Err = 0 Then Exit Do
    Err = 0
Loop
```

Another reason for using **On Error Resume Next** is to tailor error handling to each group of related statements rather than to have a single error-handling routine for each procedure.

Turning Off Error Handling

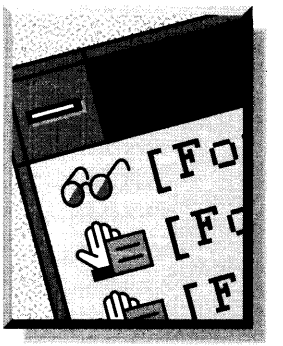
If an error trap has been enabled in a procedure, it is automatically disabled when the procedure finishes executing. However, you may want to turn off an error trap in a procedure while the code in that procedure is still executing. To turn off an enabled error trap, use the **On Error GoTo 0** statement. Once Visual Basic executes this statement, errors are detected but not trapped within the procedure. You can use **On Error GoTo 0** to turn off error handling anywhere in a procedure—even within an error-handling routine itself.

For example, try single stepping through a procedure such as this:

```
Sub ErrDemoSub ()
On Error GoTo SubHandler
' Error trapping is enabled.
' Errors need to be caught and corrected here.
    Kill "OLDFILE.XYZ"
On Error GoTo 0             ' Error trapping is turned off here.
    Kill "OLDFILE.XYZ"
On Error GoTo SubHandler    ' Error trapping is enabled again.
    Kill "OLDFILE.XYZ"
Exit Sub
SubHandler:                 ' Error-handling routine goes here.
    MsgBox "Caught error."
    Resume Next
End Sub
```

CHAPTER 11

Optimizing Your Application for Size and Speed



As you create larger and more sophisticated applications, the amount of memory they consume and the speed with which they execute become more significant. You may decide you need to *optimize* your application by making it smaller and by speeding calculations and displays.

Optimizing does not always benefit you in the long run. Sometimes the changes you make to speed up or trim down your application result in code that is more difficult to maintain, debug, or expand in the future. You may also have to balance the relative benefits of speed and compactness: Many of the techniques that improve speed do so at the expense of memory; conversely, many of the techniques that reduce the size of your application also cause it to run more slowly.

When you do need to make your application smaller or faster, however, you can make use of the optimization techniques presented in this chapter.

Contents

- Speeding Calculations and Code
- Speeding Display
- Speeding the Start of Your Application
- Minimizing Total Application Size in Memory
- Minimizing Memory Required by Graphics
- Minimizing Stack Use

Speeding Calculations and Code

In general, you can do more to improve the speed of your code by choosing more efficient algorithms than by implementing particular coding "tricks." Nevertheless, some coding techniques do increase the speed of your code, particularly within loops.

Use Integer Variables for Calculations

You can perform mathematical computations that do not involve fractional values faster if you use **Integer** or **Long** variables instead of **Variant** variables. **Integer** or **Long** variables are also the best choice for the iterator variable in **For...Next** loops. Likewise, the integer division operator (\) is much faster than the floating-point division operator (/). Increased speed comes at the expense of flexibility, though, when you encounter cases of overflow that **Variant** variables handle automatically.

Cache Frequently Used Properties in Variables

You can get and set the value of variables faster than those of properties. If you are getting the value of a property many times (such as in a loop), your code runs faster if you assign the property to a variable outside the loop and then use the variable instead of the property. You cannot do this if the property changes while your code is running.

Use Specific Object Types

When you use specific object types (such as TextBox instead of Control, or frmMain instead of Form) for variables and arguments, Visual Basic can resolve references to the properties and methods of those objects before you run the application. If you use generic object types (such as Form, Control, or MDIForm), Visual Basic must resolve references to their properties and methods every time it encounters them at run time—a significantly slower process.

For more information on object types, see Chapter 8, "Objects and Instances."

Use the Value of the Control

Visual Basic designates one property for each type of control as the *value* of that control. If you assign a setting to a control name without specifying a property, Visual Basic assigns it to this property. If you use the control in an expression without specifying a property, Visual Basic uses this property. When you want to access the property designated as the value of a control, you can do so more quickly if you do not specify it explicitly.

For more information on the value of a control, see Chapter 6, "Programming Fundamentals."

Replace Procedure Calls with In-Line Code

Performing each procedure call always involves some additional work and time. If you have a loop that calls a procedure many times, you can eliminate this overhead by removing the procedure call and placing the body of the procedure directly *in-line* within the loop. If you place the same code in-line in several loops, however, the duplicate code increases the size of your application.

Speeding Display

For an application running in a graphical environment, the speed of graphics and other display operations is crucial to the *perception* of the speed of the application. The faster forms appear and paint, the faster your application will seem to the user. You can make use of a variety of techniques in your application to make these display operations faster.

Use the Image Control to Display Bitmaps

This optimization both improves the speed and minimizes the size of your application, so take advantage of it whenever possible. When you are simply displaying pictures and reacting to clicks and mouse actions on them, for example, use the image control instead of the picture box. Don't use a picture box unless you need the capabilities only the picture box provides (such as graphics methods, containing other controls, or DDE). When you are converting applications from Visual Basic 1.0, consider replacing picture box controls with image controls.

Set the ClipControls Property of Containers to False

You can reduce the time needed to display complex forms by setting the ClipControls property of the form, and the frames and pictures on the form, to **False**. This may cause unpredictable results, however, if your code includes graphics methods.

For more information on using the ClipControls property with graphics methods, see Chapter 15, "Creating Graphics for Applications."

Use Graphic Controls and Methods Appropriately

Use graphic controls to create simple graphical elements that rarely change. Complex graphics, or graphics that change rapidly, are generally best handled with graphics methods.

Use the AutoRedraw Property Appropriately

When AutoRedraw is set to **True** for a form or control, Visual Basic maintains a bitmap that it uses to repaint that form or control. While this improves the speed of simple repaints (for example, when the form or control is revealed after a window that covers it is removed), it slows graphics methods because Visual Basic has to perform the graphics methods on the AutoRedraw bitmap and then copy the entire bitmap to the screen. This process also consumes a considerable amount of memory.

If your application generates complex graphics but doesn't change them frequently, setting AutoRedraw to **True** is appropriate. But if your application draws graphics that must change frequently, you will get better performance if you set AutoRedraw to **False** and perform the graphics methods for the form or control in the Paint event.

Note When AutoRedraw is set to **True,** Visual Basic consumes a significant amount of memory for the AutoRedraw bitmap. If you aren't performing graphics methods, you should set AutoRedraw to **False**.

Use the Show Method in the Load Event for Forms

Visual Basic does not make a form visible until it has executed all of the code in the Load event for the form. You can change this behavior by using the **Show** method as the first line in the Load event procedure for that form. This gives the user something to look at while the rest of the code in the Load event executes. Avoid calling procedures in the Load event if those procedures are in another module that might not yet be loaded.

Use the Clear Method to Empty List and Combo Boxes

If you are updating code you wrote with Visual Basic 1.0, you may have code that empties a list box or combo box by performing a loop and removing each item separately. You can replace this code with the **Clear** method, which is much faster.

Keep Forms in Memory

Forms display more quickly if Visual Basic doesn't have to load them before showing them. If you hide a form instead of unloading it, you can redisplay it more quickly; however, each loaded form—whether visible or not—consumes significant amounts of system resources. Moreover, although preloading forms as your application starts improves the speed with which those forms are displayed, it also slows the startup of your application. You may find that forms load acceptably fast and that minimizing their use of memory is more beneficial.

Speeding the Start of Your Application

First impressions are important. The first impression your application makes depends, in part, on how long it takes to display the first form. There are several techniques you can use to reduce the time it takes to make that first form visible.

Simplify Your Startup Form

Minimize the size of bitmaps and the amount of code. Load other forms from the startup form. Because the amount of time required to load a form is directly related to the size and complexity of the form and its code, making your startup form simple ensures that it is displayed as soon as possible after your application starts.

Typically, the first form is a simple About dialog box, an introductory screen, or a logon dialog box.

Avoid Time-Consuming Code in the Startup Form Load Event

Calling a procedure in another module causes Visual Basic to load that module, and that takes time. Instead of calling procedures, the code in the Load event for the startup form should use the **Show** method to display itself and the next form, and then unload itself.

Make Sure That the Run-Time DLL Is Loaded

Finished .EXE files created in Visual Basic use the VBRUN300.DLL run-time file. If this run-time file isn't already loaded when an .EXE starts, the .EXE must load it into memory. However, if you create a small .EXE file (a useful utility such as a clock or calendar) in Visual Basic and keep it running at all times, your other Visual Basic applications do not have to load VBRUN300.DLL. This reduces the time required for them to start.

Minimizing Total Application Size in Memory

In most cases, you will want to make your application as compact as possible. Smaller applications load faster, and because they consume less memory, you can run additional applications at the same time. Visual Basic also places some limits on the size of certain elements. (For more information on size restrictions, see Appendix D, "Specifications and Limitations.")

Of course, some elements—such as long identifiers, comments, and blank lines—are eliminated when you create an .EXE file, so there is no reason to restrict the length of identifiers or the size or number of comments in your code. None of these affects the size of your application in memory when it is running as an .EXE file.

Other elements in your application, however—including variables, forms, and procedures—*do* take up space in memory. It is usually in your best interests to streamline these. There are several techniques you can use to reduce the memory your application occupies when it is running as an .EXE file. These techniques are described in the following sections.

Reduce the Number of Loaded Forms

Each loaded form, whether visible or not, consumes a significant amount of memory (which varies with the number and types of controls on the form, the size of bitmaps on the form, and so on). Load forms only when you need to display them, and unload them (rather than hide them) when you no longer need them. Remember that any reference to the properties, methods, or controls of a form, or a form variable declared with **New**, causes Visual Basic to load the form.

Keep Data in Separate Data Files and Load Data Only When Needed

Data you place directly in your application at design time (as properties or as literal strings and numbers in your code) increase the memory the application consumes at run time. You can reduce memory here by loading the data from disk files at run time. This is particularly valuable for large bitmaps and strings.

Organize Your Code Modules

Visual Basic loads modules on demand—that is, it loads a module into memory only when your code calls one of the procedures in that module. If you never call a procedure in a particular module, Visual Basic never loads that module. By placing related procedures in the same module, you enable Visual Basic to load modules only as you need them.

Consider Alternatives to Variant Data Types

The **Variant** data type is extremely flexible, but it is also larger than any of the other data types. When you must squeeze every last byte out of your application, consider replacing **Variant** variables with other data types. In some cases, however, using other data types forces you to add more code to compensate for the loss of flexibility that the **Variant** provided, resulting in no net improvement in size.

Minimize Forms with LinkMode Set to Source at Design Time

Whenever a form acts as a DDE source, Visual Basic must load a table containing the names of every control on the form. The more forms that act as DDE sources, the more memory these tables consume. To minimize this use of memory, limit the number of loaded forms that act as DDE sources.

For more information about forms used as DDE sources, see Chapter 21, "Communicating with Other Applications."

Use Dynamic Arrays and Erase to Reclaim Memory

Consider using dynamic arrays instead of fixed arrays. When you no longer need the data in a dynamic array, use **Erase** or **ReDim Preserve** to discard unneeded data and reclaim the memory used by the array. You cannot reclaim the memory used by fixed arrays.

Reclaim Unused Variables

If you no longer need the contents of a module-level or global string variable or a **Variant** containing a string, assign a zero-length string to that variable.

If you no longer need an object variable, set that variable to **Nothing** to reclaim the memory used by the object reference.

Review Your Code for Unused Variables and Constants

As you develop and modify your application, you may leave behind variables and constants you declared but no longer use, especially if you have modules full of constants that you routinely add to all your applications. Visual Basic does not eliminate unused variables or constants when you create an .EXE file, so you must review your code to find and remove them.

Use the Find command on the Edit menu to search for references to a particular variable or constant. Or, if you have **Option Explicit** statements in each of your modules, you can quickly discover if a variable or constant is used in your application by removing its declaration and running the application; Visual Basic generates an error at the first point in the application where the variable or constant is used. If you do not see an error, the constant or variable was not used, and the declaration can be removed.

Eliminate Dead Code

As you develop and modify your applications, you may leave behind *dead code*—entire procedures that you do not call from anywhere in your code. Or you may routinely include modules containing utility procedures in your projects, but you actually call only some of those procedures in your code. Visual Basic does not remove dead code when you create an .EXE, so you must review your code to find and remove unused procedures.

Minimizing Memory Required by Graphics

Graphics (pictures and graphics methods) can consume a lot of memory. To some extent, this is unavoidable: Graphics contain a lot of information, so they tend to be large. But in many cases, you can reduce the impact that graphics have on the size of your application by applying some of the following techniques.

Use the Image Control to Display Bitmaps

This optimization both improves the speed and reduces the size of your application, so try to take advantage of it whenever possible. When you are converting applications from Visual Basic 1.0, consider replacing picture box controls with image controls.

Set AutoRedraw to False

When AutoRedraw is **True** for a form or control, Visual Basic maintains a bitmap that it uses to repaint that form or control. This bitmap can consume a significant amount of memory, so if you aren't performing graphics methods on that form or control, you should set its AutoRedraw property to **False**. If the AutoRedraw property has been set to **True**, always use the **Cls** method to reclaim the memory used by the bitmap before setting AutoRedraw to **False**.

Use RLE Format Bitmaps

Run-Length Encoding (RLE) is a standard form of bitmap compression used by many graphics tools in Microsoft Windows. RLE bitmaps consume significantly less memory than ordinary bitmaps and are not noticeably slower to load or display.

Load Bitmaps from Files As Needed

When you set a Picture property at design time, you add the picture to the form and thereby increase the memory the form consumes at run time. You can reduce this by using the **LoadPicture** function to load pictures from disk files at run time.

Alternatively, if you don't want to keep separate picture files, you can store pictures in forms. To do this, place bitmaps in image controls on several different storage forms at design time. At run time, load the appropriate form and assign the Picture property of the image in the storage form to the Picture property of the form or control where you want the picture to appear. This increases the total size of your application (because there is some overhead associated with each additional form); however, the amount of memory your application uses at any given time can be quite small, since it depends on the forms that are loaded at that time.

Reclaim Memory from Unused Pictures

When you are no longer using a picture in the Picture property of a form, picture box, or image control, use the **LoadPicture** function with no argument to empty the Picture property.

Free the Memory Used by the Image Property

If you use the Image property of a picture box or form, Visual Basic creates an AutoRedraw bitmap (even if the AutoRedraw property for that form or picture box is **False**). When you have finished using the Image property, you can reclaim the memory used by this bitmap by using the **Cls** method before setting AutoRedraw to **False**. For example, the following code reclaims the memory used by the Image property for a control called mypic:

```
mypic.AutoRedraw = True
mypic.Cls
mypic.AutoRedraw = False
```

Minimizing Stack Use

Visual Basic uses a *stack* to preserve your local variables and arguments during procedure calls. The size of the stack is fixed at 20K and cannot be changed. Some of this space is used by Visual Basic for purposes such as storing intermediate values when evaluating expressions, so the actual size of the stack available to your code is always smaller.

The remaining space is adequate for most purposes; however, if your procedures call procedures that call other procedures, and you declare many large local variables and arguments in each procedure, you could run out of stack space. The following sections describe techniques for minimizing the amount of space your application takes up on the stack.

Note One of the most common reasons for running out of stack space is unbounded recursion—a procedure that calls itself repeatedly until the stack is exhausted. In Visual Basic, this is often the result of a cascading event (such as changing the Text property for a text box in the Change event for that text box). When you encounter an `Out of stack space` error, look for cascading events first, before you try to reduce the amount of stack space your code uses. Use the Calls dialog box to examine the sequence of procedure calls that led to the `Out of stack space` error.

Reduce the Number and Size of Arguments and Local Variables

Every argument and nonstatic local variable consumes space on the stack. Avoid using large dynamic arrays and user-defined types as arguments or local variables. Consider using module-level or global variables instead of passing values as arguments to procedures, although this may make your code more difficult to maintain, modify, and debug.

Avoid Variants in Arguments and Local Variables

The **Variant** data type is larger than any of the other data types, so it consumes more of the stack when you use it for local variables and arguments.

Declare Procedures with the Static Keyword

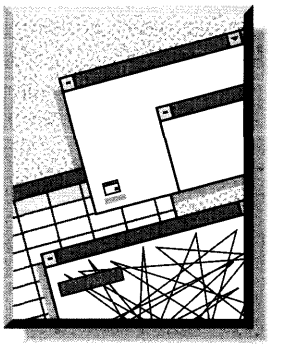
If you aren't calling a procedure recursively, declare the procedure with the **Static** keyword. This places all of the procedure's local variables in regular memory rather than on the stack. Because static variables are never discarded, however, they do consume memory in your application as long as it is running. Moreover, static variables are not reset to their initial values each time the procedure is called, so the code in the procedure must be written to handle this. If you don't want to declare the entire procedure with **Static**, consider using **Static** to declare some of the local variables.

Eliminate Local Fixed-Length String Variables

Replace large local fixed-length strings with local variable-length strings (or make the fixed-length strings static). Local fixed-length string variables consume one byte on the stack for each character in the string, plus some overhead. Local variable-length strings consume only four bytes on the stack per string, regardless of the length of the string.

CHAPTER 12

Responding to Mouse Events



Your Visual Basic applications can respond to a variety of mouse events. For example, forms, picture boxes, and image controls can detect the position of the mouse pointer, determine whether a left or right mouse button is being pressed, and can respond to different combinations of mouse buttons and SHIFT, CTRL, or ALT keys.

In addition, Visual Basic applications can support *drag-and-drop* features. You can use the **Drag** method with certain properties and events to enable operations such as dragging between forms. You can also support drag operations for users without a mouse.

Contents

- Mouse Events
- Positioning a Control with the Mouse
- Graphical Mouse Applications
- The Button and Shift Arguments
- Dragging and Dropping

MOUSE.MAK

Many of the code examples in this chapter are taken from the MOUSE.MAK sample application. If you installed the sample applications, you'll find this application in the \MOUSE subdirectory of the main Visual Basic directory (\VB\SAMPLES\MOUSE).

Mouse Events

You can use the following mouse events to enable your applications to respond to both the location and the state of the mouse. (This list excludes drag events, which are introduced later in this chapter.) These mouse events are recognized by forms, picture boxes, image controls, labels, and any control that has a list. Such controls include list boxes, combo boxes, file list boxes, drive list boxes, and directory list boxes.

Event	Description
MouseDown	Occurs when the user presses any mouse button.
MouseUp	Occurs when the user releases any mouse button.
MouseMove	Occurs each time the mouse pointer is moved to a new point on the screen.

A form can recognize a mouse event when the pointer is over a part of the form that doesn't contain any controls. A control can recognize a mouse event when the pointer is over the control.

When the user presses and holds down a mouse button, the same object continues to recognize all mouse events until the user releases the button. This is true even when the pointer is moved off the object.

The three mouse events use the following arguments.

Argument	Description
Button	A bit-field argument in which the three least-significant bits give the status of the mouse buttons.
Shift	A bit-field argument in which the three least-significant bits give the status of the SHIFT, CTRL, and ALT keys.
X, *Y*	Location of the mouse pointer, using the coordinate system of the object that receives the mouse event.

A *bit-field argument* returns information in individual bits, each indicating whether a certain condition is on or off. Techniques for programming with these arguments are described later, in the section "The Button and Shift Arguments."

Positioning a Control with the Mouse

Of the three mouse events, MouseDown is the most frequently used. In the application illustrated in Figure 12.1, the MouseDown event is combined with the **Move** method to move an image control to a different location on a form. The new location is determined by the position of the mouse pointer—when the user clicks anywhere on the form (except on the control), the control moves to that location.

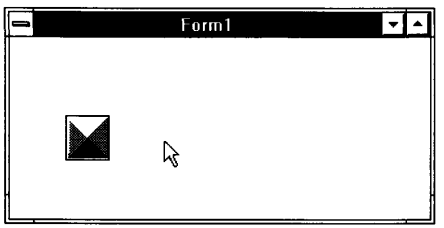


Figure 12.1 Moving a control to the location of the mouse pointer

A single procedure, Form_MouseDown, performs this application:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    Image1.Move X, Y
End Sub
```

The **Move** method places the image control's upper-left corner at the location of the mouse pointer, indicated by the arguments *X* and *Y*. You can revise this procedure to place the *center* of the control at the mouse location:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    Image1.Move (X - Image1.Width / 2), (Y - Image1.Height / 2)
End Sub
```

For More Information For information on the **Move** method, see the *Language Reference*, or search Help for *Move*.

Graphical Mouse Applications

This section shows how to use all three mouse events—MouseMove, MouseUp, MouseDown—and how to combine them with the graphics methods introduced in Chapter 15, "Creating Graphics for Applications." By combining mouse events with graphics methods, you can create any number of customized drawing or paint applications.

The Click-A-Line Application

The Click-A-Line application responds to a mouse click by drawing a line from the previous drawing location to the new position of the mouse pointer. This application uses both the MouseDown event and the **Line** method. Using the following syntax, the **Line** method will draw from the last point drawn to the point (*x2*, *y2*):

Line –(*x2*, *y2*)

Click-A-Line uses a blank form with one procedure, Form_MouseDown:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    Line -(X, Y)
End Sub
```

The first line starts at the upper-left corner, which is the default origin. Thereafter, whenever the mouse button is pressed, the application draws a straight line extending from the previous line to the present location of the mouse pointer. The result is a series of connected lines, as shown in Figure 12.2.

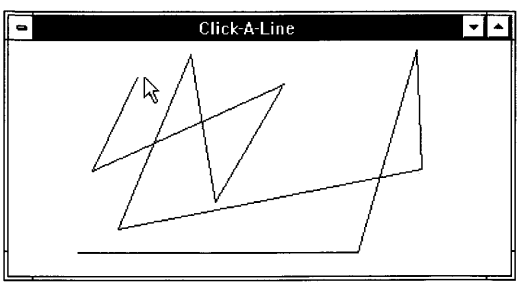


Figure 12.2 The Click-A-Line application uses the Line method to draw lines.

The Scribble Application

Graphics methods can produce very different effects when used in a MouseMove procedure instead of in a MouseDown procedure. For example, in the previous section, the **Line** method drew connected line segments. In the Scribble application, the same method is used in a Form_MouseMove procedure to produce a continuous curved line instead of connected segments.

In the Scribble application, the MouseMove event is recognized whenever the mouse pointer changes position. The following code draws a line between the current and previous location.

Figure 12.3 shows the result. Like the MouseDown procedure in the previous section, the line created by the MouseMove procedure starts at the upper-left corner:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    Line -(X, Y)
End Sub
```

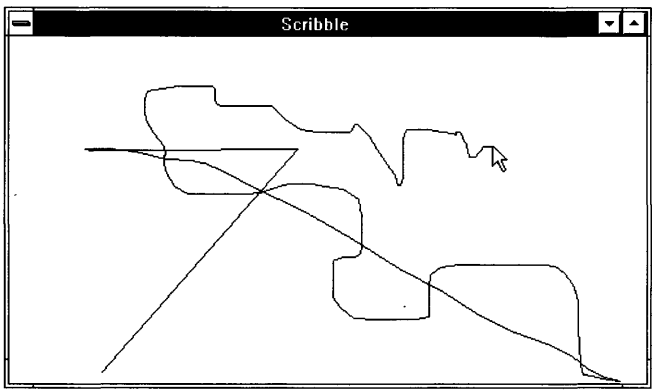


Figure 12.3 The Scribble application

How MouseMove Works

How many times does the MouseMove event get called as the user moves the pointer across the screen? Or, to put it another way, if you move the pointer from the top of the screen to the bottom, how many locations are involved?

Visual Basic doesn't necessarily generate a MouseMove event for every pixel the mouse moves over. The operating environment generates a limited number of mouse messages per second. To see how often MouseMove events are actually recognized, you can enhance the Scribble application with the following code so that it draws a small circle at each location where a MouseMove event is recognized. The results are shown in Figure 12.4.

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    Line -(X, Y)
    Circle (X, Y), 50
End Sub
```

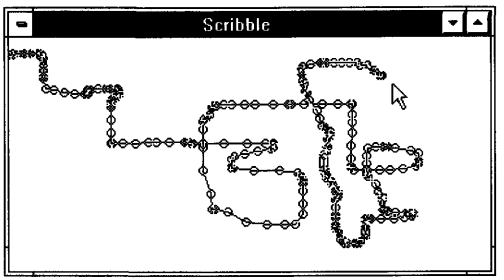


Figure 12.4 A demonstration of where MouseMove events occur

Note that the faster the user moves the pointer, the fewer MouseMove events are recognized between any two points. Many circles close together indicate that the user moved the mouse slowly.

Your application can recognize many MouseMove events in quick succession. Therefore, a MouseMove event procedure shouldn't do anything that requires large amounts of computing time.

Using All the Events Together

The Scribble application is more useful if it allows drawing only while the mouse button is held down and stops drawing when the button is released. To do this, the application would have to respond to three actions:

- The user presses the mouse button.
- The user releases the mouse button.
- The user moves the mouse pointer.

These actions correspond to the MouseDown, MouseUp, and MouseMove events. MouseDown and MouseUp will tell the application to turn drawing on and turn drawing off. You specify this by creating a form-level variable that represents the drawing state. Type the following statement in the Declarations section of form code:

```
Dim DrawNow As Integer
```

`DrawNow` will represent two values: **True** will mean "draw a line" and **False** will mean "do not draw a line."

Because variables are initialized to 0 by default, the application starts with drawing off. Then the first line in the MouseDown and MouseUp procedures turns drawing on or off by setting the value of the form-level variable DrawNow:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    DrawNow = True
    CurrentX = X
    CurrentY = Y
End Sub
Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    DrawNow = False
End Sub
```

The MouseMove procedure draws a line only if DrawNow is **True**. Otherwise, it takes no action:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If DrawNow Then Line -(X, Y)
End Sub
```

Each time the user presses a mouse button, the MouseDown event procedure is executed and turns drawing on. Then as the user holds the button down, the MouseMove event procedure is executed repeatedly as the pointer is dragged across the screen.

Note that the **Line** method omits the first endpoint, causing Visual Basic to start drawing at the mouse pointer's current coordinates. By default, the drawing coordinates correspond to the last point drawn; these were reset in the CurrentX and CurrentY assignment statements in Form_MouseDown.

The Button and Shift Arguments

You can make your applications more powerful by giving users additional options when they use the mouse. You can do this by writing code that responds differently to mouse events, depending on the button used or whether the SHIFT, CTRL, or ALT key is pressed. To provide these options, you use the arguments *Button* and *Shift* with the MouseDown, MouseUp, and MouseMove event procedures.

The Button Argument

The *Button* argument is an integer that represents the state of the mouse. As mentioned earlier in this chapter, the *Button* argument is a bit-field argument—a value in which each bit represents a state or condition. The three least-significant (lowest) bits represent the left, right, and middle mouse buttons, as shown in Figure 12.5.

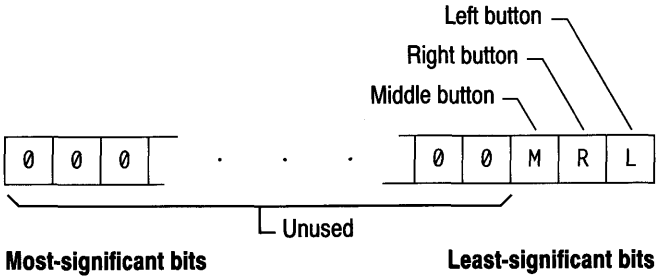


Figure 12.5 How bits represent the state of the mouse

The exact meaning of these bits is different for each mouse event.

Using Button with MouseDown and MouseUp

You use the *Button* argument with MouseDown to determine which button is being pressed, and with MouseUp to determine which button has been released. Because only one bit is set for each event, you can't test for whether two or more buttons are being used at the same time.

Note In contrast, you can use the MouseMove event to test for whether two or more buttons are being pressed simultaneously. You can also use MouseMove to test for whether only one button is being pressed, which you can't do with MouseDown or MouseUp. For more information, see the next section, "Using Button with MouseMove."

Possible binary values of *Button* are listed in the following table.

Value	Meaning
001 (decimal 1)	The left button caused the event.
010 (decimal 2)	The right button caused the event.
100 (decimal 4)	The middle button caused the event.

If a particular button isn't on the user's mouse, the corresponding bit is always 0. A two-button mouse has a left and right button but no middle button; a one-button mouse has only a left button.

You can determine which button causes a MouseDown or MouseUp event with simple code. The following procedure tests whether *Button* equals 1, 2, or 4:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button = 1 Then Print "You pressed the left button."
    If Button = 2 Then Print "You pressed the right button."
    If Button = 4 Then Print "You pressed the middle button."
End Sub
```

If the user presses more than one button, Visual Basic interprets that action as two or more separate MouseDown events. It sets the bit for the first button pressed, prints the message for that button, and then does the same for the next button. Similarly, Visual Basic interprets the release of two or more buttons as separate MouseUp events.

Using Button with MouseMove

For the MouseMove event, *Button* indicates the complete state of the mouse buttons—not just which button caused the event, as with MouseDown and MouseUp. This additional information is provided because all, some, or none of the bits might be set. This compares with just one bit per event in the MouseDown and MouseUp procedures.

Other binary values are listed in the following table.

Value	Meaning
111 (decimal 7)	All three buttons are pressed.
011 (decimal 3)	Left and right buttons pressed, not middle.
000 (decimal 0)	No buttons are pressed.

If you test MouseMove for equality to 001 (decimal 1), you're testing to see if *only* the left mouse button is being held down while the mouse is moved. If another button is held down with the left button, the following code doesn't print anything:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button = 1 Then Print "You pressed only the left button."
End Sub
```

To test for whether a particular button is down, ignoring other buttons, use the **And** operator. The following code prints the message for each button pressed, regardless of whether another button is pressed:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button And 1 Then Print "You're pressing the left button."
    If Button And 2 Then Print "You're pressing the right button."
End Sub
```

You can test for two buttons being pressed simultaneously by adding another line of code:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button And 1 Then Print "You're pressing the left button."
    If Button And 2 Then Print "You're pressing the right button."
    If (Button And 3) = 3 Then Print "You're pressing both buttons."
End Sub
```

The value of 3 is obtained by adding 1 and 2 together. This produces the binary pattern 011, in which both of the two least-significant bits are set. Using the **And** operator with 3 masks out all bits except the last two and then tests to see if these two bits are set.

There are several ways to test for multiple bits. One reliable way is to add the button constants (1, 2, or 4) together to produce a single number. You then use the combined **And** operator and equality test shown earlier. For example, the following statement tests for whether all three buttons are pressed (note that 7 is the sum of 1, 2, and 4):

```
If (Button And 7) = 7 Then Print "All three buttons pressed."
```

You may find that your code is more readable and easier to maintain if you define constants for the buttons you're detecting:

```
Const LEFT_BUTTON = 1
Const RIGHT_BUTTON = 2
Const MIDDLE_BUTTON = 4
```

For More Information These constants are defined in the file CONSTANT.TXT. For information on this file, search Help for *constants*.

Using Both Buttons in the Scribble Application

You can use the *Button* argument to enhance the Scribble application described earlier in this chapter. In addition to drawing a continuous line when the left mouse button is pressed, and stopping when the button is released, the application can draw a straight line from the last point drawn when the user presses the right button.

When writing code, it's often helpful to note each relevant event and the desired response. The three relevant events here are the mouse events:

- Form_MouseDown—Take a different action depending on the state of the mouse buttons:

  If the left button is down, set `DrawNow` to **True** and reset drawing coordinates.

  If the right button is down, draw a line.
- Form_MouseUp—If the left button is up, set `DrawNow` to **False**.
- Form_MouseMove—If `DrawNow` is **True**, draw a line.

The variable `DrawNow` and the button constants are both declared in the Declarations section of the form:

```
Dim DrawNow As Integer
Const LEFT_BUTTON = 1
Const RIGHT_BUTTON = 2
```

The MouseDown procedure has to take a different action, depending on whether the left or right mouse button caused the event:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥   Y As Single)
    If Button = LEFT_BUTTON Then
        DrawNow = -1
        CurrentX = X
        CurrentY = Y
    ElseIf Button = RIGHT_BUTTON Then
        Line -(X, Y)
    End If
End Sub
```

The following MouseUp procedure turns off drawing only when the left button is released:

```
Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Button = LEFT_BUTTON Then DrawNow = 0
End Sub
```

Note that within the MouseUp procedure, a bit set to 1 indicates that the corresponding mouse button is released.

The following MouseMove procedure is identical to the one in the previous version:

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If DrawNow Then Line -(X, Y)
End Sub
```

Detecting SHIFT, CTRL, and ALT States

You use the argument *Shift* in roughly the same way as *Button*. If the SHIFT key is pressed, *Shift* is 1; if the CTRL key is pressed, *Shift* is 2; and if the ALT key is pressed, *Shift* is 4. To determine combinations of these keys, use the total of their values. For example, if SHIFT and ALT are pressed, *Shift* equals 5 (1+4).

The three least-significant bits in *Shift* correspond to the state of the SHIFT, CTRL, and ALT keys, as shown in Figure 12.6.

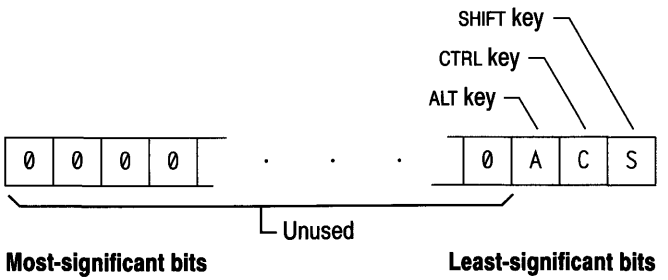


Figure 12.6 How bits represent the state of the SHIFT, CTRL, and ALT keys

Any or all of the bits in *Shift* can be set, depending on the state of the SHIFT, CTRL, and ALT keys. If only one key is pressed, the binary patterns are 001 (decimal 1) for SHIFT, 010 (decimal 2) for CTRL, and 100 (decimal 4) for ALT. If all are pressed, *Shift* has the binary pattern 111 (decimal 7). If only SHIFT and ALT are pressed, *Shift* has the binary pattern 101 (decimal 5).

As with the argument *Button*, you can use the **And** operator to mask out all bits except the ones you want to test for:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    If Shift And 1 Then Print "You pressed the SHIFT key."
    If Shift And 2 Then Print "You pressed the CTRL key."
    If Shift And 4 Then Print "You pressed the ALT key."
    If (Shift And 3) = 3 Then Print "You pressed both SHIFT and CTRL."
    If (Shift And 5) = 5 Then Print "You pressed both SHIFT and ALT."
    If (Shift And 6) = 6 Then Print "You pressed both CTRL and ALT."
    If (Shift And 7) = 7 Then Print "You pressed SHIFT, CTRL, and ALT."
End Sub
```

Dragging and Dropping

When you design Visual Basic applications, you often drag controls around on the form. The drag-and-drop features in Visual Basic allow you to extend this ability to the user at run time. The action of holding a mouse button down and moving a control is called *dragging*, and the action of releasing the button is called *dropping*.

This *drag-and-drop* capability extends to multiple-form operations. The user can drag the control anywhere on the screen, including other forms. You can respond to any drag operation as long as the destination is somewhere within your application.

As the user drags a control, Visual Basic provides a gray outline that moves with the mouse pointer, as shown in Figure 12.7.

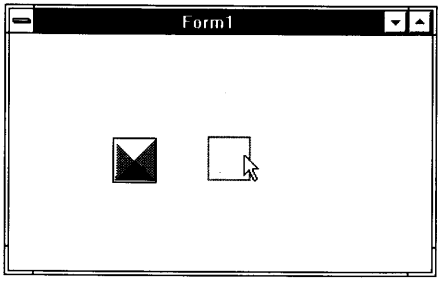


Figure 12.7 Dragging a control at run time

Note Run-time dragging of a control doesn't automatically change its location. You can do this, but you must program the relocation yourself, as described in the section "Causing Control Movement," later in this chapter. Often, dragging is used only to indicate that some action should be performed; the control retains its original position after the user releases the mouse button.

Using the following drag-and-drop properties, events, and method, you can specify both the meaning of a drag operation and how dragging can be initiated (if at all) for any given control.

Category	Item	Description
Properties	DragMode	Enables automatic or manual dragging of a control.
	DragIcon	Specifies what icon is displayed when the control is dragged.
Events	DragDrop	Recognizes when a control is dropped onto the object.
	DragOver	Recognizes when a control is dragged over the object.
Methods	Drag	Starts or stops manual dragging.

All controls except menus and timers can be dragged at run time and all controls share the properties listed in the preceding table. Forms recognize the DragDrop and DragOver events, but don't have DragMode and DragIcon properties.

Enabling Automatic Drag Mode

To allow the user to drag a control, set its DragMode property to 1. This enables automatic dragging of the control:

```
MyControl.DragMode = 1
```

You set the DragMode property to 1-Automatic at design time using the Properties window. When you set dragging to Automatic, dragging is always "on." Later, you'll see how to use the 0-Manual setting to control dragging.

Note While an automatic drag operation is taking place, the control being dragged doesn't recognize other mouse events.

Selecting the Drag Icon

As mentioned previously, Visual Basic normally uses a gray outline of the control being dragged rather than the control itself. You can substitute other images for the outline by setting the DragIcon property. This property contains an integer that corresponds to a graphic image. You can't set this integer value directly, but you can select a graphic image.

The easiest way to set the DragIcon property is to use the Properties window. After you select the DragIcon property, the arrow at the right of the Settings box changes to three dots. Click the three dots to select a file containing a graphic image from the Load Icon dialog box.

You can assign icons to the DragIcon property from the Icon Library included with Visual Basic. (The icons are pictured in Appendix B, "Icon Library.") You can also create your own drag icons with a graphics program. At run time, drag icons are always displayed in black and white.

At run time, you can select a drag icon image by assigning the DragIcon property of one control to the same property of another:

```
Image1.DragIcon = Image2.DragIcon
```

You can also set the DragIcon property at run time by assigning the Picture property of one control to the DragIcon property of another:

```
Image1.DragIcon = Image3.Picture
```

Or, you can use the **LoadPicture** function:

```
Image1.DragIcon = LoadPicture("C:\vb\icons\computer\disk04.ico")
```

For More Information For information on the Picture property and the **LoadPicture** function, see Chapter 15, "Creating Graphics for Applications," or search Help for *picture*.

Responding When the User Drops the Object

When the user releases the mouse button after dragging a control, Visual Basic generates a DragDrop event. You can respond to this event in many ways. Dropping may relocate the control at the new location (indicated by the last position of the gray outline). Remember that the control doesn't automatically move to the new location.

Two terms are important when discussing drag-and-drop operations—*source* and *target*.

Term	Meaning
Source	The control being dragged. This control can be any type except a menu or timer.
Target	The object onto which the user drops the control. This object, which can be a form or control, recognizes the DragDrop event.

A control becomes the target if the mouse position is within its borders when the button is released. A form is the target if the pointer is in a blank portion of the form.

The DragDrop event provides three arguments: *Source*, *X*, and *Y*. The argument *Source* is a reference to the control that was dropped onto the target. For example, if the user drops a control onto a blank portion of the form, the Form_DragDrop procedure is invoked:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
End Sub
```

Because *Source* is declared **As Control**, you use it just as you would a control—you can refer to its properties or call one of its methods. The Tag property is useful in this regard, because you can assign any string to it without causing side effects. Unlike other properties, Tag is used solely for identification and data storage.

This example assumes that you've set each control's Tag property to the name of the control:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
    Print Source.Tag; " dropped on background portion of form."
End Sub
```

Note You should use the *Source* argument carefully. Although you know that it always refers to a control, you don't necessarily know which type of control. For example, if the control is a text box and you attempt to refer to Source.Value, the result is a run-time error because text boxes have no Value property.

You can use the **If...Then...Else** statement with the **TypeOf** reserved word to determine what kind of control was dropped.

For More Information For information on **TypeOf**, see Chapter 8, "Objects and Instances," or search Help for *TypeOf*.

All possible control types for *Source* have a Visible property. Therefore, you can make a control invisible when it's dropped on a certain part of a form or on another control. The following procedure causes a control to disappear when it's dropped on a picture box named Vanish:

```
Sub Vanish_DragDrop (Source As Control, X As Single, Y As Single)
    Source.Visible = 0
End Sub
```

Highlighting Objects While Dragging

You may want to highlight an object as the user drags a control over it. Highlighting informs the user that if the mouse button is released, the object is dropped in the highlighted area. This is useful, particularly if the drop causes a move or copy operation or (as in the preceding example) makes something disappear.

Another way to indicate that a control is over a target is to change the drag icon. You can do this by changing the DragIcon property of the control being dragged (source control).

To respond to the fact that a dragged control is entering, passing over, or leaving an object, add statements to the DragOver event. The DragOver event procedure for a form looks like this:

```
Sub Form_DragOver (Source As Control, X As Single, Y As Single,
➥ State As Integer)
End Sub
```

The DragOver event has the same arguments as the DragDrop event. In addition, it has a *State* argument which indicates one of the following actions.

Value of state	Description
0	Enter (the mouse position just entered the object's borders).
1	Leave (the mouse position just left the object's borders).
2	Over (the mouse position moved from one part of the object to another).

Typically, you'll use the Enter (0) and Leave (1) states to create and remove highlighting. The Over (2) state is useful if you want to know when the mouse position is over a particular subdivision of an object (such as a region within a picture box). To respond that way, you would need to evaluate the arguments *X* and *Y*, which give the mouse position within the object.

To create or remove highlighting with Visual Basic, assign values to BackColor and possibly ForeColor as well. For example, the following event procedure inverts colors when the user drags a control over Text1 (Enter) and inverts colors back to their original state when the control is dragged off the text box (Leave):

```
Sub Text1_DragOver (Source As Control, X As Single, Y As Single,
➥ State As Integer)
    Const ENTER = 0
    Const LEAVE = 1
    Dim TempColor As Long

    If State = ENTER Or State = LEAVE Then
        TempColor = Text1.BackColor
        Text1.BackColor = Text1.ForeColor
        Text1.ForeColor = TempColor
    End If
End Sub
```

The constants ENTER, LEAVE, and OVER are defined in the CONSTANT.TXT file.

Controlling When Dragging Starts or Stops

Visual Basic has a Manual setting for the DragMode property that gives you more control than the Automatic setting. The Manual setting allows you to specify when a control can and cannot be dragged. (When DragMode is set to Automatic, the control can always be dragged as long as the setting isn't changed.)

For instance, you may want to enable dragging in response to MouseDown and MouseUp events, or in response to a keyboard or menu command. The Manual setting also allows you to recognize a MouseDown event before dragging starts, so that you can record the mouse position.

To enable dragging from code, leave DragMode in its default setting (0-Manual). Then use the **Drag** method whenever you want to begin or stop dragging an object:

[*control.*]**Drag** *action*

If *action* is 1, the **Drag** method initiates dragging of the control. If *action* is 2, the control is dropped, causing a DragDrop event. The value 0 for *action* cancels the drag. The effect is similar to giving the value 2, except that no DragDrop event occurs.

For example, this statement enables (turns on) dragging of a control named ThisBox:

```
ThisBox.Drag 1
```

This statement drops the control and has the same effect as releasing a mouse button:

```
ThisBox.Drag 2
```

Causing Control Movement

You may want the source control to change position after the user releases the mouse button. To make a control move to the new mouse location, just use the **Move** method (similar to the code in the first example in this chapter):

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
    Source.Move X, Y
End Sub
```

This code may not produce precisely the effects you want, because the upper-left corner of the control is positioned at the mouse location. This code positions the center of the control at the mouse location:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
    Source.Move (X - Source.Width / 2), (Y - Source.Height / 2)
End Sub
```

The code works best when the DragIcon property is set to a value other than the default (the gray rectangle). When the gray rectangle is being used, the user normally wants the control to move precisely into the final position of the gray rectangle. To do this, record the initial mouse position within the source control. Then use this position as an offset when the control is moved.

▶ **To record the initial mouse position**

1. Specify manual dragging of the control.
2. Declare two form-level variables, `DragX` and `DragY`.
3. Turn on dragging when a MouseDown event occurs. Also, store the value of *X* and *Y* in the form-level variables in this event.
4. Turn off dragging on MouseUp.

The following example illustrates how to cause drag movement for an image control named Image1. The control's DragMode property should be set to 0-Manual at design time. The Declarations section contains the form-level variables DragX and DragY, which record the initial mouse position within the image control:

```
Dim DragX As Single, DragY As Single
```

The MouseDown and MouseUp procedures for the control turn dragging on and drop the control, respectively. In addition, the MouseDown procedure records the mouse position inside the control at the time dragging begins:

```
Sub Image1_MouseDown (Button As Integer, Shift As Integer,
➥ X As Single, Y As Single)
    Image1.Drag 1
    DragX = X
    DragY = Y
End Sub

Sub Image1_MouseUp (Button As Integer, Shift As Integer,
➥ X As Single, Y As Single)
    Image1.Drag 2
End Sub
```

The Form_DragDrop procedure actually moves the control. To simplify this example, assume that Image1 is the only control on the form. The target can therefore only be the form itself. The Form_DragDrop procedure repositions the control, using DragX and DragY as offsets:

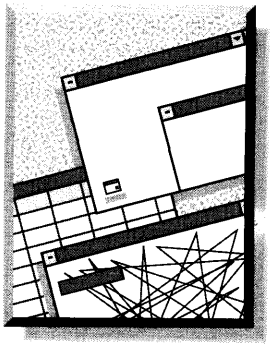
```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
    Source.Move (X - DragX), (Y - DragY)
End Sub
```

Note that this example assumes that Image1 and the form use the same units in their respective coordinate systems. If they don't, then you'll have to convert between units.

For More Information For information on coordinate systems, see Chapter 15, "Creating Graphics for Applications," or search Help for *ScaleMode*.

CHAPTER 13

Using the Grid Control



Some applications need to display information in rows and columns. You can use the grid control in Visual Basic to create applications that present tables of information.

Contents

- The Grid Control
- Creating and Sizing Rows and Columns
- Using Text in the Grid Control
- Using Graphics in the Grid Control
- Selecting and Highlighting Cells
- Adding and Removing Rows

LOAN.MAK

Many of the code examples in this chapter are taken from the LoanSheet sample application (LOAN.MAK). If you installed the sample applications, you will find this application in the \GRID subdirectory of the main Visual Basic directory (\VB\SAMPLES\GRID).

The Grid Control

The grid control lets you display information in a series of rows and columns, including the special rows and columns that display row and column headings. You can program the application to change the contents of the grid control using the properties and methods described in this chapter.

Note The grid control is a custom control. Custom controls are separate files with a .VBX extension. To use the grid control in your application, you must add the file GRID.VBX to the project. For more information on how to add a custom control to a project, see Chapter 5, "Managing Projects."

Figure 13.1 shows the grid control (named grdPayments) used in the LoanSheet application.

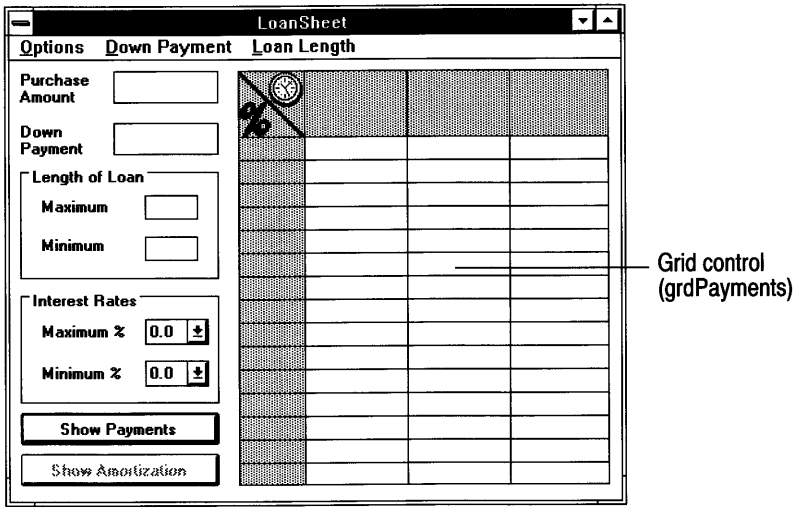


Figure 13.1 Grid control in the LoanSheet application

The grid control displays information in cells. A *cell* is a location in the grid control at which a row and a column intersect. You can set the current cell in code. The user can set the current cell by clicking a cell or by using the arrow keys. Although users can select or highlight a cell at run time, they cannot edit or alter its contents.

Creating and Sizing Rows and Columns

You can create two kinds of rows or columns in the grid control. The most common is non-fixed. A *non-fixed* row or column scrolls when scroll bars are active in the grid control; a *fixed* row or column does not scroll at any time.

The default color of cells in a non-fixed row or column is white. Cells in a fixed row or column are gray. Figure 13.2 shows the difference between fixed and non-fixed rows and columns in a grid.

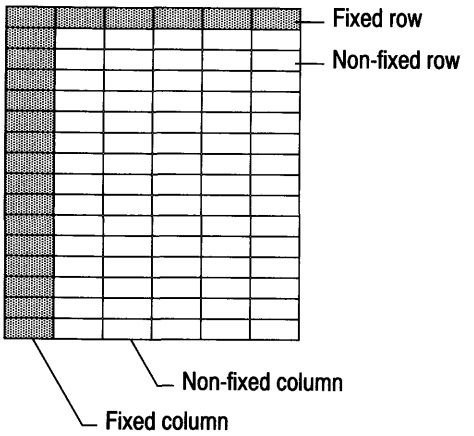


Figure 13.2 Fixed and non-fixed rows and columns in the grid control

Creating Rows and Columns

You create rows and columns in a grid by setting four properties of the grid control. The following table describes these four properties.

Property	Description
Rows	Sets or returns the total number of fixed and non-fixed rows in the grid.
Cols	Sets or returns the total number of fixed and non-fixed columns in the grid.
FixedRows	Sets or returns the number of fixed rows in the grid.
FixedCols	Sets or returns the number of fixed columns in the grid.

In the LoanSheet application, the initial number of rows and columns is set by the following property values.

Property	Initial value
grdPayments.Rows	16
grdPayments.Cols	6
grdPayments.FixedRows	1
grdPayments.FixedCols	1

When LoanSheet displays the monthly loan payments, the initial number of rows or columns may be too few to display all the combinations of interest rates and repayment periods chosen by the user. So at run time, LoanSheet adjusts the number of rows and columns to accommodate the data presented. These statements from the SizeCells procedure prepare the grid control to display data using a default number of rows and columns:

```
' If data fits within the default number of rows and columns...
If Rates <= 16 And Periods <= 3 Then
    ThisGrid.ScrollBars = 0             ' Don't display scroll bars.
    ThisGrid.Rows = 17                  ' Default number of rows.
    ThisGrid.Cols = 4                   ' Default number of columns.
```

Changing the Size of Rows and Columns

The grid control has properties that set or return the height of any given row in the grid or the width of any given column. The height of a row is set by the RowHeight property. The width of a column is set by the ColWidth property. You can set these two properties only at run time. They are not available at design time.

When the LoanSheet application displays the various monthly loan payments it has calculated, it resizes the width of the columns in the grid control using statements like these from the SizeCells procedure:

```
' Set row heights to display 16 rows in non-fixed grid height.
For Count = 1 To 16
    ThisGrid.RowHeight(Count) = GridHgt / 16
Next Count
' Set column widths to display 3 columns in non-fixed grid width.
For Count = 1 To 3
    ThisGrid.ColWidth(Count) = GridWid / 3
Next Count
```

Using Text in the Grid Control

The most common use of the grid control is to display text. You cannot add text to a grid control at design time; however, Visual Basic gives you several ways of adding text to one or more cells in the grid control at run time. This section examines how to add and align text in the grid control at run time.

Adding Text to Cells

There are two ways of adding text to cells to a grid:

- Setting the Text property
- Setting the Clip property

Use the Text property to add text to a single cell or to add the same text to a range of cells in the grid control. Use the Clip property to add different text to various cells in a selected range.

Setting the Text Property

The easiest way to add text to a single cell is to set the Text property of the grid control. The Text property applies to the cell defined by the current value of the Row and Col properties of the grid control.

For example, these statements from the ShowPmnts procedure to the LoanSheet application use the Text property to add the interest rate headings in the fixed column. Notice that the Row and Col properties are set before assigning a string to the Text property.

```
' Make the first column the current column.
grdPayments.Col = 0
For Row2Fill = 1 To Rates
    ' Select the current row.
    grdPayments.Row = Row2Fill
    ' Display the current interest rate in the cell.
    grdPayments.Text = Format$(Payments(Row2Fill, 0), "0.0") & "%"
Next Row2Fill
```

Setting the Text Property for Multiple Cells

You can add the same text to a highlighted range of cells using the Text and FillStyle properties together. When FillStyle = 0, the text is added only to the cell indicated by the current values of the Row and Col properties. When FillStyle equals 1, the text assigned to the Text property is added to all the cells whose CellSelected property is **True**.

For example, the LoanSheet application uses the FillStyle property in a general procedure called ClearGrid to clear the contents of the grid control when displaying monthly payment or amortization data. This procedure clears the grid:

```
' Select all the non-fixed cells in the grid control.
grdPayments.SelStartCol = 1
grdPayments.SelStartRow = 1
grdPayments.SelEndCol = grdPayments.Cols - 1
grdPayments.SelEndRow = grdPayments.Rows - 1
' Set grid to fill all cells with the same text.
grdPayments.FillStyle = 1
' Put blank text in all the cells.
grdPayments.Text = ""
' Set grid to fill only the current cell.
grdPayments.FillStyle = 0
' Set the current cell.
grdPayments.SelEndCol = 1
grdPayments.SelEndRow = 1
```

Note The Select properties—SelStartCol, SelStartRow, and SelEndCol—are described later in this chapter.

Setting the Clip Property

In some case, you may need to fill an entire range of cells at one time. For example, you may want to paste a large block of information from the Clipboard into a grid control. You can use the Clip property to add text to a selected range of cells using this syntax:

Grid.**Clip**[= *stringexpression*]

The Clip property fills all the cells in the selected range with the text in the *stringexpression*. The string must use a tab character (**Chr$**(9)) to indicate the next cell in the row and a carriage-return character (**Chr$**(13)) to indicate the next row in the selection.

The Clip property affects only those cells in the grid control that have been selected. If the selection contains more cells than are described in the *stringexpression*, the additional cells are emptied of text. If the selection contains fewer cells than the *stringexpression*, the remaining text in the *stringexpression* is ignored.

Aligning Text

You can control the alignment of text in the columns of a grid control using two properties:

- ColAlignment (non-fixed columns)
- FixedAlignment (fixed columns)

Using these properties, you can align text flush left, center, or flush right within the cells of the selected column. When a column contains cells from both fixed and non-fixed rows, the ColAlignment property affects only the non-fixed cells in that column, while the FixedAlignment property affects only the fixed cells in that column.

Example

These statements from the ShowPmnts procedure in the LoanSheet application set the text alignment of the column headings to center:

```
' Make the first row the current row.
grdPayments.Row = 0
' Put the column headings in the first row.
For Col2Fill = 1 To Periods
    ' Select the current column.
    grdPayments.Col = Col2Fill
```

```
    ' Set text alignment in the fixed cells to center.
    grdPayments.FixedAlignment(Col2Fill) = 2
    ' Display the current loan period in the cell.
    grdPayments.Text = Payments(0, Col2Fill)
Next Col2Fill
```

Figure 13.3 shows the grid control in LoanSheet with all the monthly payment text in place.

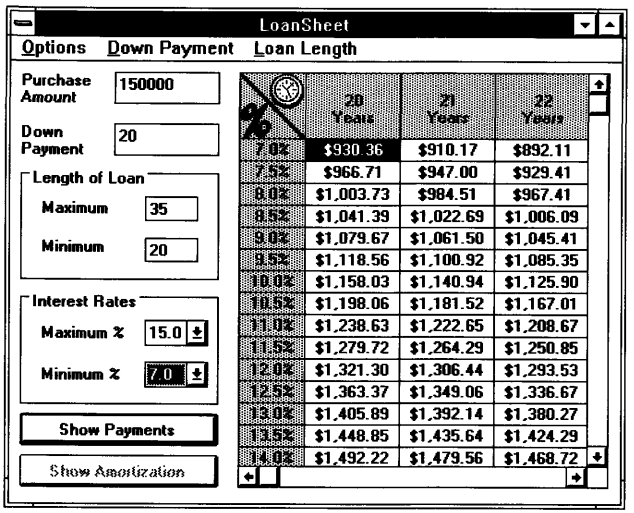


Figure 13.3 The LoanSheet application with text in the grid control

Using Graphics in the Grid Control

You can add a graphic to a cell in the grid control with the Picture property. Like the Text property, the Picture property applies to the current cell as specified by the Row and Col properties of the grid control.

Note You can display bitmaps and icons—but not metafiles—in the cells of a grid control.

Adding a Graphic

The LoanSheet application includes a graphic in the upper-left cell of the grid. This graphic serves as a heading for the top row (showing time) and first column (showing interest rates). Because you cannot set the Picture property for the grid control at design time, LoanSheet stores this graphic in an image control with its Visible property set to **False**. Figure 13.4 shows this image control.

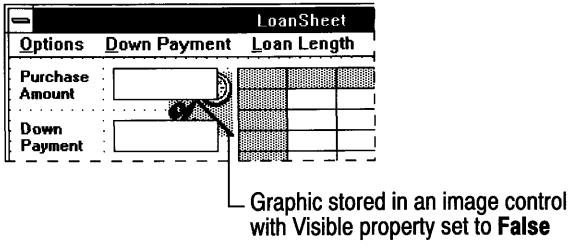


Figure 13.4 Using an image control to store a graphic for the LoanSheet grid control

The application adds this graphic to the first cell in the grid during the Load event procedure for the LoanSheet form using these statements:

```
' Place the grid graphic in cell (0, 0).
grdPayments.Row = 0
grdPayments.Col = 0
grdPayments.ColWidth(0) = imgGraphic.Width
grdPayments.RowHeight(0) = imgGraphic.Height
grdPayments.Picture = imgGraphic.Picture
```

Notice that this procedure also adjusts the column width and row height of the first cell so the graphic isn't cropped by a cell too small to display it entirely.

Removing a Graphic

To remove a graphic from a cell in a grid control, use this syntax:

Grid.**Picture = LoadPicture( )**

The LoanSheet application uses this syntax to remove the graphic in the first cell when displaying amortization data for a selected repayment period and interest rate combination. The following statements from the ShowAmort procedure show how LoanSheet removes the graphic to replace it with text:

```
' Replace the picture from the (0, 0) cell with a column heading.
grdPayments.Row = 0
grdPayments.Col = 0
grdPayments = LoadPicture()
grdPayments.Text = NL & "Payment" & NL & Format$(AmortPmnt, "Currency")
grdPayments.ColWidth(0) = 1.1 * TextWidth(grdPayments.Text)
```

Setting the Picture Property for Multiple Cells

By using the Picture and FillStyle properties together, you can add the same graphic to a highlighted range of cells. When FillStyle equals 0, the graphic is added only to the cell indicated by the current values of the Row and Col properties. When FillStyle equals 1, the graphic assigned to the Picture property is added to all the cells whose CellSelected property is **True**.

Using Graphics and Text in a Cell

You can include both a graphic and text in the same cell. The graphic remains positioned in the upper-left corner of the cell. The text appears in the remaining space between the right edge of the graphic and the right edge of the cell. Figure 13.5 shows text and graphics in the same cell with different text alignments.

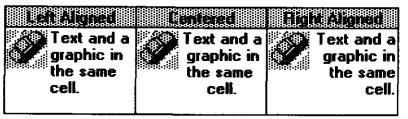


Figure 13.5 Examples of a graphic and text in the same cell

Selecting and Highlighting Cells

Although users cannot edit the contents of the cells in a grid control, they can select and highlight one or more cells. This ability is useful if you want the user to select a portion of the displayed information for some purpose. The user selects a range of cells by clicking a cell and then dragging the mouse.

If the grid control contains a fixed column and a fixed row, you can select all the cells in the grid by clicking the cell where row 0 and column 0 intersect. You can also select an entire row by clicking the cell in the fixed column of that row, and you can select an entire column by clicking the cell in the fixed row of that column.

Highlighting lets you control the appearance of cells when they are selected. The background color of highlighted cells changes to indicate that they have been selected. You can control whether the grid control highlights selected cells.

Figure 13.6 shows a grid control with a range of cells selected and highlighted in the LoanSheet application.

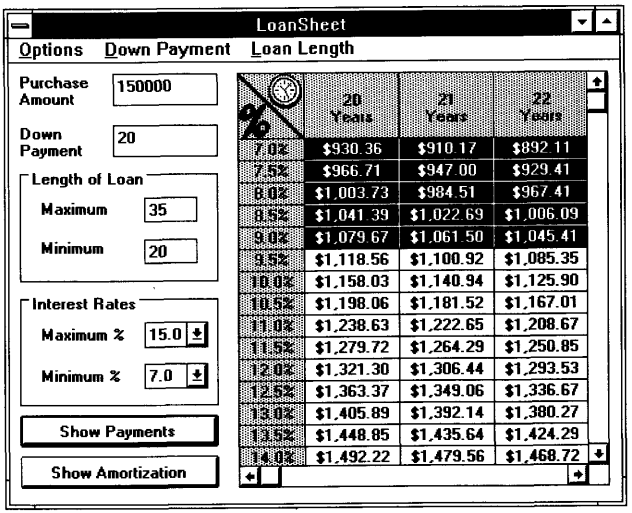


Figure 13.6 Selected and highlighted cells in a grid control

The grid control has several properties that specify various aspects of selecting and highlighting cells. The following table briefly describes these properties.

Property	Description
HighLight	When HighLight is **True**, selected cells in the grid control are highlighted.
SelEndCol	Specifies the number of the rightmost column in a selected range of cells.
SelEndRow	Specifies the number of the bottom row in a selected range of cells.
SelStartCol	Specifies the number of the leftmost column in a selected range of cells.
SelStartRow	Specifies the number of the top row in a selected range of cells.

The ClipCopy procedure in the LoanSheet application uses these properties to copy the contents, row headings, and column headings of a selected range of cells to the Clipboard.

The following statements deal only with the highlighted cells in the grid control:

```
' Initialize variables for new column (NC) and new row (NR).
NR = Chr$(13)
NC = Chr$(9)
' Initialize local variables for the rows and columns selected.
ColStrt = grdPayments.SelStartCol
ColEnd = grdPayments.SelEndCol
RowStrt = grdPayments.SelStartRow
RowEnd = grdPayments.SelEndRow
' Initialize the local variable that receives input from the grid.
ClipText = grdPayments.Clip
' Initialize the local variable that holds output for the Clipboard.
CopyText = ""
' Transfer cell text from the input variable to the output variable.
For Count = 1 To Len(ClipText)
    ' If the current character isn't a carriage return...
    If Mid$(ClipText, Count, 1) <> Chr$(13) Then
        ' ...append the character to the output variable contents.
        CopyText = CopyText & Mid$(ClipText, Count, 1)
    ' If the current character is a carriage return...
    Else
        ' ...add a carriage return and linefeed to the output variable.
        CopyText = CopyText & NR & grdPayments.Text & NC
    End If
Next Count
' Copy the final contents of the output variable into the Clipboard.
Clipboard.SetText CopyText
```

After copying the contents of the cells and the appropriate column and row headings to the Clipboard, you can paste the information into another application, such as the Microsoft® Write word processing program. Figure 13.7 shows the information highlighted in Figure 13.6 copied to the Clipboard and pasted into Microsoft Write.

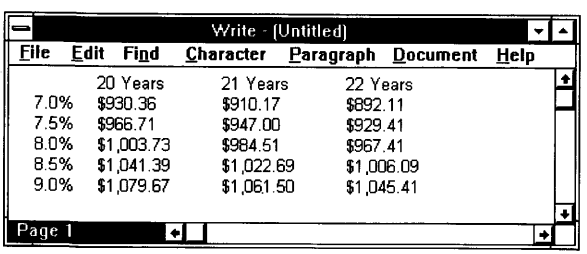


Figure 13.7 LoanSheet data pasted from the Clipboard into Microsoft Write

Adding and Removing Rows

You can add and fill the cells of new rows in the grid control at run time. You can also remove rows from the grid. The ability to add and remove rows can be very helpful in applications that use the grid control to display data records. In applications like these, you can insert a new record by adding a row or delete a record by removing its row.

Adding Rows

You add new rows and their contents at run time with the **AddItem** method. The syntax for the **AddItem** method is:

Grid.**AddItem** *item*[, *index*]

In this syntax, *item* is a string that contains the contents of the first cell in the new row. By using tab characters in the string, *item* can specify the contents of all the cells in the row. For example, the following statement inserts a row and the contents of all the cells in a three-column grid:

```
Grid1.AddItem "1st cell" & Chr$(9) & "2nd cell" & Chr$(9) & "3rd cell"
```

The value of *index* specifies the row number of the position you want the new row to occupy in the grid control.

Note You cannot use the **AddItem** method with a fixed row.

Removing Rows

You can also remove rows at run time. To remove rows, you use the **RemoveItem** method. The syntax for the **RemoveItem** method is:

Grid.**RemoveItem** *index*

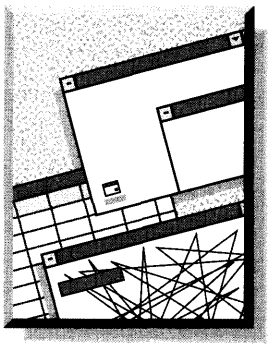
The value of *index* specifies the number of the row you want to remove from the grid control. For example, the following statements remove the third row from the grid each time a command button is clicked:

```
Sub Command1_Click ()
    Grid1.RemoveItem 2
End Sub
```

Note You cannot use the **RemoveItem** method with a fixed row.

CHAPTER 14

Multiple-Document Interface (MDI) Applications



The multiple-document interface (MDI) allows you to create an application that maintains multiple forms within a single container form. Applications such as Windows Program Manager, Windows File Manager, Microsoft® Excel, and Microsoft® Word for Windows™ have multiple-document interfaces.

An MDI application allows the user to display multiple documents at the same time, with each document displayed in its own window. Document windows are contained in a *parent window*, which provides a workspace for all the document windows in the application. For example, Windows Program Manager allows you to create and display many different program groups. Each group is displayed in its own window and is confined to the area of the Program Manager's parent window.

Contents

- Using MDI in Visual Basic
- Creating an MDI Application
- MDI NotePad—A Document-Centered Application
- Menus in MDI Applications
- Creating a Toolbar

MDINOTE.MAK

Many of the code examples in this chapter are taken from the MDI NotePad sample application. If you installed the sample applications, you will find this application in the \MDI subdirectory of the main Visual Basic directory (VB\SAMPLES\MDI).

Using MDI in Visual Basic

Your Visual Basic application can have only one MDI form, which contains all the application's child forms.

A *child form* is an ordinary form that has its MDIChild property set to **True**. Your application can include many MDI child forms.

At run time, child forms are displayed within the *internal area* of the MDI form (the area inside its borders and below the caption and menu bars). When a child form is minimized, its icon appears on the MDI form instead of on the desktop, as shown in Figure 14.1.

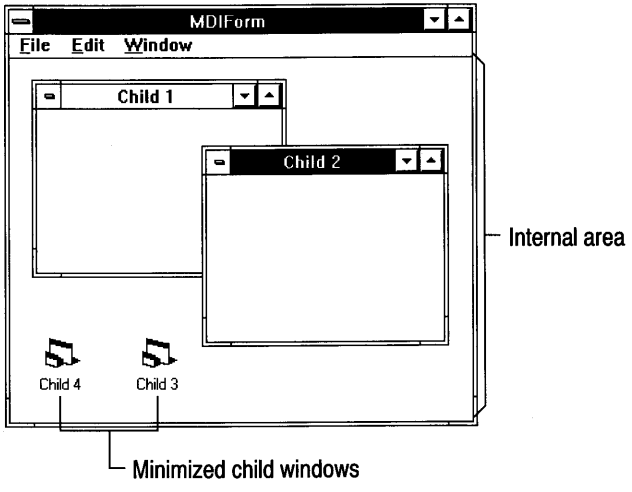


Figure 14.1 Child forms displayed within the internal area of the MDI form

Note Your application can also include standard, non-MDI forms that are not contained in the MDI form. A typical use of a standard form in an MDI application is to display a modal dialog box.

Creating an MDI Application

Use the following procedure to create an MDI form and its child forms.

▶ **To create an MDI application**

1. Create an MDI form.

   From the File menu, choose New MDI Form.

2. Create the application's child forms.

To create an MDI child form, create a new form (or open an existing one) and set its MDIChild property to **True**.

Note An application can have only one MDI form. If a project already has an MDI form, the New MDI Form menu command is unavailable.

Design-Time Features of MDI Child Forms

At design time, child forms are not restricted to the area inside the MDI form. You can add controls, set properties, write code, and design the features of child forms anywhere on the desktop—just as you can with any other Visual Basic form.

You can determine whether a form is an MDI child by looking at its MDIChild property, or by examining the Project window. If the MDIChild property is set to **True**, then it is a child form. Visual Basic displays special icons in the Project window for the MDI and MDI child forms, as in Figure 14.2.

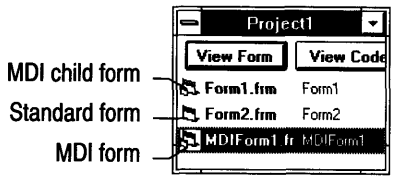


Figure 14.2 Icons in the Project window identify MDI child forms, standard forms, and MDI forms.

Tip If you are creating several MDI applications, you can save yourself work by modifying AUTOLOAD.MAK to include the MDI form and to set this form as the startup form. For more information on AUTOLOAD.MAK, see Chapter 5, "Managing Projects."

Run-Time Features of MDI Child Forms

At run time, the MDI form and all of its child forms take on special characteristics:

- All child forms are displayed within the MDI form's internal area. The user can move and size child forms like any other form; however, they are restricted to this internal area.
- When a child form is minimized, its icon appears on the MDI form instead of the user's desktop (see Figure 14.3). When the MDI form is minimized, the MDI form and all of its child forms are represented by a single icon. When the MDI form is restored, the MDI form and all the child forms are redisplayed to the state they were in before being minimized.

- When a child form is maximized, its caption is combined with the caption of the MDI form and is displayed in the MDI form's title bar (see Figure 14.3).
- You cannot hide a child form.
- The active child's menus (if any) are displayed on the MDI form's menu bar, not on the child form.

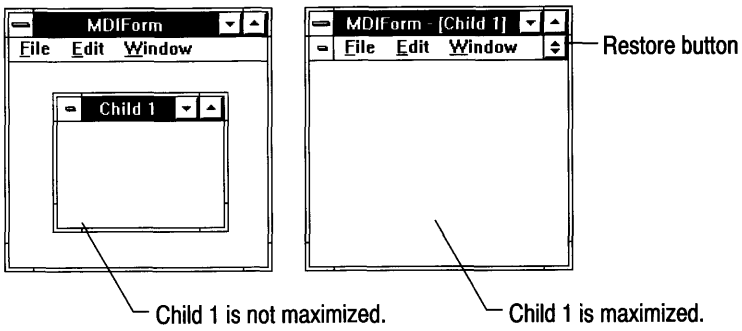


Figure 14.3 A child form caption combined with the caption of the MDI form

Windows Compatibility

The appearance of MDI applications varies depending on which version of Windows you are running. The BorderStyle, ControlBox, MinButton, and MaxButton properties are fully functional only in applications running in the Windows version 3.1 operating system. All child forms running in the Windows version 3.0 graphical environment have a sizable border and display a Control-menu box, a Minimize button, and a Maximize button.

Using Windows 3.0, you can disable the Control-menu box, Minimize button, or Maximize button on a child form by setting the appropriate property to **False**. The buttons will still be displayed on the child form; however, they will not respond to user clicks, nor will the corresponding commands appear on the form's Control menu.

If the ControlBox property is set to **False** in a Windows 3.0 MDI child form, the Control-box menu drops down, but the Close and Next Window menu commands are not displayed. In addition, the Size menu command is not displayed if the BorderStyle property is set to 0, 1, or 3.

MDI NotePad—A Document-Centered Application

As its name suggests, the multiple-document interface was designed especially for *document-centered* applications. Such applications allow the user to open many similar documents at the same time. For example, in Microsoft Excel, the user can work with several different spreadsheets at the same time, each spreadsheet appearing in a separate window.

To create such an application in Visual Basic, you need at least two forms—an MDI form and a child form. At design time, you create an MDI form to contain the application and a child form to serve as a template for the application's document. At run time, when the user requests a new document (usually implemented with the New command on the File menu), the application creates a new instance of the child form. This allows the user to create as many child forms, or documents, as necessary.

Example

You can create an application similar to the NotePad application that ships with Microsoft Windows. This NotePad application, however, uses MDI. The MDI NotePad application allows the user to enter text in a window. Each time the user chooses New from the File menu, a new child form is created and displayed.

▶ **To create the MDI NotePad application**

1. From the File menu, choose New Project.
2. From the File menu, choose New MDI Form to create the container form.

   The project should now contain an MDI form (MDIForm1) and a standard form (Form1).
3. Create a text box (Text1) on Form1.
4. Set properties for the two forms and the text box as follows.

Object name	Property	Setting
MDIForm1	Caption	MDI NotePad
Form1	Caption	Untitled
Form1	MDIChild	**True**
Text1	MultiLine	**True**
Text1	Text	(Empty)
Text1	Left	0
Text1	Top	0

5. In the Menu Design window, create a File menu for Form1.

Caption	Object name	Indented
&File	mnuFile	No
&New	mnuFileNew	Yes

6. Add the following code to the mnuFileNew_Click procedure:

```
Sub mnuFileNew_Click ()
    ' Create a new instance of Form1, called NewDoc.
    Dim NewDoc As New Form1
    ' Display the new form.
    NewDoc.Show
End Sub
```

This procedure creates and then displays a new instance (or copy) of Form1, called NewDoc. Each time the user chooses New from the File menu, an exact duplicate of Form1 is created, including all the controls and code that it contains.

7. Add the following code to the Form_Resize procedure for Form1:

```
Sub Form_Resize ()
    ' Expand text box to fill the current child form's internal area.
    Text1.Height = ScaleHeight
    Text1.Width = ScaleWidth
End Sub
```

The code for the Form_Resize event procedure, like all the code in Form1, is shared by each instance of Form1. When several copies of a form are displayed, each form recognizes its own events. When an event occurs, the code for that event procedure is called. Since the same code is shared by each instance, you might wonder how to reference the form that has called the code—especially since each instance has the same name (Form1).

If you're writing code that will be called by multiple instances of a form, it's a good idea not to use a form identifier when accessing the form's controls or properties. For example, refer to the height of the text box on Form1 as `Text1.Height` instead of `Form1!Text1.Height`. This way, the code always affects the current form.

Another way to specify the current form in code is to use the **Me** keyword. You use **Me** to reference the form whose code is currently running. This keyword is useful when you need to pass a reference to the current form instance as an argument to a procedure.

Yet another solution is to use the ActiveForm property of the MDI form. This property specifies the MDI child form that has the focus or that was most recently active. (See the section "Specifying the Active Child Form or Control" later in this chapter.)

For More Information For information on creating multiple instances of a form using the **New** keyword with the **Dim** statement, or using the **Me** keyword, see Chapter 8, "Objects and Instances"; see the *Language Reference*; or search Help for *Dim*.

Loading MDI Forms and Child Forms

When you load a child form, its parent form (the MDI form) is automatically loaded and displayed. When you load the MDI form, however, its children are not automatically loaded. You can see this when you run the preceding example.

Since the child form is the default startup form, both the child form and the MDI form are loaded and visible when the application is run. If you change the default in the MDI NotePad application to MDIForm1 (in the Project Options dialog box) and then run the application, only the MDI form is loaded. The first child form is loaded when you choose New from the File menu.

Note When an MDI child form has a sizable border (BorderStyle = 2), Windows determines its initial height, width, and position when it is loaded. The initial size and position of a child form with a sizable border is dependent on the size of the MDI form, not on the size of the child form at design time.

When an MDI child form's border is not sizable (BorderStyle = 0, 1, or 3), it is loaded using its design-time Height and Width properties.

You can't show an MDI child form or an MDI form with the modal argument. If you want to use a modal dialog box in an MDI application, use a form whose MDIChild property is set to **False**.

For More Information For information on the **Show** method, see the *Language Reference*, or search Help for *Show*.

Maintaining State Information for a Child Form

A user deciding to quit the MDI application must have the opportunity to save work. To make this possible, the application needs to be able to determine, at all times, whether or not the data in the child form has changed since the last time it was saved. This can be done using either a form-level variable or a form array.

Using a Form-Level Variable

Using module-level variables to keep track of state information is a useful technique as long as the information you are tracking is not complex and you do not need to access information for a child form outside of that form's module. (Module-level variables in a form are visible only inside that form's code, so the code in an MDI form can't access a child's module-level variables, and vice versa.)

First, declare a form-level variable in the Declarations section of a form:

```
Dim Dirty As Integer
```

Use this variable to keep track of the state of your child form. For example, in the MDI NotePad application, set this variable to **True** each time the text changes in Text1. This indicates that the contents of Text1 have changed since the last time it was saved.

```
Sub Text1_Change ()
    Dirty = True
End Sub
```

Conversely, for each time the user will save the contents of the child form, set `Dirty` to **False** to indicate that the contents of Text1 no longer need to be saved. The following code assumes there is a menu command called Save (mnuFileSave) and a procedure called FileSave that saves the contents of the text box:

```
Sub mnuFileSave ()
    ' Save the contents of Text1.
    FileSave
    ' Set the state variable.
    Dirty = False
End Sub
```

Using a Form Array

When writing a more sophisticated MDI application, you need to keep track of an assortment of data for each child form and access this data from different modules throughout your application.

A more powerful (and complicated) technique for dynamically creating child forms and keeping track of state information is to create two global arrays: an array of child forms and an array of state information. First, create a user-defined type. You use the elements of the new data type to maintain the information for each child form. Then declare a dynamic global array of the user-defined type and a parallel dynamic global array of new child forms:

```
Type FormState
    Dirty As Integer
    Deleted As Integer
End Type

Global FState () As FormState
Global Document () As New Form1
```

This code creates a user-defined type, FormState. The next-to-last line declares a global array of this type, called `FState ()`. Since the size of the array is not specified, it is a dynamic array; its size can be changed at any time using the **ReDim** statement. The last line declares a dynamic global array of the type `New Form1`. When this array is dimensioned, an instance of Form1 is created for each element in the array. For example, you can use the **ReDim** statement to create five instances of Form1:

```
' This procedure creates and then displays five instances of Form1.
Sub Command1_Click ()
    ReDim Document(4)
    For I = 0 To 4
        Document(I).Show
    Next I
End Sub
```

These two arrays are key to the MDI NotePad application. Each time the user requests a new form, the application increases the size of both arrays by one element.

For More Information For information on the **ReDim** and **Type** statements, see Chapter 7, "Variables, Constants, and Data Types"; see the *Language Reference*; or search Help for *ReDim* and *Type*.

The following procedure creates and displays a new instance of Form1 by increasing the size of the Document array. The FState array is also increased, keeping these two arrays in parallel. Each time a new form is dimensioned, the array index for that form is stored in the form's Tag property. By doing this, you can associate data in each element of the FState array with a specific instance of a form. Each form's Tag property determines which element of the FState array is associated with that form.

```
Sub MDIForm_Load ()
    ' Create the first element of both arrays.
    ReDim FState(0)
    ' Dimensioning this array creates an instance of Form1.
    ReDim Document(0)
    ' Store the index to the array in the form's Tag property.
    Document(0).Tag = 0
    ' Show the new form.
    Document(0).Show
End Sub

Sub mnuFileNew_Click ()
    Dim ArrayCount As Integer

    ' Get the number of the array's last element. This line results in
    ' an error if the array does not contain at least one element. This
    ' example avoids the problem by dimensioning the first element of
    ' the array in the Load event procedure for the MDI form.
    ArrayCount = UBOUND(Document)

    ' Increase the FState array by one element.
    ReDim Preserve FState(ArrayCount + 1)

    ' Increase the Document array by one element. Since ReDim Preserve
    ' is used, the existing elements remain unchanged. Increasing the
    ' array size by one creates a new instance of Form1.
    ReDim Preserve Document(ArrayCount + 1)

    ' Store the current index to the array in the form's Tag property.
    Document(ArrayCount + 1).Tag = ArrayCount + 1

    ' Show the new form.
    Document(ArrayCount + 1).Show
End Sub

Sub Text1_Change ()
    ' Set the Dirty flag to True to indicate that the contents
    ' of Text1 should be saved.
    FState(Me.Tag).Dirty = True
End Sub
```

```
Sub mnuFileSave ()
    ' Save the contents of Text1 in a file.
    FileSave
    ' Set the Dirty flag to False.
    FState(Me.Tag).Dirty = False
End Sub
```

Once you have set up parallel arrays such as this, you can keep track of all sorts of information. The user-defined type that was created for the MDI NotePad sample application maintains several variables for each child form. You can add to and modify this type to store whatever information is appropriate for your application.

Note The event procedures just outlined are intended primarily for conceptual purposes. The sample application, MDINOTE.MAK, uses different code to perform the same task as the previous example, but it is more efficient and more complicated.

Unloading MDI Forms with QueryUnload

The `Dirty` flag becomes useful when the user decides to exit the application. This can occur when the user chooses Close from the MDI form's Control menu, or through a menu item you provide, such as Exit on the File menu. If the user closes the application using the MDI form's Control menu, Visual Basic will attempt to unload the MDI form.

When an MDI form is unloaded, the QueryUnload event is invoked first for the MDI form and then for every child form that is open. If none of the code in these event procedures cancels the Unload event, then each child is unloaded and finally, the MDI form is unloaded.

Since the QueryUnload event is invoked before a form is unloaded, you can give the user the opportunity to save a form before unloading it. The following MDI NotePad example uses the `Dirty` flag to determine if the user should be prompted to save the child before it is unloaded. It assumes that there is a procedure, FileSave, that saves the contents of Text1 in a file.

```
Sub mnuFileExit ()
    ' When the user chooses File Exit in an MDI application, unload
    ' the MDI form to invoke the QueryUnload event for every open child.
    Unload MDIForm1
    End
End Sub

Sub Form_QueryUnload (Cancel As Integer, UnloadMode As Integer)
    Dim Msg, NL
    Dim Response As Integer
```

```
    ' If Dirty is True, prompt the user to save the child.
    ' If Dirty is False, unload the child.
    If FState(Me.Tag).Dirty Then
        NL = Chr$(10) & Chr$(13)
        Msg = "The text in this file has changed."
        Msg = Msg & NL
        Msg = Msg & "Do you want to save the changes?"
        Response = MsgBox(Msg, 51, "MDI NotePad")
        Select Case Response
        ' User chooses Cancel; do not unload the form.
        Case 2
            Cancel = True
        ' User chooses Yes; save the child form.
        Case 6
            ' Invoke the Save procedure.
            FileSave
        ' User chooses not to save, but it is OK to unload the form.
        Case 7
            Cancel = False
        End Select
    End If
End Sub
```

For More Information For information on the QueryUnload event, see the *Language Reference*, or search Help for *QueryUnload.*

Menus in MDI Applications

In an MDI application, the menus for each child are displayed on the MDI form, rather than on the child forms themselves. When a child form has the focus, that child's menu (if any) replaces the MDI form's menu on the menu bar. If there are no children loaded, or if the child with the focus does not have a menu, the MDI form's menu is displayed (see Figures 14.4 and 14.5).

It is common for MDI applications to use several sets of menus. When the user opens a document, the application displays the menu associated with that type of document. Usually, a different menu is displayed when there aren't any children loaded. For example, when there are no spreadsheets open, Microsoft Excel displays only the File and Help menus. When the user opens a spreadsheet, other menus are displayed (File, Edit, Formula, Format, Data, Options, and so on). When the user opens a chart, a different set of menus is displayed (File, Edit, Gallery, Chart, and so on).

You can create menus for your Visual Basic application by adding menu controls to the MDI form and to the child forms. One way to manage the menus in your MDI application is to place the menu controls you want displayed, when no children are loaded, in the MDI form. When you run the application, the MDI form's menu is automatically displayed when there are no child forms loaded, as shown in Figure 14.4.

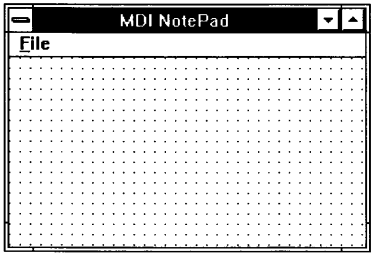


Figure 14.4 The MDI form menu is displayed when no child forms are loaded.

Place the menu controls that apply to a child form on the child form. The MDI NotePad example uses a single child form that has three menu titles (File, Edit, and Window). At run time, new instances of that form are created when the user chooses Open or New from the File menu. Since each instance of the child form has the same menu controls, these menu titles are displayed in the MDI form's menu bar, as long as there is at least one child form open (see Figure 14.5).

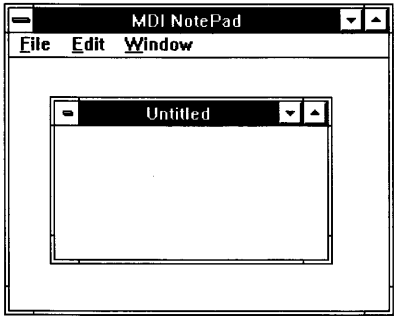


Figure 14.5 When a child form has the focus, its menu is displayed in the MDI form menu bar.

Some applications support more than one type of document. For example, in Microsoft Excel, you can open a spreadsheet or a chart. To create an application like this in Visual Basic, create two child forms. Design one child with menus that perform spreadsheet tasks and the other with menus that perform charting tasks.

At run time, when an instance of a spreadsheet form has the focus, the spreadsheet menu is displayed, and when the user selects a chart, that form's menu is displayed. If all the spreadsheets and charts are closed, the MDI form's menu is displayed. For more information on creating menus, see Chapter 4, "Menus and Dialogs."

Creating a Window Menu

Most MDI applications (for example, the Windows Program Manager and Microsoft Excel) incorporate a *Window menu.* This is a special menu that displays the captions of all open child forms, as shown in Figure 14.6. In addition, some applications place commands on this menu that manipulate the child windows, such as Cascade, Tile, and Arrange Icons.

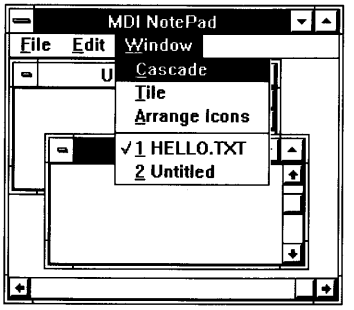


Figure 14.6 The Window menu displays the name of each open child form.

Any menu control on an MDI form or MDI child can be used to display the list of open child forms by setting the WindowList property for that menu control to **True**. At run time, Visual Basic automatically manages and displays the list of captions and displays a check mark next to the one that had the focus most recently. In addition, a separator bar is automatically placed above the list of windows.

▶ To set the WindowList property

1. Select the form where you want the menu to appear, and from the Window menu, choose Menu Design.
2. In the lower portion of the Menu Design window, select the menu where you want the list of open child forms displayed.
3. Set the Window List check box.

At run time, this menu displays the list of open child forms. In addition, the WindowList property for this menu control returns as **True**.

The WindowList property applies only to MDI forms and MDI child forms. It has no effect on standard (non-MDI) forms.

For More Information For information on the WindowList property, see the *Language Reference*, or search Help for *WindowList*.

Arranging Child Forms

As was mentioned earlier, some applications list actions such as Tile, Cascade, and Arrange Icons on a menu, along with the list of open child forms. Use the **Arrange** method to rearrange child forms in the MDI form. You can display cascading child forms or horizontally tiled child forms, or you can arrange the child form icons in an orderly fashion along the lower portion of the MDI form (see Figures 14.7, 14.8, and 14.9, respectively). The following example shows the Click event procedures for the Cascade, Tile, and Arrange Icons menu controls.

```
Sub mnuCascade_Click ()
    ' Cascade child forms.
    frmMDI.Arrange CASCADE
End Sub

Sub mnuTile_Click ()
    ' Tile child forms (horizontal).
    frmMDI.Arrange TILE_HORIZONTAL
End Sub

Sub mnuArrangeIcons_Click ()
    ' Arrange all iconized child forms.
    frmMDI.Arrange ARRANGE_ICONS
End Sub
```

Note The constants CASCADE, TILE_HORIZONTAL, and ARRANGE_ICONS, along with other constants used in this chapter, are contained in the CONSTANT.TXT file. You can use these constants in your code if you load this file into any of your application's code modules.

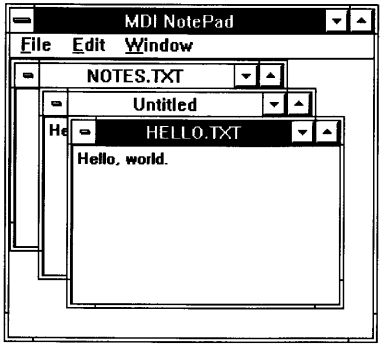


Figure 14.7 Cascading child forms in an MDI application

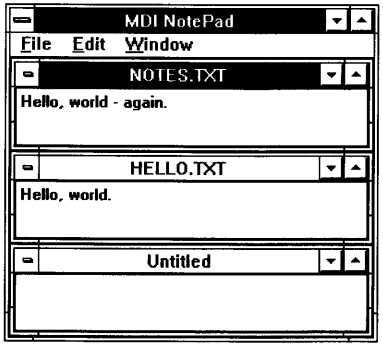


Figure 14.8 Horizontally tiled child forms in an MDI application

Note The TILE_VERTICAL constant affects only applications running under Windows 3.1. When using Windows 3.0, this constant has the same effect as TILE_HORIZONTAL.

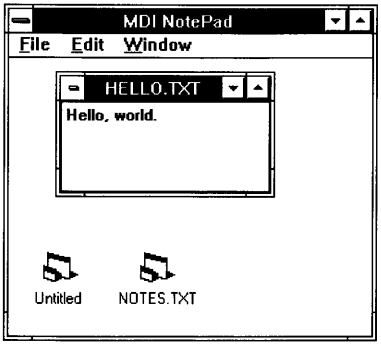


Figure 14.9 The Arrange Icons menu control displays all minimized child forms in an orderly fashion along the lower portion of the MDI form.

Note When you tile or cascade child forms that have a fixed border style, each child form is positioned as if it had a sizable border. This can cause child forms to overlap.

Creating a Toolbar

The *toolbar* (also called a ribbon or control bar) has become a standard feature in many Windows–based applications. A toolbar provides quick access to the most frequently used menu items and commands in an application.

▶ **To create a toolbar**

1. Place a picture box on the MDI form.

   The width of the picture box automatically stretches to fill the width of the MDI form's client area. The *client area* is the area *inside a form's borders*, not including the title bar, menu bar, or any toolbars, status bars, or scroll bars that may be on the form. When a form does not contain any toolbars or status bars, its client area and internal area are the same.

   Note You can place only those controls that support the Align property directly on an MDI form (the picture box is the only standard control that supports this property).

2. Inside the picture box, place any controls you want displayed on the toolbar.

   Typically, you create buttons for the toolbar using command buttons or image controls. Figure 14.10 shows a toolbar containing image controls.

   Note When an MDI form contains a picture box, the internal area of the MDI form does not include the area of the picture box. For example, when an MDI form contains a picture box, the ScaleHeight property of the MDI form returns the internal height of the MDI form, which does not include the height of the picture box.

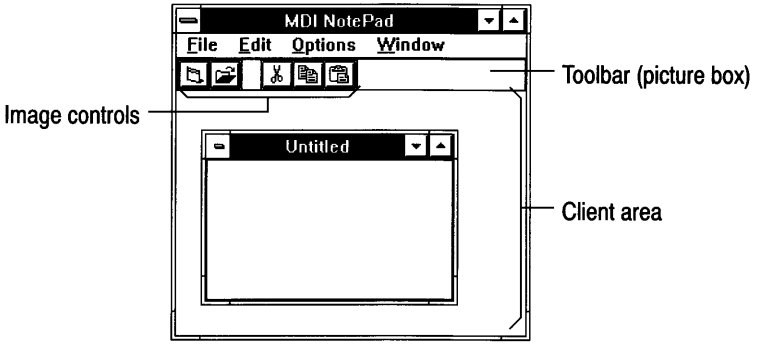


Figure 14.10 You can create buttons for the toolbar using image controls.

3. Set design-time properties.

   One advantage of using a toolbar is that you can present the user with a graphical representation of a command. The image control is a good choice as a toolbar button because you can use it to display a bitmap. Set its Picture property at design time to display a bitmap that provides the user with a visual cue of the command performed when the button is chosen.

4. Write code.

   Since toolbar buttons are usually used to provide easy access to other commands, most of the time you call other procedures from within each button's Click event.

Tip You can use controls that are invisible at run time (such as the timer control) with an MDI form without displaying a toolbar. To do this, place a picture box on the MDI form, place the control in the picture box, and set the picture box's Visible property to **False**.

Toolbars are used to provide the user with a quick way to access some of the application's commands. Since it is not possible to call another form's event procedure, organize your code in well-defined procedures in a separate module so that it is easily accessible from other form and code modules.

For example, the first button on the toolbar in Figure 14.10 is a shortcut for the File New command. There are now three places in the MDI NotePad application where the user can request a new file:

- On the MDI form (File New on the MDI form menu)
- On the child form (File New on the child form menu)
- On the toolbar (File New button)

Rather than duplicate this code three times, you can take the original code from the child's mnuFileNew_Click event and place it in a **Sub** procedure in a code module. You can call this **Sub** procedure from any of the preceding event procedures. Here's an example:

```
' This routine is placed in a code module.
Sub FileNew ()
    Dim frmNewPad As New frmNotePad
    frmNewPad.Show
End Sub

' The user chooses File New on the child form.
Sub mnuchildFileNew_Click ()
    FileNew
End Sub

' The user chooses File New on the MDI form.
Sub mnumdiFileNew_Click ()
    FileNew
End Sub

' The user chooses the File New button on the toolbar.
Sub btnFileNew_Click ()
    FileNew
End Sub
```

Note You can use a similar technique to create a status bar. A *status bar* typically appears at the bottom of an MDI form and displays information about the currently selected item or the state of the application. By setting the Align property of a picture box to 2-Align Bottom, you can make that picture box appear at the bottom of an MDI form.

Specifying the Active Child Form or Control

Sometimes you want to provide a command that operates on the control with the focus in the currently active child form. For example, suppose you want to create a button that copies selected text to the Clipboard. First, you place a picture control on the MDI form. Then you place an image control inside the picture box and set its Picture property so that it displays an appropriate bitmap. In the image control's Click event, you call EditCopyProc, a procedure that you've placed in a code module that copies selected text to the Clipboard.

Since the application can have many instances of the same child form, EditCopyProc needs to know which form to act on. To do this, you use the ActiveForm property of the MDI form. The ActiveForm property returns the child form that has the focus or was most recently active.

Note At least one MDI child form must be loaded when you access the ActiveForm property, or an error is returned.

Here's an example of a copy routine that can be called from a child menu, a menu on the MDI form, or a toolbar button:

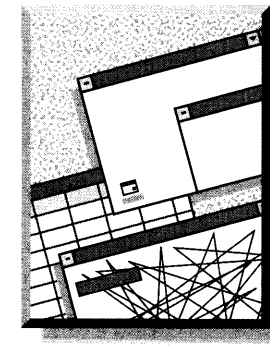
```
Sub EditCopyProc ()
    ' Copy selected text to Clipboard.
    ClipBoard.SetText frmMDI.ActiveForm.Text1.SelText
End Sub
```

Note, however, that this example assumes there is a control named Text1 on the active form, and that this is the only control from which the user would ever want to copy. When you can't make this assumption, you can use the ActiveControl property. Similar to the ActiveForm property, the ActiveControl property returns the control with the focus on the active child form. This copy routine allows the user to copy text from any text box on a child form:

```
Sub EditCopyProc ()
    ' If the active control is a text box...
    If TypeOf frmMDI.ActiveForm.ActiveControl is TextBox Then
    ' ...copy selected text to Clipboard.
        ClipBoard.SetText frmMDI.ActiveForm.ActiveControl.SelText
    End If
End Sub
```

CHAPTER 15

Creating Graphics for Applications



Graphics add style, interest, and visual structure to the interface of an application. Visual Basic gives you a variety of ways to create and use graphics in an application. This chapter discusses the methods, controls, and techniques you use to create graphics.

Contents

- The Fundamentals of Graphics
- Using Graphical Controls
- Using Graphics Methods
- Using Graphics Properties

BLANKER.MAK

Many of the code examples in this chapter are taken from the BLANKER.MAK sample application. If you installed the sample applications, you will find this application in the \GRAPHICS subdirectory of the main Visual Basic directory (\VB\SAMPLES\GRAPHICS).

The Fundamentals of Graphics

This chapter discusses the two approaches to creating graphics for an application using:

- Graphical controls
- Graphics methods

You can use controls and methods in Visual Basic to draw points, lines, boxes, circles, and other shapes and then change their location and appearance on a form. Though creating graphics using graphics methods is different than using graphical controls, there are basic ideas that apply to creating graphics in Visual Basic. This section discusses the fundamental concepts that apply to all graphics.

Using Twips

By default, all Visual Basic movement, sizing, and graphical-drawing statements use a unit of one twip. A *twip* is 1/20 of a printer's point (1,440 twips equal an inch, and 567 twips equal a centimeter). These measurements apply to how large an object will be when printed. Actual physical distances on the screen vary according to the monitor size.

Later in this chapter, the section "Changing an Object's Coordinate System" describes how to select units other than twips. For now, just use the default scale.

Using the Coordinate System

Every graphical operation described in this chapter (including resizing, moving, and drawing) uses the coordinate system of the drawing area or container. Though you can use the coordinate system to achieve graphical effects, the most important thing to know at this point is how you use the coordinate system to define the location of forms and controls in your application.

The Coordinate System

The coordinate system is a two-dimensional grid that defines locations on the screen, in a form, or other container (such as a picture box or Printer object). You define locations on this grid using coordinates in the form:

(*x*, *y*)

The value of *x* is the location of the point along the x-axis with the default location of 0 at the extreme left. The value of y is the location of the point along the y-axis with the default location of 0 at the extreme top. This coordinate system is illustrated in Figure 15.1.

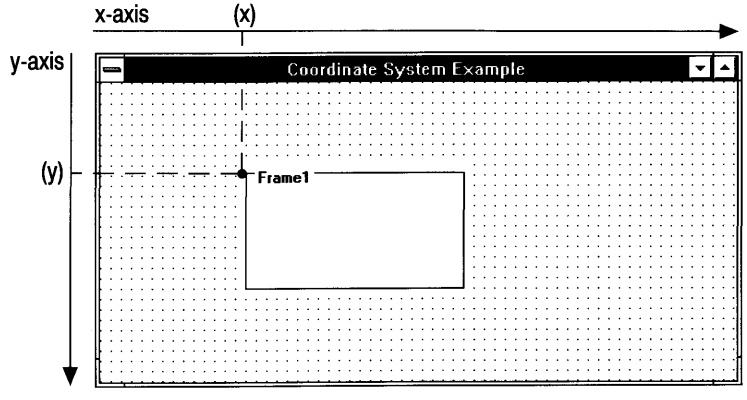


Figure 15.1 The coordinate system of a form

The following rules apply to the Visual Basic coordinate system:

- When you move or resize a control, the coordinate system you use is that of the control's container. If you draw the object directly on the form, the form is the container. If you draw the control inside a frame or picture box, the frame or the control is the container.
- All graphics and **Print** methods use the coordinate system of the container. For example, statements that draw inside a picture box use the coordinate system of that control.
- Statements that resize or move a form always express the form's position and size in twips.

  When you create code to resize or move a form, you should first check the Height and Width properties of the Screen object to make sure the form will fit on the screen.
- The upper-left corner of the screen is always (0, 0). The default coordinate system for any container starts with the (0, 0) coordinate in the upper-left corner of the container.

The units of measure used to define locations along these axes are collectively called the *scale*. In Visual Basic, each axis in the coordinate system can have its own scale.

You can change the direction of the axis, the starting point, and the scale of the coordinate system, but use the default system for now. The section "Changing an Object's Coordinate System," later in this chapter, discusses how to make these changes.

Using Simple Color

Visual Basic uses a consistent system for all color properties and graphics methods. A color is represented by a long integer, and this value has the same meaning in all contexts that specify a color.

Specifying Colors at Run Time

There are four ways to specify a color value at run time:

- Use the **RGB** function.
- Use the **QBColor** function to choose one of sixteen Microsoft QuickBasic™ colors.
- Use a previously defined constant, such as one of the values in the file CONSTANT.TXT.
- Enter a color value directly.

This section discusses how to use the **RGB** and **QBColor** functions as simple ways to specify color. See the section "Using Color Properties" for information on using constants to define color or directly entering color values.

Using the RGB Function

You can use the **RGB** function to specify any color.

▶ **To use the RGB function to specify a color**

1. Assign each of the three primary colors (red, green, and blue) a number from 0 to 255, 0 denoting the least intensity and 255 the most.
2. Give these three numbers as input to the **RGB** function, using the order red-green-blue.
3. Assign the result to the color property or color argument.

Every visible color can be produced by combining one or more of the three primary colors. For example:

```
Form1.BackColor = RGB(0, 128, 0)          ' Set background to green.
Form2.BackColor = RGB(255, 255, 0)        ' Set background to yellow.
PSet (100, 100), RGB(0, 0, 64)            ' Set point to dark blue.
```

For More Information For information on the **RGB** function, see the *Language Reference*, or search Help for *RGB*.

Using the QBColor Function

If you're accustomed to QuickBasic color values, you can use them by calling the **QBColor** function. This function takes a single number that specifies a QuickBasic color number from 0 to 15, and returns a long integer that can be used in any Visual Basic color argument or color property setting. For example:

```
Form1.BackColor = QBColor(5)
```

Using the **QBColor** function is a convenient way to define the values of colors that change during run time. Note that the *qbcolor* argument corresponds to the color values used by the graphics statements in other versions of Basic, such as Visual Basic for MS-DOS, Microsoft QuickBasic, and Microsoft Basic Professional Development System.

For More Information For information on the **QBColor** function, see the *Language Reference*, or search Help for *QBColor*. For information on using color in your applications, search Help for *Color palette*.

Using Graphical Controls

Visual Basic provides three controls designed to create graphical effects in an application. These controls are:

- The image control
- The line control
- The shape control

The icons for these controls in the Toolbox are shown in Figure 15.2.

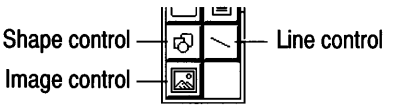


Figure 15.2 Graphical controls in the Toolbox

Advantages of Graphical Controls

The image, line, and shape controls are very useful for creating graphics at design time. One advantage is that graphical controls require fewer system resources than other Visual Basic controls. This difference improves the performance of your Visual Basic application.

Another advantage to graphical controls is that you can create graphics with less code than with graphics methods. For example, you can use either the **Circle** method or the shape control to place a circle on a form. The **Circle** method requires you to create the circle with code at run time, while you can simply draw the shape control on the form and set the appropriate properties at design time.

Limitations of Graphical Controls

While graphical controls are designed to maximize performance with minimal demands on the application, they accomplish this goal by limiting other features common to controls in Visual Basic. Graphical controls:

- Cannot appear on top of other controls, unless they are inside a container that can appear on top of other controls (such as a picture box).
- Cannot receive focus at run time.
- Cannot serve as containers for other controls.
- Do not have an hWnd property.

Image Control

An *image control* is a rectangular area into which you can load picture files. Visual Basic can read picture files in any of the following standard formats.

Picture format	Description
Bitmap	A *bitmap* defines an image as a pattern of dots (pixels). A bitmap has the file extensions .BMP or .DIB. Also called "paint-type" graphics.
Icon	An *icon* is a special kind of bitmap. Icons have a maximum size of 32 pixels by 32 pixels. An icon has the file extension .ICO.
Metafile	A *metafile* defines an image as coded lines and shapes. A metafile has the file extension .WMF. Only files that are compatible with Microsoft Windows version 3.0 or later can be loaded. Also called "draw-type" graphics.

An image control is very similar to a picture box. However, you can stretch pictures in an image control to fit the control's size. You cannot do this with a picture box. Figure 15.3 shows an image control that displays a bitmap of an arch.

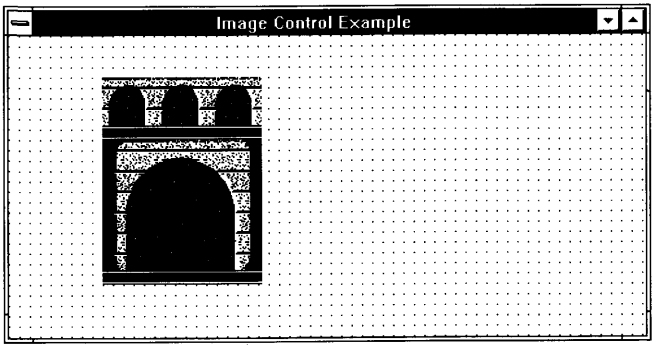


Figure 15.3 A bitmap in an image control

For More Information For information about the properties, events, and methods of the image control, search Help for *image controls*.

Line Control

A *line control* is a straight-line segment that you draw at design time. You can control the position, length, color, and style of line controls to customize the look of applications. Figure 15.4 shows a line control used to graphically separate the label containing the text "Company Name" from the rest of the form.

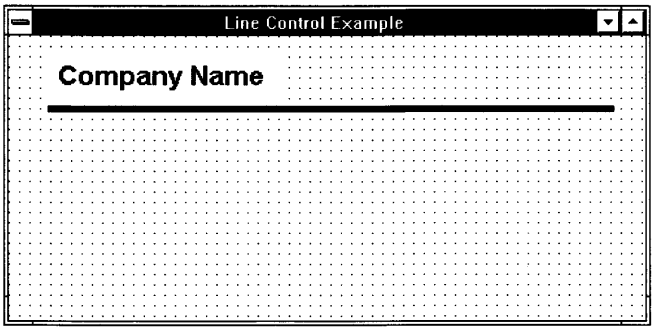


Figure 15.4 A line control

For More Information For information about the properties and methods of the line control, search Help for *line control.*

Shape Control

A *shape control* is a visual element that can assume one of several predefined shapes. These shapes are:

- Rectangle (with square or rounded corners)
- Square (with square or rounded corners)
- Oval
- Circle

In addition to shape, you can control the color, transparency, fill and outline colors, and size of a shape control.

Figure 15.5 shows all the predefined shapes used by the shape control.

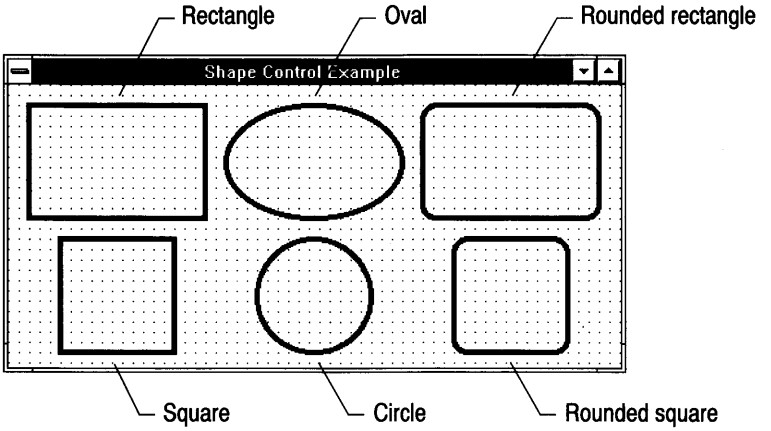


Figure 15.5 Shape controls

For More Information For information about the properties and methods of the shape control, search Help for *shape control.*

Adding Pictures to Your Forms

Pictures can be displayed in three places in Visual Basic applications:

- On a form
- In a picture box
- In an image control

Pictures can come from Microsoft PC Paintbrush® (included with Windows versions 3.0 and 3.1), other graphics programs, or clip-art libraries. Visual Basic provides a large collection of icons you can use as graphics in applications.

You use different techniques for adding a picture to a form, a picture box, or an image control depending on whether you do it at design time or run time.

Adding a Picture at Design Time

There are two ways to add a picture at design time:

- Load a picture onto a form, or into a picture box or image control from a picture file:

  In the Properties window, select Picture from the Properties list and click the three dots at the right of the Settings box. Visual Basic displays a dialog box, from which you select a picture file.

  If you set the Picture property for a form, the picture you select is displayed on the form, behind any controls you've placed on it. Likewise, if you set the Picture property for a picture box, the picture is displayed in the box, behind any controls you've placed on it.

- Paste a picture onto a form or into a picture box or image control:

  Copy a picture from another application (such as Paintbrush) onto the Clipboard. Return to Visual Basic, select the form, picture box or image control, and choose Paste from the Edit menu.

Once you've set the Picture property for a form, picture box or image control—either by loading a picture or pasting one—the word displayed in the Settings box is "(Bitmap)," "(Icon)," or "(Metafile)." To change the setting, load or paste another picture. To set the Picture property to "(None)" again, double-click the word displayed in the Settings box and press the DEL key.

Adding a Picture at Run Time

There are three ways to add a picture at run time:

- Use the **LoadPicture** function to specify a file name and assign the picture to the Picture property.

  The following statement loads the file CARS.BMP into a picture box named picDisplay (you name a control by setting its Name property):

  ```
  picDisplay.Picture = LoadPicture("C:\PICTS\CARS.BMP")
  ```

  You can load a new picture file onto a form or into a picture box or image control whenever you want. Loading a new picture completely replaces the existing picture, although the source files of the pictures are never affected.

- Copy a picture from one object to another.

  Once a picture is loaded or pasted onto a form or into a picture box or image control, you can assign it to other forms, picture boxes, or image controls at run time. For example, this statement copies a picture from a picture box named picDisplay to an image control named imgDisplay:

  ```
  imgDisplay.Picture = picDisplay.Picture
  ```

- Copy a picture from the Clipboard object. For more information, see Chapter 17, "Interacting with the Environment"; the *Language Reference*; or search Help for *Clipboard*.

Note If you load or paste pictures from files at design time, the pictures are saved and loaded with the form. When you create an .EXE file, you don't need to give your users copies of the picture files; the .EXE file itself contains the images. In contrast, to load pictures at run time with the **LoadPicture** function, you must supply the picture files to your users along with your application. For this reason, it's often better to load or paste pictures at design time. Your application can then copy pictures from one object to another, as described in the preceding text.

Removing a Picture at Run Time

You can also use the **LoadPicture** function to remove a picture at run time without replacing it with another picture. The following statement removes a picture from an image control named imgDisplay:

```
imgDisplay.Picture = LoadPicture("")
```

For More Information See the Butterfly sample application in Chapter 2, "Your First Visual Basic Application," for an example of loading pictures at run time.

Moving and Sizing Pictures

If a form, picture box, or image control is moved (at design time or run time), its picture automatically moves with it. If a form, picture box, or image control is resized so it's too small to display a picture, the picture gets clipped at the right and bottom. A picture is also clipped if you load or copy it onto a form or into a picture box or image control that is too small to display all of it.

AutoSize Property

If you want a picture box to automatically expand to accommodate a new picture, set the AutoSize property for the picture box to **True**. Then when a picture is loaded or copied into the picture box at run time, Visual Basic automatically expands the control down and to the right enough to display all of the picture. If the image you load is larger than the edges of the form, it appears clipped because the form size doesn't change.

You can also use the AutoSize property to automatically shrink a picture box to reflect the size of a new picture.

Note Image controls do not have an AutoSize property, but automatically size themselves to fit the picture loaded into them. Forms don't have an AutoSize property, and they do not automatically enlarge to display all of a picture.

Stretch Property of Image Controls

If you want a picture in an image control to automatically expand to fit a particular size, use the Stretch property. When the Stretch property is **False**, the image control automatically adjusts its size to fit the picture loaded into it. To resize the picture to fit the image control, set the Stretch property for the image control to **True**.

Selecting Art for the Picture Control

Where do you get picture files? If you want icons, you can use the Icon Library included with Visual Basic, or create them yourself with programs such as the IconWorks sample application. You can create .BMP files with Microsoft® PC Paintbrush®. Or you can buy a clip-art collection that includes bitmap or icon files, or metafiles.

The Visual Basic Icon Library includes a set of icons organized into these groups:

- Arrows and pointers
- Communication (networks, phones, printers, servers)
- Computer (computers, disks, keyboards, mouse devices, MDI forms)
- Drag-and-drop (hands and arrows dragging and dropping icons)
- Elements (earth, clouds, water, fire, phases of the moon)
- Flags (international flags)
- Industry and transportation (bicycles, cars, factories, airplanes, tools)
- Mail (mailboxes, envelopes, stamps, forms)
- Office (card files, charts, file cabinets, folders, graphs, labels, rulers, paper clips)
- Traffic signs
- Writing (books, notepads, pens, pencils)
- Miscellaneous (binoculars, houses, light bulbs, first aid signs, people, locks and keys, punctuation marks)

To see all of the pictures in the Icon Library, refer to Appendix B, "Icon Library," or load the IconWorks sample application included with Visual Basic.

Moving Controls Dynamically

With Visual Basic, one of the easiest effects to achieve is moving a control at run time. You can either directly change the properties that define the position of a control or use the **Move** method. This section describes how to move controls at run time.

Using the Left and Top Properties

The Left property is the distance between the upper-left corner of the control and the left side of the form. The Top property is the distance between the upper-left corner of the control and the top of the form. Figure 15.6 shows the Left and Top properties of a control.

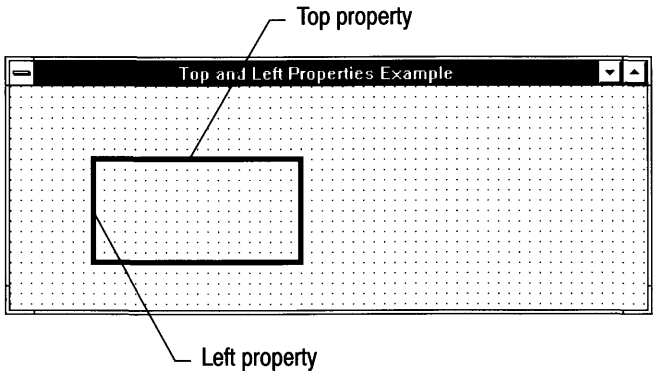


Figure 15.6 The Left and Top properties

You can move a control by changing the settings of its Left and Top properties with statements such as these:

```
txtField1.Left = txtField1.Left + 200
txtField1.Top = txtField1.Top - 300
```

Moving a Line Control

As mentioned previously, line controls don't have Left or Top properties. Instead, you use special properties to control the position of line controls on a form. The following table lists these properties and how they determine the position of a line control.

Property	Description
X1	The x-coordinate of the start of the line. The coordinate is given in current scale units. The start of the line is the end created when you start drawing.
Y1	The y-coordinate of the start of the line.
X2	The x-coordinate of the end of the line. The end of the line is the end created when you stop drawing.
Y2	The y-coordinate of the end of the line.

The Jumpy Line demo of the Blanker application randomly changes the position of a line control on the DemoForm using these statements:

```
linLineCtl.X1 = Int(DemoForm.Width * Rnd)       ' Set random X position for 1st line end.
linLineCtl.Y1 = Int(DemoForm.Height * Rnd)      ' Set random Y position for 1st line end.
linLineCtl.X2 = Int(DemoForm.Width * Rnd)       ' Set random X position for 2nd line end.
linLineCtl.Y2 = Int(DemoForm.Height * Rnd)      ' Set random Y position for 2nd line end.
Cls                                             ' Clear stray pixels from moving line.
Delay                                           ' Pause display briefly before next move.
```

Using the Move Method

Changing the Left and Top or X and Y properties produces a jerky effect as the control first moves horizontally and then vertically. The **Move** method produces a smoother diagonal movement.

The syntax for the **Move** method is:

[*object.*]**Move** *left* [, *top*[, *width*[, *height*]]]

The *object* is the form or control to be moved. If *object* is omitted, the current form moves. The arguments *left* and *top* are the new settings for the Left and Top properties of *object*, while *width* and *height* are new settings for its Width and Height properties. Only *left* is required, but to specify other arguments, you must include all arguments that appear in the argument list before the argument you want to specify.

Absolute Movement

Absolute movement occurs when you move an object to specific coordinates in its container. The following statement moves a control named txtField1 using absolute movement to the coordinates (100, 200):

```
txtField1.Move 100, 200
```

Relative Movement

Relative movement occurs when you move an object by specifying the distance it should move from its current position. The following statement moves txtField1 using relative movement to a position 100 twips down and to the right of its current position:

```
txtField1.Move txtField1.Left + 100, txtField1.Top + 100
```

Example

This section shows control movement in the Blanker sample application. The Rebound demo moves a picture box diagonally around the form, so the picture box appears to "bounce" off the sides of the form. This demo uses a picture box instead of an image control because the image control flickers as the movement causes it to repaint.

Figure 15.7 shows the main form of the Blanker application (DemoForm) and the picture box used in this example.

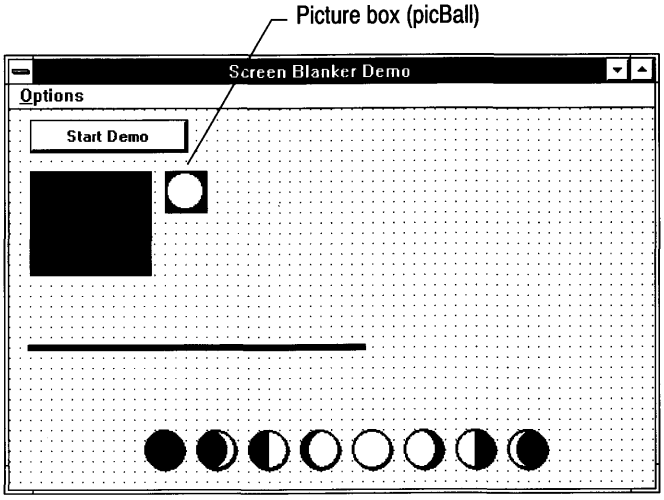


Figure 15.7 Picture box (picBall) in the Blanker application

The name of the picture box is picBall. This control begins moving around the form after you choose the Rebound command from the Options menu and then click the command button. The event procedure for this command button then calls the CtlMoveDemo procedure.

The CtlMoveDemo procedure randomly selects a starting direction from one of these four possibilities:

- Left and up
- Right and up
- Left and down
- Right and down

The picBall picture box moves along the chosen direction until the control reaches one of the four edges of the form. Then the picture box changes direction away from the edge it has reached; the variable `Motion` controls the direction. For example, when the picture box is moving left and up, this portion of the procedure changes the value of `Motion` and directs the code to move picBall in another direction.

The following statements come from the CtlMoveDemo procedure in the Blanker application:

```
Select Case Motion
Case 1                                                    ' If motion is left and up...
    picBall.Move picBall.Left - 20, picBall.Top - 20      ' move the control 20 twips.
    If picBall.Left <= 0 Then                             ' If control touches left edge...
        Motion = 2                                        ' change motion to right and up.
    ElseIf picBall.Top <= 0 Then                          ' If control touches top edge...
        Motion = 4                                        ' change motion to left and down.
```

Notice that the line of code that moves picBall subtracts 20 twips from the current values of its Left and Top properties to establish the new location of the control. This ensures that the control always moves relative to its current position.

The speed and smoothness of the control's movement depend on the number of twips (or other units) used in the **Move** method. Increasing the number of twips increases the speed but decreases the smoothness of motion. Decreasing the number of twips decreases the speed but improves the smoothness of the control's motion.

For More Information For information on the **Move** method, see the *Language Reference*, or search Help for *Move*.

Creating Simple Animation

You can create simple animation by changing pictures at run time. The easiest way to do this is to toggle between two images. You can also use a series of pictures to create animation with several frames. By also moving the picture dynamically, you can create more elaborate effects.

Toggling Between Two Pictures

Some of the icons in the Icon Library can be used in pairs. For instance, there are two matching envelope icons in the Icon Library, one with the envelope unopened and one with the envelope torn open, as shown in Figure 15.8. By switching between the two (also called "toggling"), you can create an animation that shows your user the status of mail.

Note Although this example uses icons, you can also use metafiles and bitmaps.

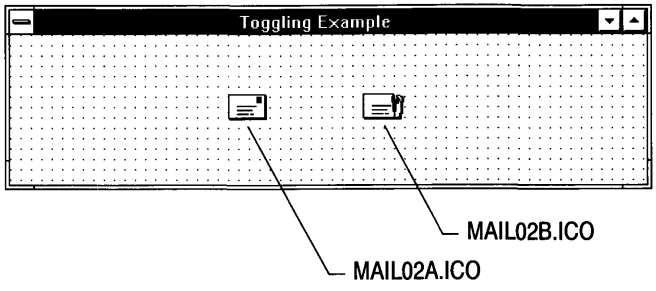


Figure 15.8 Mail icons from the Icon Library

The following statement changes the Picture property of an image control named imgMailStatus to toggle its picture from an unopened envelope to an open envelope:

```
imgMailStatus.Picture = imgMailOpen.Picture
```

Rotating Through Several Pictures

You can also rotate through several pictures to make longer animations. This technique is basically the same as toggling between two pictures. Using several pictures requires the application to select which bitmap acts as the current image. One way to control the individual pictures in an animation is with a control array.

A *control array* is a group of controls that share the same name and type. They also share the same event procedures, but they are physically separate, and each control has its own property settings. See Chapter 4, "Menus and Dialogs," for more information about control arrays.

Example

The Blanker sample application includes an animation that shows a rotating moon. The Spinning Moon demo uses an array of eight image controls to create the animation. The array of image controls is shown in Figure 15.9.

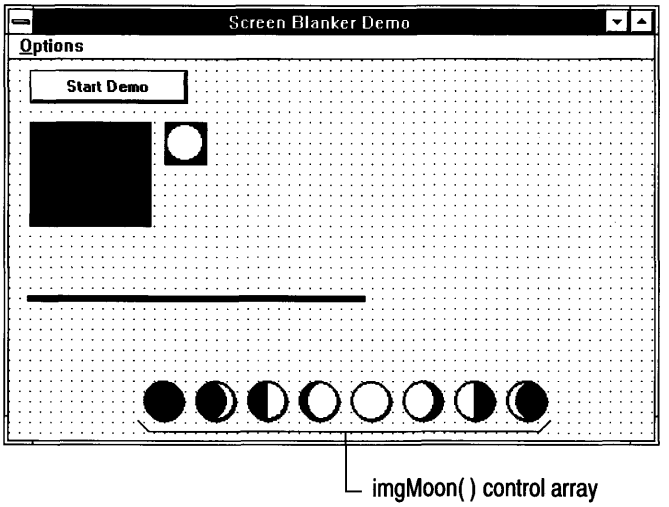


Figure 15.9 An array of image controls in the Blanker application

This example uses an image control to benefit from its faster screen painting. The flicker of the image control as it moves across the form is hidden by changing the Picture property of the control to create animation. The name of the control array is imgMoon(). The moon in the Spinning Moon animation spins while moving around the form. After you choose the command in the Options menu, the event procedure for the StartDemo command button calls the ImageDemo event procedure.

The ImageDemo procedure uses essentially the same code as the CtlMoveDemo procedure to bounce the spinning moon around the form.

```
Select Case Motion
Case 1
    ' If motion is up and left, move 100 twips diagonally.
    imgMoon(0).Move imgMoon(0).Left - 100, imgMoon(0).Top - 100
    IncrFrame                                          ' Show next frame.
    If imgMoon(0).Left <= 0 Then                       ' If moon reaches left edge...
        Motion = 2                                     ' go up and right.
    ElseIf imgMoon(0).Top <= 0 Then                    ' If moon reaches top edge...
        Motion = 4                                     ' go down and left.
```

The control being moved is imgMoon(0), the first image control in the control array. After moving the control by 100 twips, the code calls the IncrFrame procedure. This procedure changes the Picture property of imgMoon(0) to show the current frame.

```
FrameNum = FrameNum + 1                          ' Get the next frame number.
If FrameNum > 7 Then                             ' If the next frame is the last frame,
    FrameNum = 0                                 ' reset moon to the first frame.
End If
imgMoon(0).Picture = imgMoon(FrameNum).Picture   ' Change moon to show the frame.
```

The procedure increments `FrameNum`, a variable used to keep track of the current frame in the animation. Because there are only eight image controls in the control array, the procedure continuously loops the animation by resetting the value of `FrameNum` to 0 when it reaches 8.

Therefore, `FrameNum` has a value from 0 to 7, matching the Index properties of the controls in the imgMoon() control array. The Picture property of imgMoon(0) then changes to match that of the appropriate control in the array, so each time the ImageDemo procedure moves imgMoon(0), it uses IncrFrame to rotate through the pictures in the control array to create animation.

Resizing Controls Dynamically

In a Visual Basic application, you can change the size and shape of a picture box, image control, or form at run time, just as you can change its position.

The following properties affect size.

Property	Description
Align	If set to align a picture box to the top (1) or bottom (2) of a container, the width of the picture box always equals the width of the inside of the container.
Height	Height of the object expressed in twips.
Width	Width of the object expressed in twips.
AutoSize	If **True**, always causes Visual Basic to adjust the picture box dimensions to the size of the contents.
Stretch	If **True**, the bitmap or metafile stretches to fit the size of the image control. If **False**, the size of the image control changes to match the size of the bitmap or metafile it contains.

Align applies to picture boxes; Height and Width apply to all forms and all controls except timers, menus, and lines. AutoSize applies only to labels and picture boxes. Stretch applies only to image controls.

In this example, a command button named cmdGrow grows larger each time the user clicks it:

```
Sub cmdGrow_Click ()
    cmdGrow.Height = cmdGrow.Height + 300
    cmdGrow.Width = cmdGrow.Width + 300
End Sub
```

Showing and Hiding Controls

Objects that appear and disappear can add interesting visual effects to your Visual Basic application. This capability is especially useful in games or simulations, in which you might need a new object to enter the screen. It is also useful for controlling the display of information that is relevant only in certain situations. This section explains how to show and hide existing controls, and the next section explains how to add new controls.

The ability to show or hide controls works only at run time. Controls are always visible at design time.

All controls except timers have a Visible property. Setting the Visible property to **False** makes the control "disappear," even though it is still loaded in memory. For example, this statement makes a text box named txtField1 disappear:

```
txtField1.Visible = False
```

The next statement makes the text box reappear by setting the Visible property to **True**. Because txtField1 was not actually deleted from memory, it automatically reappears with all its old settings (unless you've changed some of the property settings in the meantime).

```
txtField1.Visible = True
```

The Blanker sample application uses this technique to hide and show the different controls used in the various Screen Blanker demos. Figure 15.10 shows all the controls that must appear and disappear on the DemoForm at run time as demos are chosen from the Options menu.

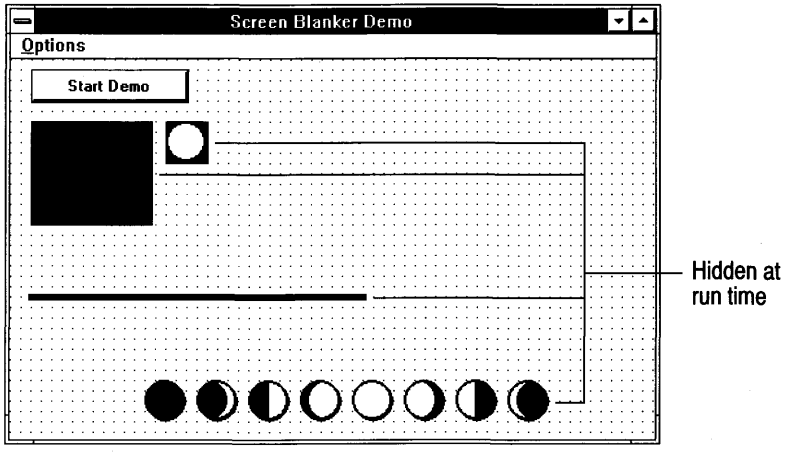


Figure 15.10 Controls hidden at run time in the Blanker application

Demos that display a control switch the Visible property of that control to **True** at run time. When the demo is stopped by moving or clicking the mouse, the event procedure hides the control by setting the Visible property to **False** again.

You can also use a single statement to hide a control if it's visible and show it if it's hidden. The statement takes advantage of the fact that the Visible property is Boolean—it's either "on" or "off"—to toggle it between the two settings:

```
Sub Command1_Click ()
    txtField1.Visible = Not txtField1.Visible
End Sub
```

You can use similar code to toggle other Boolean properties.

For More Information For information on how to show or hide forms at run time (using the **Show** and **Hide** methods), see the *Language Reference,* or search Help for *Show.*

Adding and Removing Controls

The previous section explained how to hide and show existing controls. You can also add and remove controls at run time using the **Load** and **Unload** statements. However, the control to be added must be an element of an existing control array.

Without control arrays, dynamically creating new controls is not possible. Each new control uses the common event procedures written for the array.

Note Loading and unloading controls takes longer at run time than creating all the controls you need at design time and then changing their Visible property at run time. However, if you don't know at design time how many copies of a control the application needs, loading and unloading is the best approach to use.

The following code assumes you've created a text box named txtField1 and set its Index property to 0, and defined TotalCopies as an integer greater than 0. On loading the form, the procedure loads new text boxes, places them apart vertically, and "turns on" the Visible property of each.

```
Sub Form_Load ()
    For I = 1 To TotalCopies                                    ' Make # of copies
needed.
        Load txtField1(I)                                        ' Load a copy.
        txtField1(I).Top = txtField1(I - 1).Top + 500          ' Move copy in place.
        txtField1(I).Visible = True                              ' Make copy
visible.
        txtField1(I).Text = "Field1(" + Format$(I) + ")"       ' Set text in box.
    Next I
End Sub
```

Note Visual Basic generates an error if you attempt to use the **Load** statement with an index number already in use within the array.

You can use the **Unload** statement to remove any control created with **Load**. However, **Unload** cannot be used to remove controls created at design time, whether or not they are part of a control array.

Using a Control Array

When you load a new element of a control array, all property settings except Visible, Index, and TabIndex are copied from the lowest element in the array. As a result, the code in the preceding example makes all the controls the same size and stacks them all in one place. Also, because new controls are created with the Visible property set to **False**, none of them are visible, although the text box is. New controls are created on top of each other in the container. Graphical controls exist in a layer behind the layer of nongraphical controls.

Example

The Madhouse demo of the Blanker application makes 20 copies of a shape control and then randomly changes the appearance and location of each copy. The Madhouse demo uses the **Load** method to make copies of a shape control in a control array.

The shape control is shown in Figure 15.11.

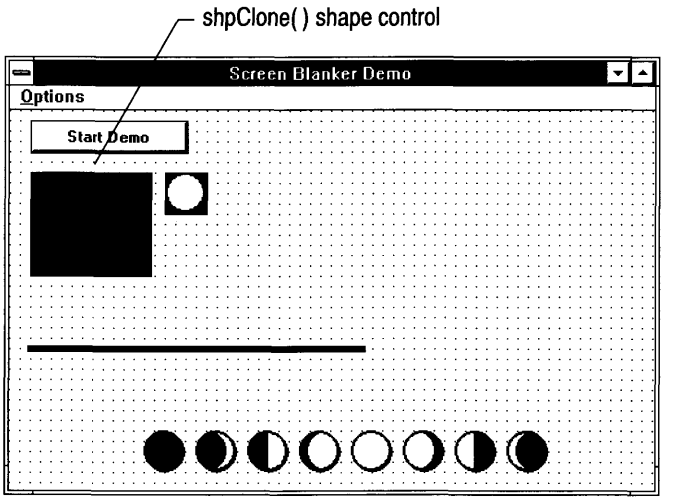


Figure 15.11 A shape control in the Blanker application

The name of the control array containing the shape control is shpClone(). At design time, there is only one shape control in the array. But if Madhouse has been chosen from the Options menu, clicking the command button calls the ShapeDemo procedure. The ShapeDemo procedure then loads 20 copies of shpClone(0) using the **Load** method as shown:

```
For MakeClone = 1 To 20              ' Number of copies = 20.
    Load shpClone(MakeClone)         ' Load a copy of shpClone(0).
Next MakeClone
```

After loading the last copy, the ShapeDemo procedure randomly selects one of the shape controls in the array. It selects the control to be changed by picking a random number from 1 to 20 and then selecting the shape control in the array with that number as its Index property. It then moves the selected control, changing its color, size, and shape as well as other aspects of its appearance.

When you stop the demo by clicking the command button on the form, Blanker unloads the copies of the shape control using these statements:

```
For UnClone = 1 To 20                ' For all 20 copies...
    Unload shpClone(UnClone)         ' unload the selected copy.
Next UnClone
```

Using Graphics Methods

In addition to the graphical controls, Visual Basic provides several methods designed to create graphics in an application. This section describes the graphics methods, summarized in the following table, that apply to forms and picture boxes.

Method	Description
Cls	Clears all graphics and **Print** output.
PSet	Sets the color of an individual pixel.
Point	Returns the color value of a specified point.
Line	Draws a line, rectangle, or filled-in box.
Circle	Draws a circle, ellipse, or arc.

Note The **Print** method can also be considered a graphics method, because its output is written to the object and is saved in the memory image (if AutoRedraw is on) just like the **PSet**, **Line**, and **Circle** methods.

Advantages of Graphics Methods

The graphics methods work well in situations where using graphical controls create too much work. For example, creating gridlines on a graph would need an array of line controls but only a small amount of code using the **Line** method. Tracking the position of line controls in an array as the form changes size is more work than simply redrawing lines with the **Line** method.

When you want a visual effect to appear only briefly on a form, like a streak of color when you display an About dialog, you can use a line or two of code for this temporary effect instead of another control.

Graphics methods offer some visual effects that are not available in the graphical controls. For example, you can create arcs or paint individual pixels only using the graphics methods. Graphics you create with these graphics methods appear on the form in a layer of their own. This layer is below all other controls on a form, so using the graphics methods can work well when you want to create graphics that appear behind everything else in your application. For more information, see the section "Layering Graphical Controls and Graphics Methods," later in this chapter.

Limitations of Graphics Methods

Creating graphics with the graphics methods takes place in the code of your application. For this reason, you must run the application to see the effect of a graphics method. With graphical controls, you can see the control on the form at design time, so graphics methods don't work as well for creating simple design elements of the interface of an application. Changing the appearance of graphical controls at design time is easier to do than modifying and testing the code for a graphics method.

Creating Graphics When a Form Loads

When creating graphics that appear on a form when it loads, place the graphics methods in the Form_Paint event, not the Form_Load event. Remember, forms are not visible during the Form_Load event. Because Visual Basic does not process graphics methods on a form that is not visible, graphics methods in the Form_Load event are ignored.

If you put the graphics methods in the Form_Paint event, the graphics appear when the form becomes visible.

The Fundamentals of Drawing

Every graphics method draws output on a form, a picture box, or the Printer object. To indicate where you want to draw, precede a graphics method with the name of a form or picture box control. If you omit the object, Visual Basic assumes you want to draw on the form to which the code is attached. For example, the following statements draw a point on:

- A form named MyForm

```
MyForm.PSet (500, 500)
```

- A picture box named picPicture1

```
picPicture1.PSet (500, 500)
```

- The current form

```
PSet (500, 500)
```

Each drawing area has its own coordinate system that determines what units apply to the coordinates. In addition, every drawing area has its own complete set of graphics properties. See Chapter 16, "Displaying and Printing Information," for more information about the Printer object.

Clearing the Drawing Area

Any time you want to clear a drawing area and start over, use the **Cls** method. The specified drawing area is repainted in the background color (BackColor). The **Cls** method has the simplest syntax of any of the graphics methods:

[*object.*]**Cls**

Using the **Cls** method without a specified *object* clears the form to which the code is attached.

Plotting Points

Another simple graphics operation is controlling an individual pixel. The **PSet** method sets the color of a pixel at a specified point:

[*object.*]**PSet** (*x*, *y*)[, *color*]

The arguments *x* and *y* are single precision, so they can take either integer or fractional input. The input can be any numeric expression, including variables.

If you don't include the color argument, **PSet** sets a pixel to the foreground color (ForeColor). For example, the following statements set various points on the current form (the form to which the code is attached), MyForm, and picPicture1:

```
PSet (300, 100)
PSet (10.75, 50.33)
MyForm.PSet (230, 1000)
picPicture1.PSet (1.5, 3.2)
```

Adding a color argument gives you more control:

```
PSet (50, 75), RGB(0, 0, 255)       ' Set 50, 75 to bright blue.
```

Example

The Blanker application plots points with randomly selected colors to create the Confetti demo. The PSetDemo procedure creates the confetti:

```
Sub PSetDemo ()
    R = 255 * Rnd                       ' Set Red to random value.
    G = 255 * Rnd                       ' Set Green to random value.
    B = 255 * Rnd                       ' Set Blue to random value.
    XPos = Rnd * ScaleWidth             ' Set horizontal position.
    YPos = Rnd * ScaleHeight            ' Set vertical position.
    PSet (XPos, YPos), RGB(R, G, B)     ' Plot point with random color.
End Sub
```

The resulting confetti display is shown in Figure 15.12.

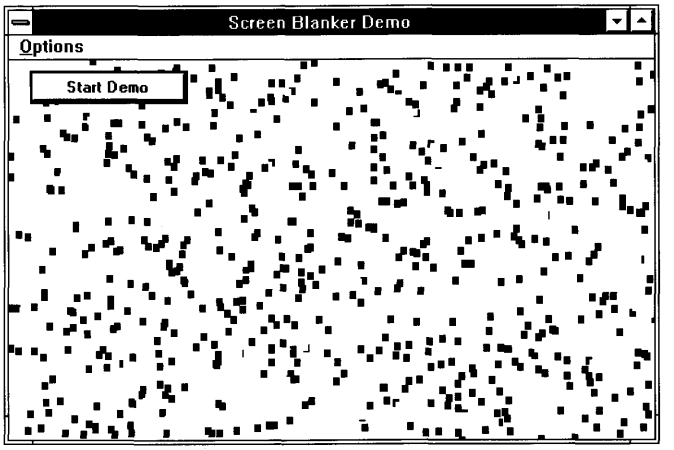


Figure 15.12 Confetti display in the Blanker application

To "erase" a point, set it to the background color:

```
PSet (50, 75), BackColor
```

As described in the section "The Step Keyword," later in this chapter, you can precede the (*x*, *y*) coordinates by **Step**, which makes the point relative to the last location drawn.

Closely related to the **PSet** method is the **Point** method, which returns the color value at a particular location:

```
PointColor = Point (500, 500)
```

For More Information For information on the **PSet** method, see the *Language Reference*, or search Help for *PSet*.

Drawing Lines and Shapes

Although clearing the drawing area and plotting individual points can be useful, the most interesting graphics methods draw complete lines and shapes.

Drawing Lines

To draw a line between two coordinates, use the simple form of the **Line** method, which has this syntax:

[*object.*]**Line** [(*x1*, *y1*)]–(*x2*, *y2*)[, *color*]

For example, this statement draws a slanted line on a form.

```
Line (500, 500)-(2000, 2000)
```

Object is optional; if omitted, the method draws to the form to which the code is attached (the current form). The first pair of coordinates is also optional. As with all coordinate values, the arguments *x* and *y* can be either integer or fractional numbers. Visual Basic draws a line that includes the first end point, but not the last end point. This behavior is useful when drawing a closed figure from point to point. To get the last point, use this syntax:

PSet Step(0, 0)[, *color*]

The first pair of coordinates (*x1, y1*) is optional. If you omit these coordinates, Visual Basic uses the object's current *x, y* location (drawing coordinates) as the end point. The current location can be specified with the CurrentX and CurrentY properties, but otherwise is equal to the last point drawn by a previous graphics or **Print** method. If you haven't previously used a graphics or **Print** method or set CurrentX and CurrentY, the default location is the object's upper-left corner.

For example, the following statements draw a triangle by connecting three points. The results are shown in Figure 15.13.

```
CurrentX = 1500          ' Set x-coordinate of starting point.
CurrentY = 500           ' Set y-coordinate of starting point.
Line -(3000, 2000)       ' Draw line down and right of starting point.
Line -(1500, 2000)       ' Draw line to the left of current point.
Line -(1500, 500)        ' Draw line up and right to starting point.
```

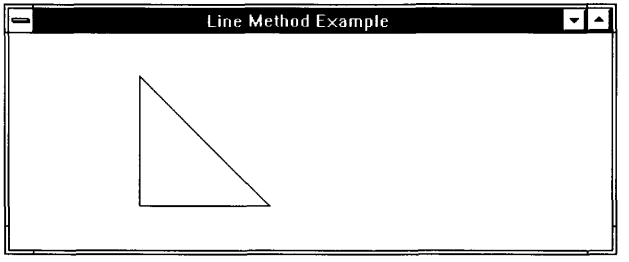


Figure 15.13 A triangle drawn with the Line method

By adding variables and loops, you can create interesting patterns. In the Blanker sample application, the LineDemo procedure for the Crossfire demo draws random lines on the form with these statements:

```
Sub LineDemo ()
    R = 255 * Rnd                    ' Set Red to random value.
    G = 255 * Rnd                    ' Set Green to random value.
    B = 255 * Rnd                    ' Set Blue to random value.
    X2 = Int(Width * Rnd + 1)        ' Set X position to random spot.
    Y2 = Int(Height * Rnd + 1)       ' Set Y position to random spot.
    Line -(X2, Y2), RGB(R, G, B)     ' Connect X1Y1 to X2Y2.
End Sub
```

The **Line** method in this procedure doesn't specify a starting point, thereby assuming the values for CurrentX and CurrentY.

The results produced by the Crossfire demo are shown in Figure 15.14.

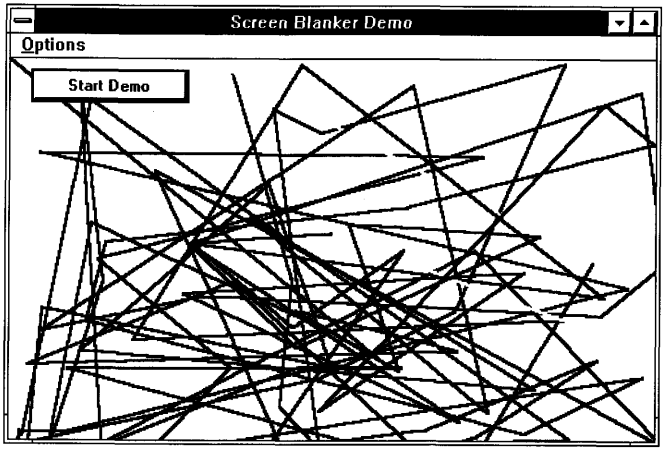


Figure 15.14 The Crossfire demo in the Blanker application

The Step Keyword

The **PSet**, **Line**, and **Circle** methods specify one or more points using this syntax:

(*x*, *y*)

You can precede each of these points with the **Step** keyword, specifying that the location of the point is relative to the last point drawn. Visual Basic adds the values *x* and *y* to the values of the last point drawn. For example, the statement:

```
Line (100, 200)-(150, 250)
```

is equivalent to:

```
Line (100, 200)-Step(50, 50)
```

In many situations, the **Step** keyword saves you from having to constantly keep track of the last point drawn. Often you are more interested in the relative position of two points than their absolute position.

Drawing Boxes

You can draw and fill boxes using the **Line** method. The following example draws a box with an upper-left corner at (500, 500) and measuring 1,000 twips on each side:

```
Line (500, 500)-Step(1000, 0)
Line -Step(0, 1000)
Line -Step(-1000, 0)
Line -Step(0, -1000)
```

However, Visual Basic provides a much simpler way to draw a box. When you use the **B** option with the **Line** method (**B** comes after the argument *color*), Visual Basic draws a rectangle, treating the specified points as opposite corners of the rectangle. Thus, you could replace the four statements of the previous example with:

```
Line (500, 500)-Step(1000, 1000), , B
```

Note that two commas are required before **B**, to indicate the color argument was skipped. (Otherwise, **B** could be interpreted as a variable giving the color value.)

As long as you do not change the setting of the FillStyle property, the box appears empty. Actually, the box does get filled using the current FillStyle and FillColor settings. But FillStyle defaults to 1-Transparent, which specifies an invisible pattern. You can change the FillStyle property to any of the settings listed in the following table.

Setting	Description
0	Solid. Fills in box with the color set for the FillColor property.
1	Transparent (the default). Graphical object appears empty, no matter what color is used.
2	Horizontal lines.
3	Vertical lines.
4	Upward diagonal lines.
5	Downward diagonal lines.
6	Crosshatch.
7	Diagonal crosshatch.

Thus, setting FillStyle to 0 fills the box solidly with the color set for the FillColor property.

Another way to fill the box is to specify **F** after the **B**. (Note that **F** cannot be used without **B**.) When you use the **F** option, the **Line** method ignores FillColor and FillStyle. The box is always filled solid when you use the **F** option. The following statement fills the box with a solid pattern, using the DrawColor property. The result is shown in Figure 15.15.

```
Line (500, 500)-Step(1000, 1000), , BF
```

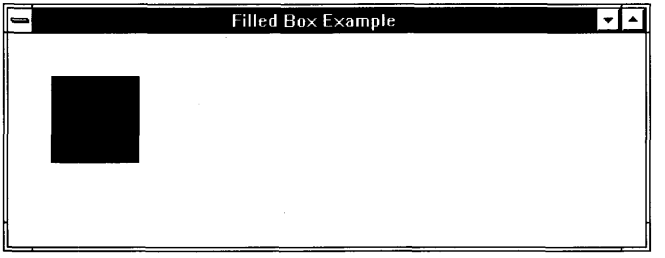


Figure 15.15 A box filled with a solid pattern

Drawing Circles

The **Circle** method draws a variety of circular and elliptical (oval) shapes. In addition, **Circle** draws arcs (segments of circles) and pie-shaped wedges. You can produce many kinds of curved lines using variations of the **Circle** method.

To draw a circle (as opposed to an ellipse), Visual Basic needs the location of a circle's center and the length of its radius. The syntax for a perfect circle is:

[*object.*]**Circle** [**Step**](*x, y*), *radius*[, *color*]

The brackets indicate that both *object* and the **Step** keyword are optional. If you don't specify *object*, the current form is assumed. The arguments *x* and *y* are the coordinates of the center, and *radius* is the radius of the circle. For example, this statement draws a circle with a center at (1200, 1000) and radius of 750:

```
Circle (1200, 1000), 750
```

The exact effect of this statement depends on the size and coordinate system of the form. Because the size of the form is unknown, you don't even know if the circle will be visible. Using the drawing area's scale properties puts the center of the circle at the center of the form:

```
Circle ((ScaleWidth + ScaleLeft)/2, (ScaleHeight + ScaleTop)/2), ScaleWidth/4
```

For now, all you need to know about ScaleWidth and ScaleHeight is that they help position graphics in the center of a form. The section "Changing an Object's Coordinate System," later in this chapter, discusses the ScaleWidth and ScaleHeight properties in detail.

Note The radius of the circle is always specified in terms of horizontal units. If your coordinate system uses the same horizontal and vertical units (which it does by default), you can ignore this fact. However, if you use a custom scale, horizontal and vertical units may correspond to different distances. In the preceding examples, the radius is specified in horizontal units, and the actual height of the circle is guaranteed to be equal to its actual width.

Example

The Blanker application creates circles as part of the Rainbow Rug demo. This demo draws a series of dashed line circles around the center of the form. In time the circles resemble a woven circular rug. The CircleDemo procedure creates the circles in the Rainbow Rug demo with the following statements:

```
Sub CircleDemo ()
    R = 255 * Rnd                              ' Set Red to a random value.
    G = 255 * Rnd                              ' Set Green to a random value.
    B = 255 * Rnd                              ' Set Blue to a random value.
    XPos = ScaleWidth / 2                      ' Set x-coordinate in middle of form.
    YPos = ScaleHeight / 2                     ' Set y-coordinate in middle of form.
    Radius = ((YPos * .9) + 1) * Rnd           ' Set radius between 0 & 90% of form
                                               ' height.
    Circle (XPos, YPos), Radius, RGB(R, G, B)  ' Draw the circle using a random color.
End Sub
```

The results of the Rainbow Rug demo are shown in Figure 15.16.

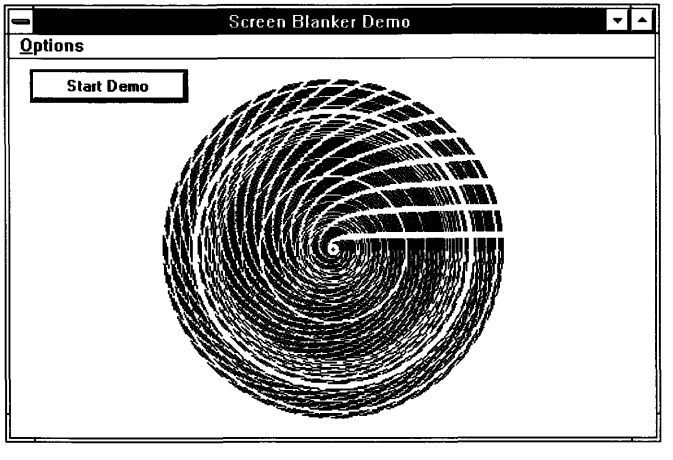


Figure 15.16 The Rainbow Rug demo in the Blanker application

Drawing Ellipses

The aspect ratio of a circle controls whether or not it appears perfectly round (a circle) or elongated (an ellipse). The complete syntax for the **Circle** method is:

[*object.*]**Circle** [**Step**](*x, y*), *radius*, [*color*], [*start*], [*end*], [*aspect*]

Here, *aspect* is a positive floating-point number. This means you can specify integer or fractional expressions, but not negative values.

The arguments *start* and *end* are used in displaying arcs, as described in the next section. These arguments are optional, but the commas are necessary if you want to skip arguments. For example, if you include the *radius* and *aspect* arguments, but no *color*, *start*, or *end* argument, you must add four successive commas to indicate that you're skipping the three arguments:

```
Circle (1000, 1000), 500, , , , 2
```

The argument *aspect* specifies the ratio of the vertical to horizontal dimensions. Large values for *aspect* produce ellipses stretched out along the vertical axis, while small values for *aspect* produce ellipses stretched out along the horizontal axis. Since an ellipse has two radii —one horizontal *x*-radius and one vertical *y*-radius — Visual Basic applies the single argument *radius* in a **Circle** statement to the longer axis. If *aspect* is less than one, *radius* is the *x*-radius; if *aspect* is greater than or equal to one, *radius* is the *y*-radius.

Note The argument *aspect* always specifies the ratio between the vertical and horizontal dimensions in terms of true physical distance. To ensure that this happens (even when you use a custom scale), the radius is specified in terms of horizontal units.

The following procedure illustrates how different *aspect* values determine whether **Circle** uses the argument *radius* as the *x*-radius or the *y*-radius of an ellipse. The output is shown in Figure 15.17.

```
Sub Form_Click ()
    FillStyle = 0                          ' Draw solid ellipse.
    Circle (600, 1000), 800, , , , 3
    FillStyle = 1                          ' Draw empty ellipse.
    Circle (1800, 1000), 800, , , , 1/3
End Sub
```

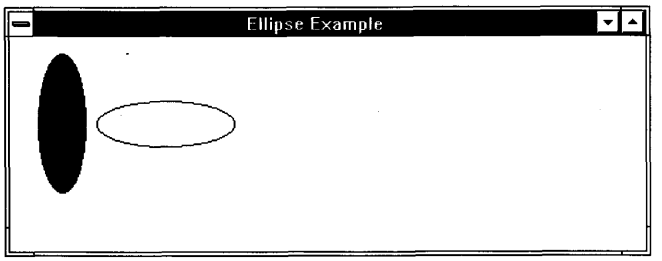


Figure 15.17 Ellipses drawn with the Circle method

Drawing Arcs

An arc is a segment of an ellipse or a circle; think of an arc as a curved line. To understand how the **Circle** method draws arcs, you need to know how Visual Basic measures angles.

Visual Basic uses the radian as its unit of angle measure, not only in the **Circle** method, but also in the trigonometric functions such as **Cos**, **Sin**, or **Tan**.

The radian is closely related to the radius of a circle. The circumference of a circle equals 2 * Pi * *radius*, where Pi is equal to approximately 3.14159265. Similarly, the number of radians in one complete angle of revolution (or 360 degrees) equals 2 * Pi, or a little more than 6.28.

This statement calculates Pi to double precision:

```
Dim Pi As Double
Pi = 4 * Atn(1)
```

If you think of angles in terms of degrees, here are some common equivalents.

Angle in degrees	Angle in radians
360	2*Pi (approximately 6.283)
180	Pi (approximately 3.142)
90	Pi/2 (approximately 1.571)
60	Pi/3 (approximately 1.047)
45	Pi/4 (approximately 0.786)

Imagining a clock face on the screen, **Circle** measures angles by starting at the three o'clock position and rotating counterclockwise, as shown in Figure 15.18.

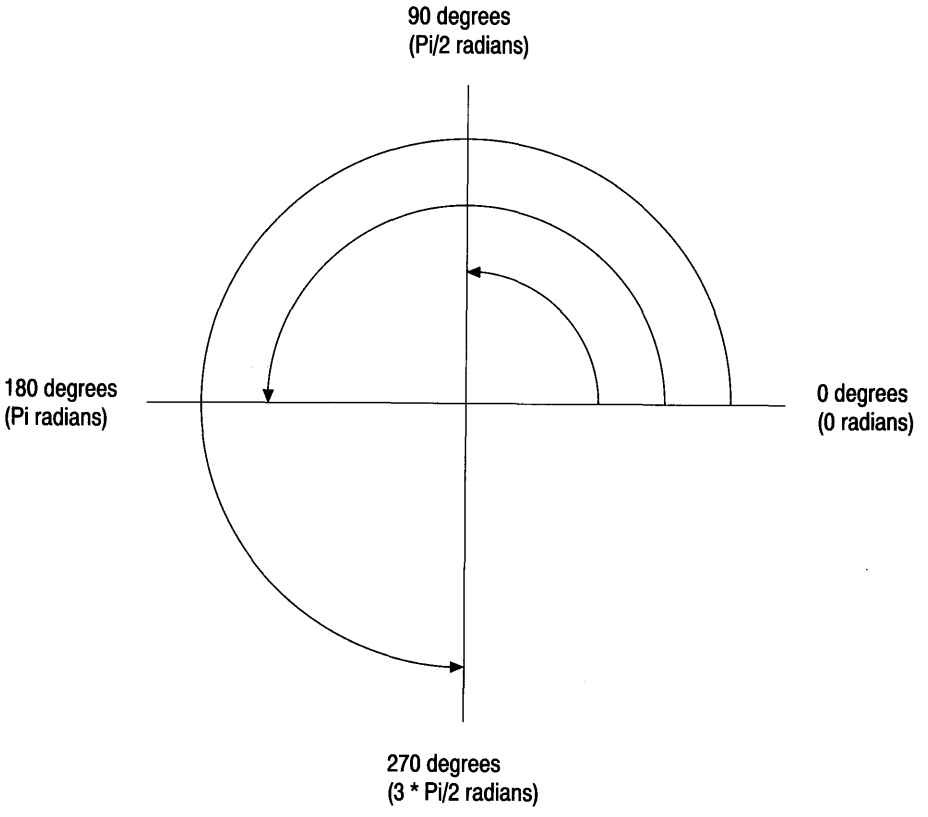


Figure 15.18 How the Circle method measures angles for a circle

The general formula for converting from degrees to radians is to multiply degrees by Pi/180.

To draw an arc, give angle arguments defining the arc's limits. The syntax is:

[*object.*]**Circle** [**Step**] (*x*, *y*), *radius*, [*color*], *start*, *end*[, *aspect*]

In the following example, the **Circle** method is used to draw seven arcs. The results are shown in Figure 15.19.

```
Sub Form_Click ()
    Height = 5500                                          ' Define form height.
    Const PI = 3.14159265                                  ' Define Pi as constant.
    StartAngle = 0                                         ' Define first starting angle.
    For Radius = 1000 To 2200 Step 200                     ' For each of 7 radii:
    EndAngle = (EndAngle + (PI / 2)) Mod (2 * PI)          ' Calculate end of arc;
        Circle (2500, 2500), Radius, , StartAngle, EndAngle     ' draw arc;
        StartAngle = StartAngle + (PI / 4)                 ' add 90 deg for next start;
    Next Radius                                            ' and continue with next arc.
End Sub
```

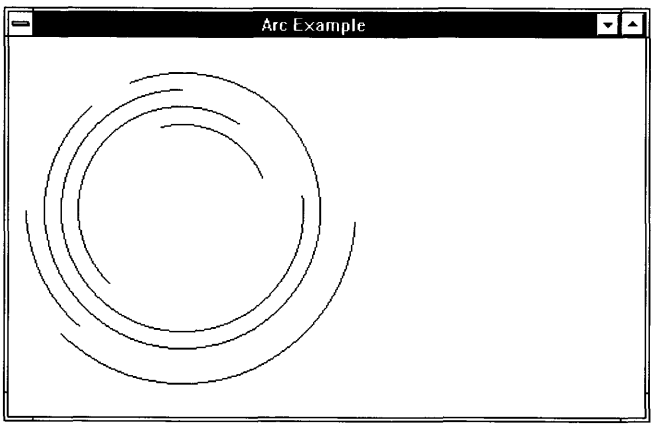


Figure 15.19 Using the Circle method to draw arcs

If the argument *start* or *end* is negative, Visual Basic draws a line connecting the center of the circle to the negative end point. For example, the following procedure draws a pie with a slice removed, as shown in Figure 15.20.

```
Sub Form_Click ()
    Const PI = 3.14159265
    Circle (3500, 1500), 1000, , -PI / 2, -PI / 3
End Sub
```

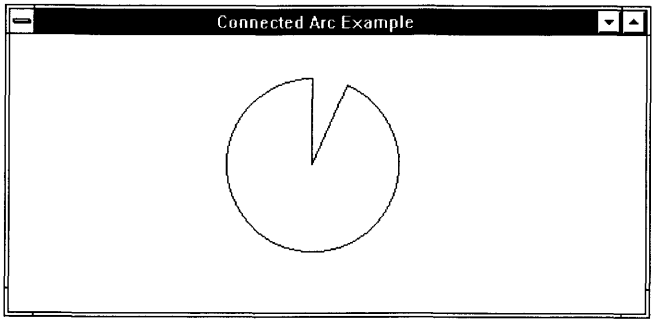


Figure 15.20 Using negative arguments to connect an arc to its center

Figure 15.20 also illustrates that –Pi/2 and –Pi/3 indicate the same angles (points on the circle) that Pi/2 and Pi/3 do. The only significance of the negative sign is that it causes Visual Basic to draw a line between the point and the center of the circle.

Note that the arc is drawn the long way between the two angles rather than the short way. This is because Visual Basic always draws arcs in a counterclockwise (positive) direction. If the angles are given in the reverse order (Pi/3, Pi/2), Visual Basic draws the arc using the short distance between the two angles, as shown in Figure 15.21.

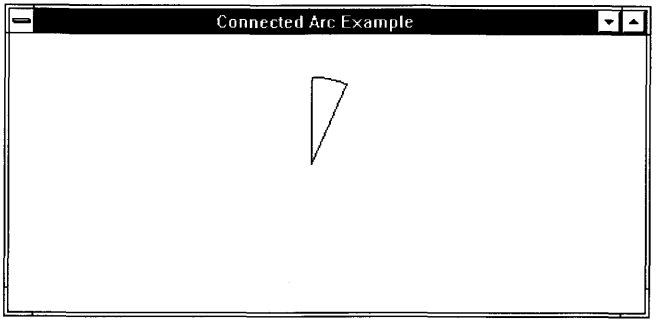


Figure 15.21 The effect of reversing arguments

Changing an Object's Coordinate System

You set the coordinate system for a particular object (form or control) using the object's scale properties and the **Scale** method. You can use the coordinate system in one of three different ways:

- Use the default scale.
- Select one of several standard scales.
- Create a custom scale.

Changing the scale of the coordinate system can make it easier to size and position graphics on a form. For example, an application that creates bar charts in a picture box can change the coordinate system to divide the control into four columns, each representing a bar in the chart. The next several sections explain how to set default, standard, and custom scales to change the coordinate system.

Using the Default Scale

Every form and picture box has several scale properties (ScaleLeft, ScaleTop, ScaleWidth, ScaleHeight, and ScaleMode) and one method (**Scale**) you can use to define the coordinate system. The default scale for objects in Visual Basic places the coordinate (0,0) at the upper-left corner of the object. The default scale uses units called twips. There are 1,440 twips to one inch.

If you change the scale of an object and want to return to the default scale, use the **Scale** method with no arguments.

Selecting a Standard Scale

Instead of defining units directly, you can define them in terms of a standard scale by setting the ScaleMode property to one of the settings shown in the following table.

ScaleMode setting	Description
0	User-defined. If you set ScaleWidth, ScaleHeight, ScaleTop, or ScaleLeft directly, the ScaleMode property is automatically set to 0.
1	Twips. This is the default scale. There are 1,440 twips to one inch.

ScaleMode setting	Description
2	Points. There are 72 points to one inch.
3	Pixels. A pixel is the smallest unit of resolution on the monitor. The number of pixels per inch depends on the resolution of the monitor.
4	Characters. When printed, a character is 1/6 of an inch high and 1/12 of an inch wide.
5	Inches.
6	Millimeters.
7	Centimeters.

All of the modes in the table, except for 0 and 3, refer to printed lengths. For example, an item that is two units long in ScaleMode 7 is two centimeters long when printed.

These statements set the ScaleMode property:

```
ScaleMode = 5                    ' Set scale to inches for this form.
picPicture1.ScaleMode = 3        ' Set scale to pixels for picPicture1.
```

Setting a value for ScaleMode causes Visual Basic to redefine ScaleWidth and ScaleHeight so that they are consistent with the new scale. ScaleTop and ScaleLeft are then set to 0. Setting ScaleWidth, ScaleHeight, ScaleTop, or ScaleLeft directly automatically sets ScaleMode to 0.

Creating a Custom Scale

You can use an object's ScaleLeft, ScaleTop, ScaleWidth, and ScaleHeight properties to create a custom scale. Unlike the **Scale** method, these properties can be used either to set the scale or to get information about the current scale of the coordinate system.

Using ScaleLeft and ScaleTop

The ScaleLeft and ScaleTop properties assign numeric values to the upper-left corner of an object. For example, these statements set the value of the upper-left corner for the current form and upper-left corner for a picture box named picArena. These scale values are shown in Figure 15.22.

```
ScaleLeft = 100
ScaleTop = 100
picArena.ScaleLeft = 100
picArena.ScaleTop = 100
```

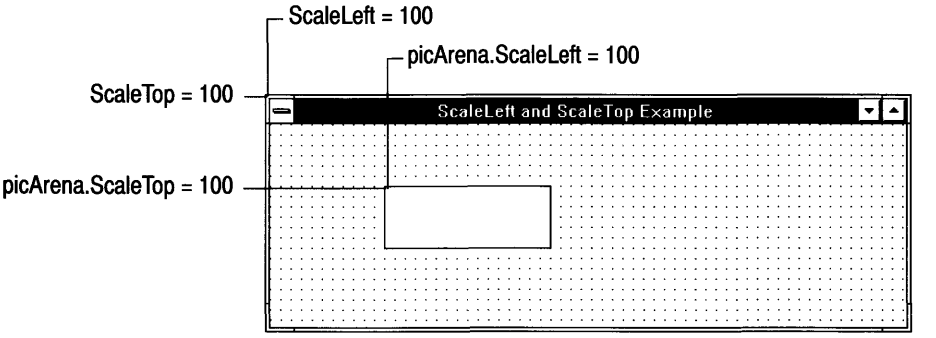


Figure 15.22 The ScaleLeft and ScaleTop properties for a form and a control

These example statements define the upper-left corner as (100, 100). Although the statements don't directly change the size or position of these objects, they alter the effect of subsequent statements. For example, a subsequent statement that sets a control's Top property to 100 places the object at the very top of its container.

Using ScaleWidth and ScaleHeight

The ScaleWidth and ScaleHeight properties define units in terms of the current width and height of the drawing area. For example:

```
ScaleWidth = 1000
ScaleHeight = 500
```

These statements define a horizontal unit as 1/1,000 of the current internal width of the form and a vertical unit as 1/500 of the current internal height of the form. If the form is later resized, the units remain the same.

Note ScaleWidth and ScaleHeight define units in terms of the internal dimensions of the object — these dimensions do not include the border thickness. Thus, ScaleWidth and ScaleHeight always refer to the amount of room available *inside* the object. The distinction between internal and external dimensions (specified by Width and Height) is particularly important with forms, which can have a thick border. The units can also differ: Width and Height are always expressed in terms of the *container's* coordinate system; ScaleWidth and ScaleHeight determine the coordinate system of the object itself.

Setting Properties to Change the Coordinate System

All four of these scale properties can include fractions, and they can also be negative numbers. Negative settings for the ScaleWidth and ScaleHeight properties change the orientation of the coordinate system.

The scale shown in Figure 15.23 has ScaleLeft, ScaleTop, ScaleWidth, and ScaleHeight all set to 100.

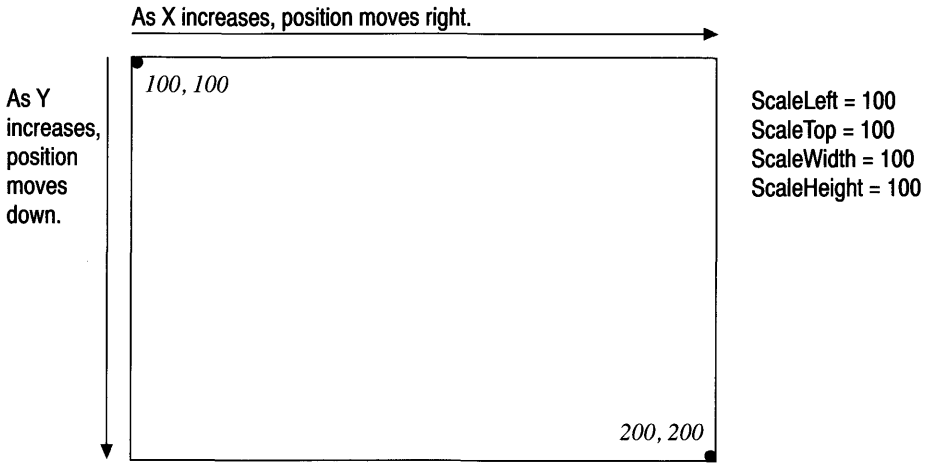


Figure 15.23 Scale running from (100, 100) to (200, 200)

Note that coordinates increase in value as you run from top to bottom, and from left to right. If you want to change this scale so that coordinates increase as you go from bottom to top, then you set ScaleTop to 200 and ScaleHeight to −100, as shown in Figure 15.24.

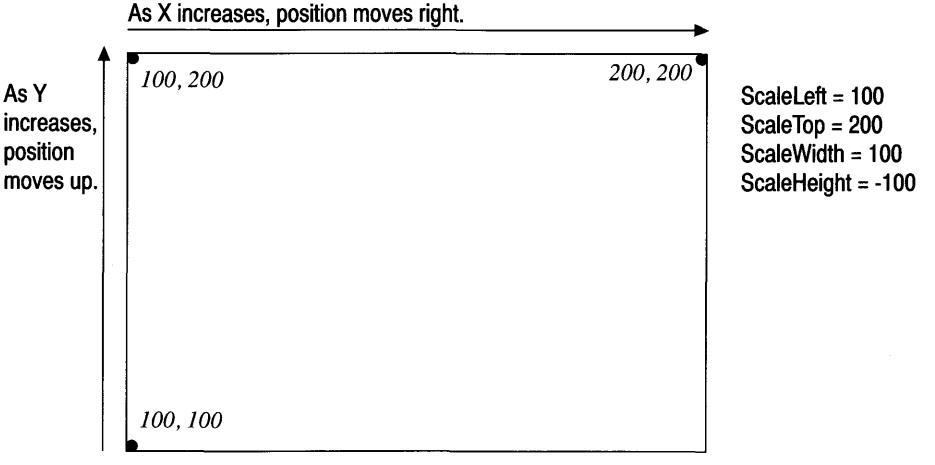


Figure 15.24 A scale with a negative ScaleHeight setting

Example

The Color Bars demo in the Blanker application changes the coordinate system of the form to draw a series of vertical bars that change height with time. Each time you click the cmdStartStop command button while Color Bars is selected in the Options menu, the following statements change the coordinate system of DemoForm:

```
ScaleLeft = 1                          ' Set x-coord. of left edge to 1.
ScaleTop = 10                          ' Set y-coord. of top edge to 10.
ScaleWidth = Int(13 * Rnd + 3)         ' Set width to random value.
ScaleHeight = -10                      ' Height is 10; y-axis runs up.
```

The new coordinate system is 10 units high and between 3 and 15 units wide (the exact value set at random each time you start the Blanker). The lower-left corner of the form has the coordinates (1, 0) as determined by the values of ScaleLeft (for the x-coordinate), ScaleTop and ScaleHeight (for the y-coordinate).

The colored bars that appear on the form are created by these statements. The height of each bar is set at random between 0 and 10. The results produced by the Color Bars demo are shown in Figure 15.25.

```
For Box = 1 To ScaleWidth                                    ' 1 box uses 1 unit of width.
    R = 255 * Rnd                                            ' Set Red to random value.
    G = 255 * Rnd                                            ' Set Green to random value.
    B = 255 * Rnd                                            ' Set Blue to random value.
    Line (Box, 0)-Step(1, (Int(11 * Rnd))), RGB(R, G, B), BF ' Draw and fill the box.
Next Box                                                     ' Do the next box.
Delay                                                        ' Pause display briefly.
Cls                                                          ' Clear stray pixels.
```

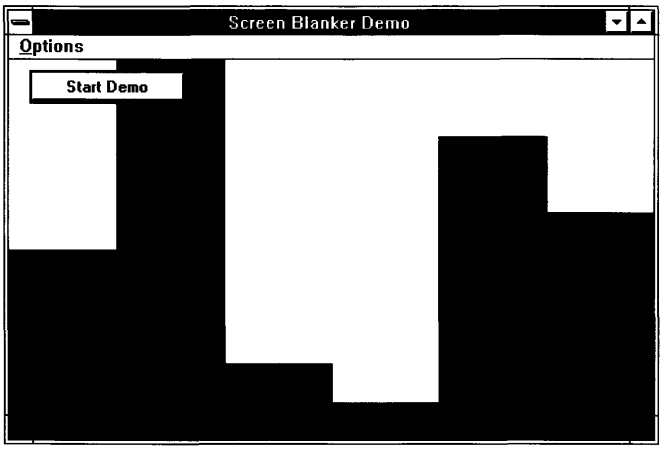


Figure 15.25 The Color Bars demo in the Blanker application

The Scale Method: A Shortcut

A more efficient way of changing the coordinate system than setting individual properties is the **Scale** method. You specify a custom scale using this syntax:

[*object.*]**Scale** (*x1*, *y1*) – (*x2*, *y2*)

The values of *x1* and *y1* determine the settings of the ScaleLeft and ScaleTop properties. The differences between the two x-coordinates and the two y-coordinates determine the settings of ScaleWidth and ScaleHeight, respectively. For example, suppose you set the coordinate system for a form by setting end points (100, 100) and (200, 200):

```
Scale (100, 100)-(200, 200)
```

This statement defines the form as 100 units wide and 100 units high. With this scale in place, the following statement moves a shape control one-fifth of the way across the form:

```
shpMover.Left = shpMover.Left + 20
```

Specifying a value of *x2* >*x1* or *y2* >*y1* has the same effect as setting ScaleWidth or ScaleHeight to a negative value.

For More Information For information on setting coordinate systems, search Help for *coordinate system.*

Using Graphics Properties

Forms and various controls have graphics properties. The following table lists these properties.

Category	Properties
Display processing	AutoRedraw, ClipControls
Current drawing location	CurrentX, CurrentY
Drawing techniques	DrawMode, DrawStyle, DrawWidth, BorderStyle, BorderWidth
Filling techniques	FillColor, FillStyle
Colors	BackColor, ForeColor, BorderColor, FillColor

Forms and picture boxes have additional properties:

- Scale properties described in the section "Changing an Object's Coordinate System."
- Font properties described in Chapter 16, "Displaying and Printing Information."

There are two properties of forms and picture boxes you'll probably want to use right away: BackColor and ForeColor. BackColor paints the background of the drawing area. If BackColor is light blue, then the entire area is light blue when you clear it. ForeColor (foreground) determines the color of text and graphics drawn on an object, although some graphics methods give you the option of using a different color.

Using AutoRedraw

Each form and picture box has an AutoRedraw property. AutoRedraw is a Boolean property that, when set to **True**, causes graphics output to be saved in memory. You can use the AutoRedraw property to create persistent graphics.

Persistent Graphics

Microsoft Windows manipulates the screen image to create an illusion of overlapping windows. When one window is moved over another, temporarily hiding it, and then is moved away again, the window and its contents need to be redisplayed. Windows takes care of redisplaying the window and controls. But your Visual Basic application must handle redisplaying graphics in a form or picture box.

If you create graphics on the form using graphics methods, you usually want them to reappear exactly as you placed them. This is what is meant by *persistent graphics*. You can use the AutoRedraw property to create persistent graphics.

AutoRedraw and Forms

The default setting of AutoRedraw is **False**. When AutoRedraw is set to **False**, any graphics created by graphics methods that appear on the form are lost if another window temporarily hides them. Also, graphics that extend beyond the edges of the form are lost if you enlarge the form. The effects of setting AutoRedraw to **False** are shown in Figure 15.26.

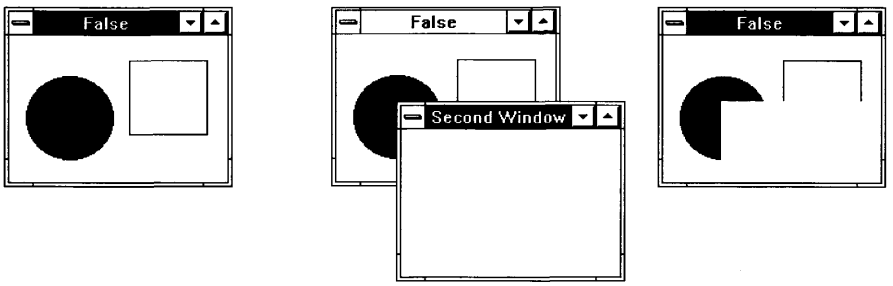


Figure 15.26 The effects of setting AutoRedraw to False

When the AutoRedraw property of a form is set to **True**, Visual Basic applies graphics methods to a "canvas" in memory. The application copies the contents of this memory canvas to redisplay graphics temporarily hidden by another window. In most cases, the size of this canvas for forms is the size of the screen. If the form's MaxButton property is **False** and the border of the form is not sizable, the size of the canvas is the size of the form.

This canvas also lets the application save graphics that extend beyond the edges of the form when the form is resizable. The effects of setting AutoRedraw to **True** are shown in Figure 15.27.

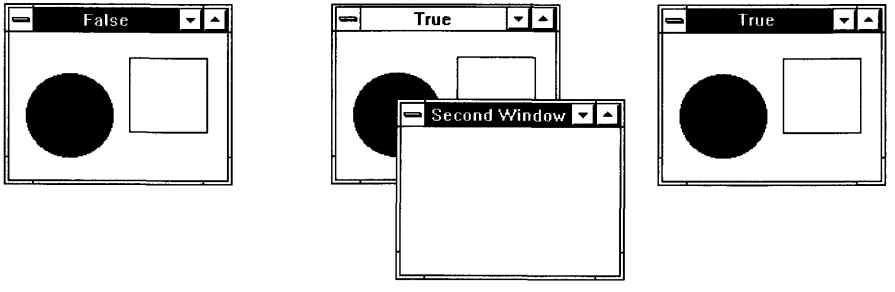


Figure 15.27 The effects of setting AutoRedraw to True

AutoRedraw and Picture Boxes

When the AutoRedraw property of a picture box is set to **True**, Visual Basic saves only the visible contents of the picture box in memory. This is because the memory canvas used to save the contents of the picture box is the same size as the picture box. Graphics that extend outside the picture box are cropped and never appear later, even if the size of the picture box changes.

Conserving Memory

There is a trade-off between persistent graphics and application performance. Using AutoRedraw uses memory to store and redisplay graphics in the application. This memory usage may reduce the application's performance. You can use computer memory more efficiently by selectively using the AutoRedraw property for forms and picture box controls.

Using AutoRedraw Only for Picture Boxes on a Form

If you want to conserve memory, create a picture box for all drawing effects and leave AutoRedraw set to **False** for the form. If you set AutoRedraw to **True** for the picture box, any graphics in that picture box will be persistent. The application conserves memory by saving only the contents of the picture box, and not the entire form, on the canvas in memory. This is because the size of the canvas for the form is generally the size of the screen.

Using Nonpersistent Graphics

Alternatively, you can leave AutoRedraw set to **False** for the form and all its picture boxes. Memory is conserved because the application never saves graphics in memory. But then the graphics are not automatically persistent. You have to manage redrawing all graphics in code as needed.

You can include code in the Paint event for a form or picture box that redraws all lines, circles, and points as appropriate. This approach usually works best when you have a limited amount of graphics that you can reconstruct easily.

A Paint event procedure is called whenever part of a form or picture box needs to be redrawn —for example, when a window that covered the object moves away, or when resizing causes graphics to come back into view. If AutoRedraw is set to **True**, the object's Paint procedure is never called unless your application calls it explicitly. This happens because the visible contents of the object are stored in the memory canvas, so the Paint event isn't needed.

Keep in mind that the decision to use nonpersistent graphics can affect the way graphics paint on the form or container. The sections "Using ClipControls" and "Layering Graphical Controls and Graphics Methods" discuss other factors that may determine whether or not you should use nonpersistent graphics.

Changing AutoRedraw at Run Time

You can change the setting of AutoRedraw at run time. If AutoRedraw is **False**, graphics and output from the **Print** method are written only to the screen, not to memory. If you clear the object with the **Cls** method, any output written when AutoRedraw was set to **True** does not get cleared. This output is retained in memory, and you must set AutoRedraw to **True** again and then use the **Cls** method to clear it.

Using ClipControls

Each form, picture box, and frame control has a ClipControls property. ClipControls is a Boolean property that, when set to **True**, causes the container to define a clipping region when painting the container around all controls except:

- The shape control
- The line control
- The image control
- Labels
- Any custom graphical controls

By setting the ClipControls property to **False**, you can improve the speed with which a form paints to the screen. The speed improvement is greatest on forms with many controls that do not overlap, like dialog boxes.

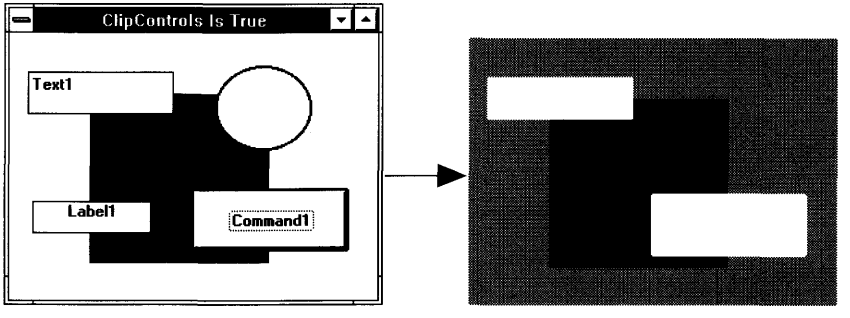
Clipping Regions

Clipping is the process of determining which parts of a form or container are painted when the form or container is displayed. The outline used to determine what parts of the form or container are painted or "clipped" defines the *clipping region* for that form or container. Clipping regions are useful when a Windows–based application needs to save one part of the display and simultaneously repaint the rest.

Clipping Forms and Containers

The default setting of ClipControls is **True**. When the ClipControls property is **True**, Windows defines a clipping region for the background of the form or container before a Paint event. This clipping region surrounds all nongraphical controls. When using ClipControls, labels act like graphical controls.

During a Paint event, Windows repaints only the background inside the clipping region, avoiding the nongraphical controls. Figure 15.28 shows a form with four controls, a box painted with the **Line** method, and the clipping region created for that form by setting ClipControls to **True**. Notice that the clipping region did not clip around the label or shape controls on the form. The box drawn in the background with the **Line** method paints only in the clipping region.



Form with two non-graphical controls, a shape control, a label, and a box created with the **Line** method.

Clipping region shown in light gray; shape control and label not clipped. The box paints only in the clipping region.

Figure 15.28 The clipping region created when ClipControls is True

When ClipControls is **False**, Windows does not define a clipping region for the background of the form or container before a Paint event. Also, output from graphics methods within the Paint event appears only in the parts of the form or container that need to be repainted. Since calculating and managing a clipping region takes time, setting ClipControls to **False** may cause forms with many nonoverlapping controls (such as complex dialog boxes) to display faster.

The Effects of AutoRedraw and ClipControls

Various combinations of AutoRedraw and ClipControls settings can affect an application in three ways:

- Memory consumption
- Speed with which a form paints on the screen
- Display layering of graphical controls and graphics methods

Setting AutoRedraw to **True** lets you create persistent graphics, but with the trade-off of increased memory consumption. Setting ClipControls to **False** may cause forms with many nonoverlapping controls to paint faster, but may also cause undesirable graphical effects depending on the setting of AutoRedraw and whether graphics methods are used outside of Paint events.

This section discusses the effects of different combinations of AutoRedraw and ClipControls on the way graphical controls and graphics methods paint to the screen.

Layering Graphical Controls and Graphics Methods

As you create graphics, keep in mind that graphical controls and labels, nongraphical controls, and graphics methods appear on different layers in a container. The behavior of these layers depends on three factors:

- The AutoRedraw setting
- The ClipControls setting
- Whether graphics methods appear inside or outside the Paint event

Normal Layering

Normally, the layers of a form or other container are, from front to back:

Layer	Contents
Front	Nongraphical controls like command buttons, check boxes, file controls.
Middle	Graphical controls and labels.
Back	Drawing space for the form or container. This is where the results of graphics methods appear.

Anything in one layer covers anything in the layer behind, so graphics you create with the graphical controls appear behind the other controls on the form, and all graphics you create with the graphics methods appear below all graphical and nongraphical controls. The normal arrangement of layers is shown in Figure 15.29.

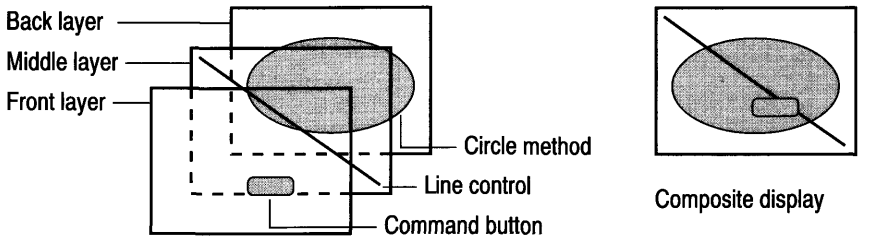


Figure 15.29 Normal layering of graphics on a form

Effects on Layering

You can produce normal layering using any of several approaches. Combining settings for AutoRedraw and ClipControls and placing graphics methods inside or outside the Paint event affects layering and the performance of the application.

The following table summarizes the effects created by different combinations of AutoRedraw and ClipControls and placement of graphics methods.

AutoRedraw	ClipControls	Graphics methods in/out of Paint event	Layering behavior
True	**True** (default)	Paint event ignored	Normal layering.
True	**False**	Paint event ignored	Normal layering. Forms with many controls that do not overlap may paint faster because no clipping region is calculated or created.
False (default)	**True** (default)	In	Normal layering.
False	**True**	Out	Nongraphical controls in front. Graphics methods and graphical controls appear mixed in the middle and back layers. Not recommended.
False	**False**	In	Normal layering, affecting only pixels that were previously covered or that appear when resizing a form.
False	**False**	Out	Graphics methods and all controls appear mixed in the three layers. Not recommended.

The Effects of AutoRedraw

Setting AutoRedraw to **True** always produces normal layering. While using AutoRedraw is the easiest way to layer graphics, applications with large forms may suffer from reduced performance due to the memory demands of AutoRedraw.

The Effects of ClipControls

When AutoRedraw is **True**, the setting of ClipControls has no effect on how graphics layer on a form or in a container. But ClipControls can affect the speed with which the form displays. When ClipControls is **False**, the application doesn't create a clipping region. Not having to calculate or paint to avoid holes in a clipping region may cause the form to display faster.

Also, when AutoRedraw and ClipControls are both **False**, the application repaints only the pixels of a form or container that are exposed by:

- Covering the form or container with another window and then moving the window away.
- Resizing the form or container.

The Effects of the Paint Event

When AutoRedraw is **False**, the best place to use graphics methods is within the Paint event of the form or container. Confining graphics methods to the Paint event causes those methods to paint in a predictable sequence.

Using graphics methods outside a Paint event when AutoRedraw is **False** can produce unstable graphics. Each time the output of a graphics method appears on the form or container, it may cover any controls or graphics methods already there (if ClipControls is **False**). When an application uses more than a few graphics methods to create visual effects, managing the resulting output can be extremely difficult unless the methods are all confined to the Paint event.

Using Drawing Properties

So far, you've drawn fairly standard lines, using a small width, solid pattern, and foreground color. This section shows how to vary the properties of lines to create more interesting effects. Properties that affect how lines are drawn are:

- Color argument
- DrawWidth
- BorderWidth
- DrawStyle
- BorderStyle
- DrawMode

Using the Color Argument

To vary the color of the line, use the optional color argument. For example, this statement draws a dark blue line:

```
Line (500, 500)-(2000, 2000), RGB(0, 0, 255)
```

If you omit the color argument, the ForeColor property for the object where the line is being drawn determines its color.

Using DrawWidth and BorderWidth

The DrawWidth property specifies the width of the line in pixels for the **Line** method and line controls. DrawWidth also affects the **PSet** and **Circle** methods. For example, the following procedure draws lines of several different widths. The results are shown in Figure 15.30.

```
Sub Form_Click ()
    DrawWidth = 1
    Line (100, 1000)-(3000, 1000)
    DrawWidth = 5
    Line (100, 1500)-(3000, 1500)
    DrawWidth = 8
    Line (100, 2000)-(3000, 2000)
End Sub
```

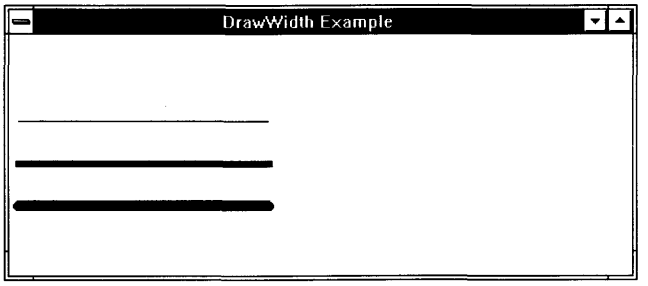


Figure 15.30 The effects of changing the DrawWidth property

The BorderWidth property specifies the outline thickness of shape controls. Figure 15.31 shows three shape controls with different BorderWidth values.

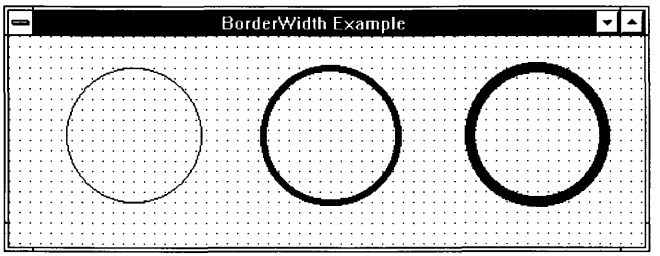


Figure 15.31 The effects of changing the BorderWidth property

Using DrawStyle and BorderStyle

The DrawStyle property specifies whether the lines created with the **Line** method or a line control are solid or have a broken pattern. The BorderStyle property of a shape control serves the same function as the DrawStyle property.

Note Forms, picture boxes, labels, text boxes, image controls, and shape controls all have a BorderStyle property. The BorderStyle property of a shape control serves a different purpose and uses different settings than the BorderStyle property in other controls and in forms. For shape controls, the BorderStyle property works like the DrawStyle property as described in this section. For forms and other controls, the BorderStyle property determines whether the control or form has a border and if so, whether the border is fixed or sizable.

Solid and Inside Solid Styles

The inside solid style (DrawStyle or BorderStyle = 6) is nearly identical to the solid style. They both create a solid line. The difference between these settings becomes apparent when you use a wide line to draw a box or a shape control. In these cases, the solid style draws the line half inside and half outside the box or shape. The inside solid style draws the line entirely inside the box or shape. See the section "Drawing Boxes," earlier in this chapter, to see how to draw a box.

Example

The following procedure demonstrates all of the supported settings of the DrawStyle property by creating a loop in which the setting goes from 0 to 6, one step at a time. The results are shown in Figure 15.32.

```
Sub Form_Click ()
    Dim I As Integer, Y As Long
    For I = 0 To 6
        DrawStyle = I
        Y = (200 * I) + 1000
        Line (200, Y)-(2400, Y)
    Next I
End Sub
```

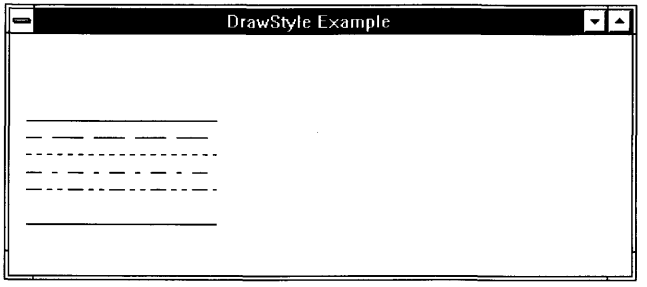


Figure 15.32 The effects of changing the DrawStyle property

For More Information For information about the DrawStyle and BorderStyle property settings, see the *Language Reference*, or search Help for *DrawStyle* or *BorderStyle*.

Using DrawMode

The DrawMode property determines what happens when you draw one pattern on top of another. Although changing the DrawMode property usually has some effect (especially with color systems), it is often not necessary to use this property when you are drawing on a blank or pure white background, or on a background of undifferentiated color.

You can set DrawMode to a value from 1 to 16. Common settings appear in the following table.

Setting	Setting name	Description
4	Not Copy Pen	Draws the inverse of the line pattern, regardless of what is already there.
7	Xor Pen	Displays difference between line pattern and the existing display, as explained later in this section. Drawing an object twice with this mode restores the background precisely as it was.
11	Nop	No operation. In effect, this turns drawing off.
13	Copy Pen (default)	Applies the line's pattern, regardless of what is already there.

For More Information For information on the DrawMode property, see the *Language Reference*, or search Help for *DrawMode*.

The Xor Pen

A DrawMode setting of 7 is useful for animation. Drawing a line twice restores the existing display precisely as it was before the line was drawn. This makes it possible to create one object that "moves over" a background without corrupting it, because you can restore the background as you go. Most modes are not guaranteed to preserve the old background.

For example, the following code moves a circle every time the mouse is clicked. No matter what pattern was underneath the circle, it gets restored.

```
Sub Form_Click ()
    ForeColor = 255: DrawMode = 7
    Circle (CurrentX, CurrentY), 1000
    CurrentX = CurrentX + 220
    CurrentY = CurrentY + 220
    Circle (CurrentX, CurrentY), 1000
End Sub
```

The Xor Pen draw mode (and most of the other DrawMode settings) works by comparing each individual pixel in the draw pattern (called the "Pen") and the corresponding pixel in the existing area (called the "Destination"). On monochrome systems, the pixel is turned either on or off, and Visual Basic performs a simple logical comparison: It turns a pixel on if either the Pen or Destination pixel is on, but not if both are on.

In color systems, each pixel is assigned a color value. For DrawMode settings such as Xor Pen, Visual Basic compares each corresponding pair of pixels in the Pen and Destination and performs a binary (bitwise) comparison. The result determines the color value of the resulting pixel, as shown in Figure 15.33.

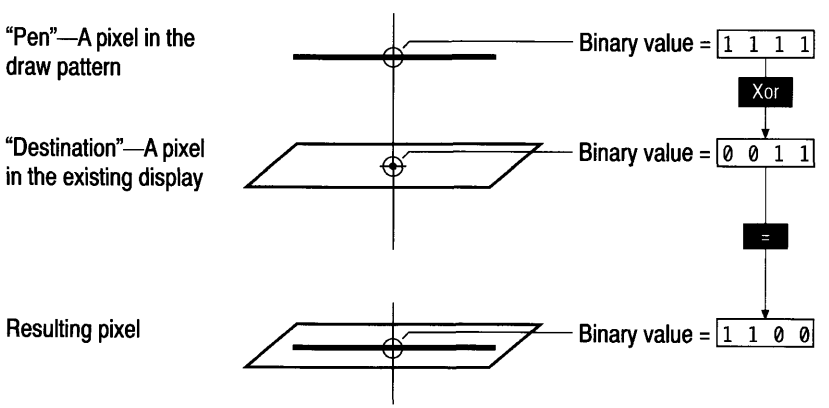


Figure 15.33 Using the Xor Pen to set the binary value of a pixel in a line

Using Color Properties

Many of the controls in Visual Basic have properties that determine the colors used to display the control. In this section, we discuss these color properties and how they apply to the controls and methods in this chapter. Keep in mind that some of these properties also apply to controls other than the graphical controls. The following table describes the color properties.

Property	Description
BackColor	Sets the background color of the form or control used for drawing. If you change the BackColor property after using graphics methods to draw, the graphics are erased by the new background color.
ForeColor	Sets the color used by graphics methods to create text or graphics in a form or control. Changing ForeColor does not affect text or graphics already created.
BorderColor	Sets the color of the border of a shape control.
FillColor	Sets the color that fills circles created with the **Circle** method and boxes created with the **Line** method.

For More Information For detailed descriptions of these color properties, search Help for *BackColor*, *ForeColor*, *BorderColor*, and *FillColor*.

Defining Colors

The color properties can use any of several methods to define the color value. In the section "Using Simple Color," we looked at using the **RGB** and **QBColor** functions as two of the methods you can use to define colors. In this section we add two methods for defining colors:

- Using defined constants
- Using direct color settings

Using Defined Constants

You don't need to understand how color values are generated if you use the constants from the file CONSTANT.TXT. Just copy these definitions into a module. For example, the following statement defines the symbolic constant RED:

```
Global Const RED = &HFF&
```

With this definition in a module, you can use the symbolic name RED whenever you want to specify red as a color argument or color property setting. For example:

```
BackColor = RED
```

The CONSTANT.TXT file also provides constants for the 21 system colors used for painting the various parts of the display. These values tell VB to use the colors selected for Windows screen elements in the Control Panel colors dialog. Using such numbers causes your application to appear differently on each user's system, but the advantage is that the application can be made to look more consistent with other Windows–based applications for any given user.

For More Information For information on the CONSTANT.TXT file, search Help for *constants*.

Using Direct Color Settings

All the methods discussed so far to define color are indirect methods. They are indirect because Visual Basic interprets them into the single approach it uses to represent color. If you understand how colors are represented in Visual Basic, you can assign numbers to color properties and arguments that specify color directly. In most cases, it's much easier to enter these numbers in hexadecimal.

The valid range for a normal RGB color is 0 to 16,777,215 (&HFFFFFF&). Each color setting (property or argument) is a four-byte integer. The high byte of a number in this range equals 0. The lower three bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The Red, Green, and Blue components are each represented by a number between 0 and 255 (&HFF).

Consequently, you can specify a color as a hexadecimal number using this syntax:

&HBBGGRR&

The *BB* specifies the amount of blue, *GG* the amount of green, and *RR* the amount of red. Each of these fragments is a two-digit hexadecimal number from 00 to FF. The median value is 80. Thus, the following number specifies gray, which has the median amount of all three colors:

```
&H808080&
```

Setting the most significant bit to 1 changes the meaning of the color value: It no longer represents an **RGB** color, but an environment-wide color specified through the Control Panel. The values that correspond to these system-wide colors range from &H80000000 to &H80000012 and are included in the CONSTANT.TXT file.

Using 256 Colors

Visual Basic supports 256 colors on systems with video adapters and display drivers that handle 256 colors. The ability to display 256 simultaneous colors is particularly valuable in multimedia applications or applications that need to display near–photographic-quality images.

You can display 256-color bitmaps and define up to 256 colors for graphics methods in:

- Forms
- Picture boxes
- Image controls

Note Support for 256 colors does not apply to Windows metafiles. Visual Basic displays metafiles using the default palette of 16 VGA colors.

The Fundamentals of Color Palettes

Color palettes provide the basis for 256-color support in Visual Basic applications. Much like the palettes that painters use to hold colors that they might paint on a canvas, Windows uses palettes to keep track of the colors that applications might display on the screen. The palette that Windows uses to support the color needs of running applications is called the *system palette.*

Note The color palettes discussed in this section are not the same as the Color palette you display by choosing Color Palette from the Window menu. In this section, the term "Color palette" refers to a table listing as many as 256 colors. Color palettes can contain fewer than 256 colors.

The Default VGA Palette

The default VGA color palette for most video adapters and display drivers uses 16 standard colors. When an application for Windows using the default VGA color palette needs to display a color on the screen, it can use any of the 16 solid colors in the palette or create a dithered color.

Dithering

Dithering is a process used to simulate colors not available from the video adapter and display driver. The system simulates unavailable colors by creating patterns of pixels with colors from the system palette. For example, on a monochrome system, shades of gray can be simulated by dithering various patterns of black-and-white pixels. Colors outside the default VGA palette can be simulated by dithering standard VGA colors.

Color Palettes

On systems with video adapters and display drivers that display 256 simultaneous colors on the screen, the Windows operating system uses a much larger color palette than the standard VGA palette. The larger palette can hold up to 256 colors.

Twenty of these 256 colors, called *static* colors, are reserved by the system and cannot be changed by an application. Static colors include the 16 colors in the default VGA palette plus four additional colors chosen to create a good blend of standard colors. The system palette always contains these static colors.

Because the static colors are reserved, you can specify up to 236 additional colors for use in your applications. Windows adds any extra colors requested by an application to the system palette until all 256 colors in the system palette are used. The process of filling the system palette, which can affect the way your application uses color, is described later in this chapter.

Displaying 256-Color Bitmaps

Forms, picture boxes, and image controls automatically display 256-color bitmaps in 256 colors if the user's display hardware and software can support that many colors on screen. If the user's system supports only the default VGA palette, then Visual Basic applications will dither nondefault colors using the colors in the default palette.

Each bitmap contains its own color palette, called a logical palette. A *logical palette* defines the colors needed to accurately display the colors in the bitmap. A 256-color bitmap has a logical palette that can contain up to 256 simultaneous colors. When displaying a 256-color bitmap, Windows uses colors from the logical palette of the bitmap to fill the empty spots in the system palette.

Drawing with Color Palettes

Using color with graphics methods to draw on forms or inside picture boxes normally limits you to the standard 16 VGA colors, but you can use up to 256 colors with graphics methods. To do this, load a 256-color bitmap into the form or picture box. The bitmap doesn't have to be very large; even a single pixel can define up to 256 colors for the form or picture box. This is because the logical palette of a bitmap can list up to 256 colors, regardless of whether all those colors appear in the bitmap.

Once the bitmap and its logical palette are loaded into the form or picture box, use the **RGB** function to specify colors for graphics methods. The color that actually appears on the screen is a product of:

- The color you specify
- The colors in the logical palette of the bitmap loaded into the container
- The colors in the system palette

The colors in the system palette determine the colors that appear on the screen. When you specify a color for a graphics method using the **RGB** function, Visual Basic matches that color to a color in the logical palette of the bitmap loaded into the form or picture box. This matched color must then be matched to the closest color in the system palette. The matching color in the system palette appears on the screen. Figure 15.34 illustrates this process of color matching.

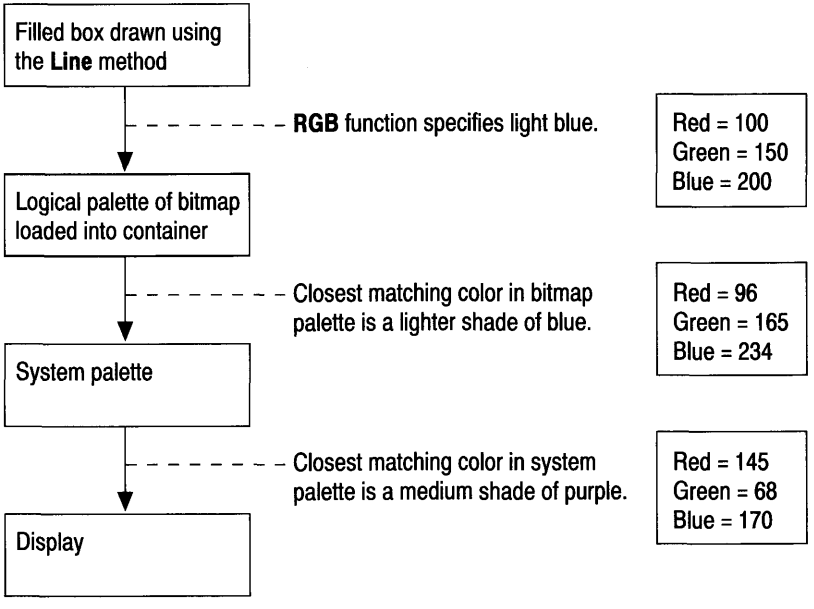


Figure 15.34 Color matching from a specified color to the display

Visual Basic ships with three bitmaps with color palettes you can load into forms and picture boxes. The following table describes these bitmaps.

Device-independent bitmap (.DIB) file	Palette description
RAINBOW.DIB	Standard range of all hues
PASTEL.DIB	Lighter hues, primarily blues
BRIGHT.DIB	Bright shades of all hues

Each of these bitmaps is only one pixel in size, so you can use their palettes while keeping the bitmaps themselves out of sight. Depending on the display resolution and pixel size of the user's system, these bitmaps might appear as a single dot in the form or picture box.

Note Because a form or picture box can contain only one bitmap at a time, if you load one of these device-independent bitmaps and then load another bitmap, the palette of the new bitmap controls the available colors. If you need to display a bitmap in a container, make sure the palette of that bitmap contains all the colors you need in its palette.

Defining Random Colors

The Picture property of the DemoForm of the Blanker application contains the RAINBOW.DIB bitmap. Because the PSetDemo (Confetti), LineDemo (Crossfire), CircleDemo (Rainbow Rug), and ScaleDemo (Color Bars) procedures use the **RGB** function to define their random colors, these demos display 256 colors on systems that support 256 colors on the screen.

Creating Your Own Color Palettes

You can create bitmaps with color palettes if you have an application that lets you edit bitmaps and their palettes. The Microsoft Multimedia Development Kit contains two applications, BitEdit and PalEdit, that you can use to create and edit bitmaps with color palettes.

Copying and Pasting Palettes to the Clipboard

You can use the **SetData** method to copy color palettes to the Clipboard and the **GetData** method to paste palettes from the Clipboard. Use the CF_PALETTE format with these methods.

For More Information For information on the **SetData** and **GetData** methods, search Help for *SetData* or *GetData*.

Using Multiple Color Palettes

When you work with color palettes, keep in mind that although the display hardware can often generate thousands of solid colors, it can display only 256 of them simultaneously on the screen.

This limitation becomes important when you use more than one color palette in your application. For example, on a single form, you might display a 256-color bitmap in an image control while drawing with a color palette in a picture box. If these two palettes don't use exactly the same 256 colors, Windows must decide which logical palette places its colors in the system palette first. Remember, the system palette determines what actually appears on the screen.

A similar situation occurs when you run a Visual Basic application that uses at least one color palette while you are running another Windows–based application that also uses a color palette. Any time a color palette in your application must compete with other color palettes—whether or not the competing palettes are in your application—Windows must determine what colors appear in the system palette.

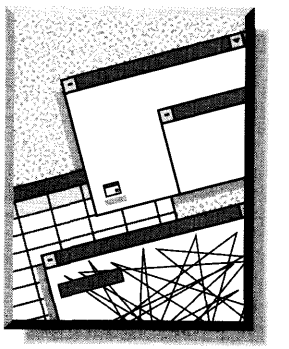
Color Palettes and Z-Order

Z-order is a relative ordering that determines how controls overlap each other on a form. Each time a form, image control, picture box, or application appears at the top of the z-order of the system, the logical palette of that object gets priority from Windows to fill the system palette. Windows clears all colors (except the static colors) from the system palette and fills the empty spots as needed by the application or object in the foreground. The logical palette of the application or object in the foreground is called the *foreground palette*. All other logical palettes are called *background palettes*.

Windows assigns each color in the foreground palette to empty spots in the system palette. This process continues until either all the colors in the foreground palette are in the system palette or the system palette is full. If there are still empty spots in the system palette after adding all the colors in the foreground palette, Windows adds additional colors from background palettes until there are no more background palettes or until the system palette is full.

CHAPTER 16

Displaying and Printing Information



As your applications become more sophisticated, you'll want to take advantage of the tools Visual Basic offers for displaying information in the most polished and meaningful form. These tools let you select fonts, format tables, and change number formats. You can also use them to send text and graphics to the printer using the Printer object.

Contents

- The Fundamentals of Fonts
- Setting Font Characteristics
- Displaying Text in Forms and Picture Boxes
- Displaying Tabular Data
- Formatting Numbers, Dates, and Times
- The Fundamentals of Printing
- Printing with the Printer Object
- Printing with the **PrintForm** Method

TIMECARD.MAK

Many of the code examples in this chapter are taken from the TimeCard sample application. If you installed the sample applications, you will find this application in the \PRINT subdirectory of the main Visual Basic directory (\VB\SAMPLES\PRINT).

The Fundamentals of Fonts

Although Visual Basic lets you create applications that rely heavily on graphics, text is still the most common way to display information. But effective use of text in an application involves more than deciding what the text should say. You must also select the fonts and font sizes. This section presents basic information about different fonts and the issues involved in choosing the best fonts for your application.

Screen and Printer Fonts

Fonts generally fall into one of these two categories:

- Screen fonts designed for display on computer monitors
- Printer fonts designed for use by printers to reproduce text on a printed page

The difference between screen fonts and printer fonts becomes important if you want your application to print. For many applications, the ability to print text as it appears on the screen (what you see is what you get, or WYSIWYG, printing) depends on a close match between the screen fonts used in the application and the printer fonts available to the printer.

Scalable and Nonscalable Fonts

Fonts can also be categorized as scalable or nonscalable. *Scalable* fonts are defined by mathematical routines that can reproduce the outlines of each character at any size. *Nonscalable* fonts, which reproduce each character in a bitmap, are built to specific font sizes and are therefore more limited than scalable fonts. There are scalable and nonscalable varieties of both screen and printer fonts.

Selecting Fonts for Your Application

When selecting fonts for your application, keep in mind that scalable fonts provide the greatest flexibility; you can use them with both the screen and the printer. For example, the TrueType™ fonts that ship with Windows version 3.1 can be used both for displaying text in your application and for printing. Because TrueType fonts are scalable, you can also use them in a wide range of sizes.

Remember, however, that a user of your application may not have the same fonts you used to create the application. For example, if you select a TrueType font that a user doesn't have, Windows selects the closest matching font on the user's system. Depending on the design of your application, this may cause problems for the user. For example, the font Windows selects may enlarge text so that labels overlap on the screen.

You should plan for these possibilities when selecting fonts. One way to avoid font problems is to distribute the necessary fonts with your application. (You will probably need to obtain permission from the copyright holder of the font to distribute it with your application.)

You can also program your application to check the fonts listed in the WIN.INI file for the fonts you use. If the font isn't there, program the application to use a font from the list.

Another way to avoid font problems is to use fonts users are most likely to have on their systems. If you use fonts from a specific version of Windows, you may have to specify that version as a system requirement of your application.

Setting Font Characteristics

Forms and most controls can display text. In controls, text is often available through the Caption or Text properties. Forms, controls that display text, and the Printer object introduced later in this chapter support font properties. Font properties determine the visual characteristics of the text, including:

- Font (typeface)
- Font size (in points)
- Special characteristics (bold, italics, underline, or strikethrough)

The effect of changing font properties varies, depending on the technique used to display text. If the text is specified by a property (such as Text or Caption), then changing a font property applies to all the text in that control. Labels, text boxes, frames, buttons, check boxes, and all the file-system controls use a property to specify text. None of them supports the **Print** method.

If the application displays text with the **Print** method, then changing a font property affects all uses of **Print** after the property change. Text printed before the property change is not affected. Only forms, picture boxes, and the Printer object support the **Print** method.

Because changes in font properties apply to all the text in text boxes and labels, you cannot mix fonts in these controls. If you need to mix fonts (for example, you want to make some words bold but leave others in normal font), then create a picture box and use the **Print** method to display text. The next section, "Displaying Text in Forms and Picture Boxes," explains how to use the **Print** method.

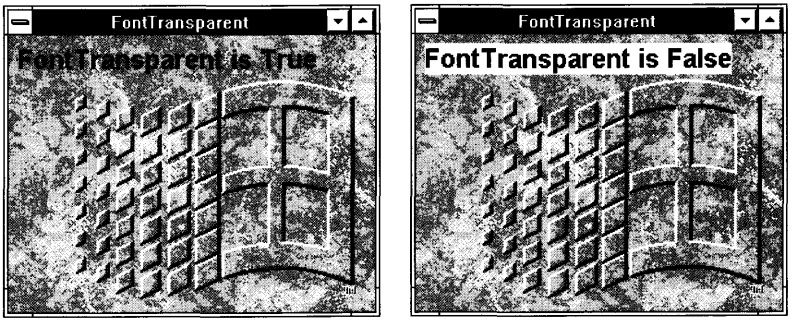
Font Properties

The six font properties are listed in the following table.

Property	Type	Description
FontName	String	Specifies name of font (for example, "Modern," "Courier," or "Script").
FontSize	Integer	Specifies font size in points (72 to an inch when printed).
FontBold	Boolean	If **True**, the text is bold.
FontItalic	Boolean	If **True**, the text is italic.
FontStrikethru	Boolean	If **True**, Visual Basic strikes through the text (for example, ~~text~~).
FontUnderline	Boolean	If **True**, the text is underlined.

The FontTransparent Property

Forms and picture boxes have an additional font property, FontTransparent. When FontTransparent is **True**, any text printed in the form or picture box avoids blocking out the background of the form or picture box. Figure 16.1 shows the effects of the FontTransparent property.



FontTransparent set to **True**

FontTransparent set to **False**

Figure 16.1 The effects of the FontTransparent property

Setting Font Properties

The settings available for each of the properties are listed in the Properties window. You can set any of the font properties at design time (by using the Properties window) or at run time. For example, the following statements set various font properties for a label named lblTotalHours:

```
lblTotalHours.FontName = "Modern"          ' Change font to Modern.
lblTotalHours.FontBold = True              ' Turn on bold.
lblTotalHours.FontItalic = True            ' Turn on italics.
```

The order of selection is important, since not all fonts support all font variations. Set the FontName property first. Then you can set any of the Boolean properties, such as FontBold and FontItalic, to **True** or **False**. For information on font properties, see the *Language Reference*, or search Help for *font formatting*.

Note When setting the FontSize property to sizes smaller than 8 points, Windows automatically changes to a different font if the selected font isn't supported in the smaller size. To avoid unpredictable results from this aspect of font management in Windows, set the FontSize property first when you use a font size smaller than 8 points. Set the FontName property next. Then set the FontSize property again, followed by additional font properties.

Example

The TimeCard application is an online time card used to clock in and out of work. It displays most of the text indicated by the arrays of labels in four columns, as described in the following table.

Column	Description	Control array name
First	Shows the day of the week.	lblDay()
Second	Shows the time you clocked in.	lblInTime()
Third	Shows the time you clocked out.	lblOutTime()
Fourth	Shows the total hours worked that day.	lblHours()

Figure 16.2 shows TimeCard with a week of time entered.

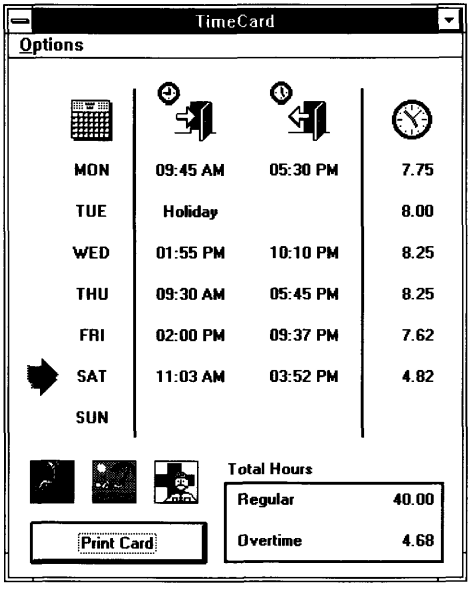


Figure 16.2 The TimeCard application

The default screen font used by TimeCard is MS Sans Serif, a nonscalable screen font supplied by Windows 3.1. If you print the information with MS Sans Serif as the screen font, the card prints in the printer's default font. The default printer font varies from printer to printer depending on the driver.

The Options menu of TimeCard has a Select Font command you can use to change the application's screen and printer fonts. Figure 16.3 shows the dialog that appears when you choose the Select Font command.

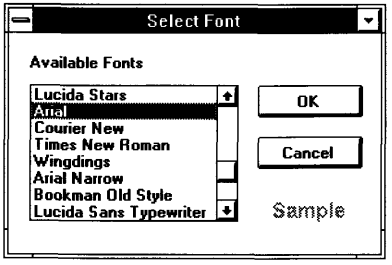


Figure 16.3 The Select Font dialog

The Select Font dialog lists fonts with names common to both the screen fonts and printer fonts available on the system. The GetFonts procedure builds the list of available fonts. The following code compares the screen font names to all the printer font names.

```
Sub GetFonts ()
    MatchCnt = 0                                    ' Initialize match counter.
    For PTarget = 0 To (Printer.FontCount - 1)      ' For each printer font name...
        For Match = 0 To (Screen.FontCount - 1)     ' For each screen font name...
            If Printer.Fonts(PTarget) = Screen.Fonts(Match) Then
                ' If the names match, then increment the match counter,
                ' resize the array to store the name, and save font name.
                MatchCnt = MatchCnt + 1
                ReDim Preserve BothFonts(MatchCnt)
                BothFonts(MatchCnt - 1) = Printer.Fonts(PTarget)
                Match = Screen.FontCount - 1        ' Exit the loop and get the next font.
            End If
        Next Match                                  ' Check next screen font for match.
    Next PTarget                                    ' Compare the next printer font.
End Sub
```

Displaying Text in Forms and Picture Boxes

You can cause text to appear directly on a form. When you print to a form, though, the text appears in a layer *behind* any controls that have been placed on the form. So printing to a form usually works best on a form specifically created to hold the text. For more information about how text and graphics appear in layers on a form, see the section "Layering Graphical Controls and Graphic Methods" in Chapter 15, "Creating Graphics for Applications."

You can also print text to a picture box control. A *picture box* is a simple rectangular area into which you can load or copy a bitmap, draw graphics, or print text. One advantage to using a picture box is that it gives you a well-defined area separate from other controls to display text and graphics.

Using the Print Method

To print on a form or picture box, use the **Print** method, preceded by the name of a form, a picture box, the Printer object, or the Debug object. (See the section "Printing from Application Code" in Chapter 9, "Debugging," for more information about the Debug object.) The **Print** method syntax is:

[*object.*]**Print** [*expressionlist*] [{ ; | , }]

The *object* is optional; if omitted, the **Print** method applies to the form to which the code is attached (the current form). For example, the following statements print messages to:

- A form named MyForm:

```
MyForm.Print "This is a form."
```

- A picture box named picMiniMsg:

```
picMiniMsg.Print "This is a picture box."
```

- The current form:

```
Print "This is the current form."
```

- The Printer object:

```
Printer.Print "This text is going to the printer."
```

The *expressionlist* is the text that appears on the form or picture box. Multiple items in the *expressionlist* must be separated by commas or semicolons or both, as explained in the next section.

If the form or picture box is too small to display all the text, the text is cut off. Where the form or picture box cuts off the text depends on the coordinates of the location at which you began printing the text. The form or picture box does not scroll.

Combining Items on a Line

The items you print can include property values, constants, and variables (either string or numeric). The **Print** method prints the value of numeric items. Positive number values have a leading and a trailing space. Negative numeric values display their sign instead of a leading space.

Use a semicolon (;) or a comma (,) to separate one item from the next. If you use a semicolon, Visual Basic prints one item after another, without intervening spaces. If you use a comma, Visual Basic skips to the next print zone.

For example, the following statement prints to the current form:

```
Print "The value of X is "; X; "and the value of Y is "; Y
```

If X contains the value 2 and Y contains the value 7, the statement produces this output:

```
The value of X is 2 and the value of Y is 7
```

By default, each **Print** method prints the text and moves to the next line. If there are no items, **Print** simply skips a line. A series of **Print** statements (in the following example, for a picture box named picLineCount) automatically uses separate lines:

```
picLineCount.Print "This is line 1."
picLineCount.Print "This is line 2."
```

By placing a semicolon (or comma) at the end of the first statement, however, you cause the output of the next **Print** statement to appear on the same line:

```
picLineCount.Print "This all appears ";
picLineCount.Print "on the same line."
```

Referring to Specific Locations

You can control placement of **Print** output by either or both of these techniques:

- Erasing a given form or picture box with the **Cls** (clear) method
- Setting drawing coordinates with the CurrentX and CurrentY properties

  The drawing coordinates determine where the next **Print** or graphics operation begins to place its output.

The Cls Method

The **Cls** method erases all text and graphics created in the object with previous **Print** and graphics methods. The **Cls** method also resets the drawing coordinates to (0,0), which is the upper-left corner by default. For example, these statements clear:

- A picture box named Picture1:

```
Picture1.Cls
```

- The current form:

```
Cls
```

The CurrentX and CurrentY Properties

You can set the drawing coordinates of forms and picture boxes directly with the CurrentX and CurrentY properties. For example, these statements reset the drawing coordinates to the upper-left corner for Picture1 and for the current form:

- A picture box named Picture1:

```
Picture1.CurrentX = 0
Picture1.CurrentY = 0
```

- The current form:

```
CurrentX = 0
CurrentY = 0
```

Any new text you print appears on top of any text and graphics already at that location. To erase text selectively, draw a box with the **Line** method and fill it with the background color. Keep in mind that the drawing coordinates specified by CurrentX and CurrentY usually change location when you use a graphics method.

For more information on drawing coordinates, see the section "Using Graphics Methods" in Chapter 15, "Creating Graphics for Applications."

By default, forms and picture boxes use a coordinate system in which each unit corresponds to a twip (1,440 twips equal an inch, and approximately 567 twips equal a centimeter). You may want to change the ScaleMode of the form, picture box, or Printer object to points, since this is the unit of measure used to indicate the height of text. Using the same unit of measure for the text and for the object where you will print the text makes it easier to calculate the position of the text. See the section "Using Twips" in Chapter 15, "Creating Graphics for Applications," for more information about twips.

The TextHeight and TextWidth Methods

Before using the **Print** method, you can use the **TextHeight** and **TextWidth** methods to determine where to position the CurrentX and CurrentY properties. **TextHeight** returns the height of a line of text, taking into account the object's font size and style. The syntax is:

[*object.*]**TextHeight**(*stringexpression*)

If the string contains embedded carriage-return characters (**Chr$**(13)), then the text corresponds to multiple lines, and **TextHeight** returns the height of the number of lines of text contained in the string. If there are no embedded carriage returns, **TextHeight** always returns the height of one line of text.

One way to use the **TextHeight** method is to set the CurrentY property to a particular line. For example, the following statements set the drawing coordinates to the beginning of the fifth line:

```
CurrentY = TextHeight("sample") * 4
CurrentX = 0
```

Assuming there are no carriage returns in the sample text, you would use this syntax to set CurrentY to the *nth* line:

CurrentY = [*object.*]**TextHeight**(*sampletext*) * (*n* – *1*)

The brackets indicate that *object* is optional; if *object* is omitted, the method applies to the current form. The *object* can be a form, a picture box, or the Printer object.

The **TextWidth** method returns the width of a string, taking into account the object's font size and style. This method is useful since many fonts have proportional-width characters. The **TextWidth** method helps you determine whether the width of the string is larger than the width of the form, picture box, or Printer object.

For example, the following statements use **TextWidth** and **TextHeight** to center the text "TimeCard" in a box by positioning CurrentX and CurrentY:

```
CurrentX = (BoxWidth - TextWidth("TimeCard")) / 2
CurrentY = (BoxHeight - TextHeight("TimeCard")) / 2
```

For More Information For information on the **TextHeight** and **TextWidth** methods, see the *Language Reference*, or search Help for *Text Height* and *TextWidth*.

Displaying Tabular Data

The **Print** method offers two ways to print tables of information: You can use preset tabs, or you can set up your own tabs. The preset tabs provided by the **Print** method are especially easy to use. If the data you're printing is longer than about 13 characters, however, you should set your own tabs.

Using Built-in Tabs

The easiest way to print tables is to use the print zones built into Visual Basic. Each *print zone* is 14 columns wide, and each column is the width of the average character in the font and font size you specify.

When you insert a comma between two items in a **Print** method, the second item starts printing at the next print zone. The following statements use built-in tabs to display text on a form:

```
Form1.FontName = "MS Sans Serif"
Form1.FontSize = 10
Form1.Print "First name", "Last name", "Phone number"
Form1.Print
Form1.Print "Jonathan", "Vanderbilt", "555-4444"
Form1.Print "Arlene", "Lillegard", "555-7891"
Form1.Print "Joanne", "Buchanan", "99-11-12345"
```

Figure 16.4 shows the results of the preceding statements on the form in 10-point MS Sans Serif font.

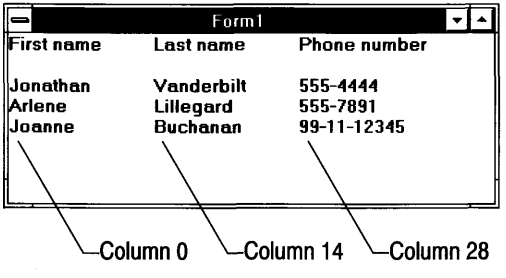


Figure 16.4 Using built-in print zones

Like most Windows fonts, MS Sans Serif is a proportional font and includes characters of varying widths. In Figure 16.4, each print zone has the same width as 14 average MS Sans Serif 10-point characters. When setting up tables, use the widest character in the text to calculate the minimum column width necessary for your application.

Setting Up Your Own Tabs

For more control over the appearance of tables, you can set your own tab positions. To specify a tab position, call the **Tab** function and include the position as one of the items in a **Print** statement. The syntax is:

Tab(*column*);

For example, the following statements print a message in the MS Sans Serif font starting at column 5 (each column is the width of the average character in the selected font):

```
Picture1.FontName = "MS Sans Serif"
Picture1.Print Tab(5); "This message starts in column 5."
```

Figure 16.5 shows a table printed to a form. The first tab, at column 2, indents the lines slightly from the left edge of the picture box. The tab at column 22 sets the position for the "Description" heading and the text printed below it.

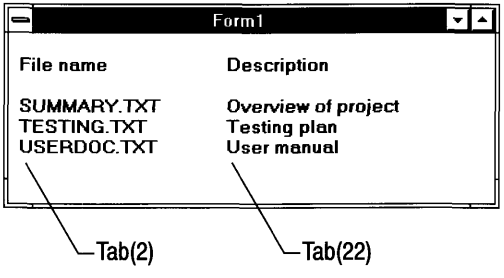


Figure 16.5 Using tabs in a table

The following code prints this table on the form:

```
Form1.FontName = "MS Sans Serif"
Form1.FontSize = 10
Form1.Print
Form1.Print Tab(2); "File name"; Tab(22); "Description"
Form1.Print
Form1.Print Tab(2); "SUMMARY.TXT"; Tab(22); "Overview of project"
Form1.Print Tab(2); "TESTING.TXT"; Tab(22); "Testing plan"
Form1.Print Tab(2); "USERDOC.TXT"; Tab(22); "User manual"
```

For More Information For more information on the **Tab** function and its use with the **Print** method, see the *Language Reference*, or search Help for *Tab*.

Formatting Numbers, Dates, and Times

Visual Basic gives you great flexibility in displaying number formats as well as date and time formats. You can easily display international formats for numbers, dates, and times.

The Format$ and Format Functions

The **Format$** function converts a numeric value into a string and at the same time gives you control over the appearance of the string. For example, you can specify the number of decimal places, leading or trailing zeros, and currency formats. The **Format** function converts a numeric value into the **Variant** data type. For more information on the **Variant** data type, see Chapter 7, "Variables, Constants, and Data Types."

The **Format$** and **Format** functions share this common syntax:

**Format[$]**(*numeric-expression*[, *fmt$*])

Format$ and Format Syntax

The argument *numeric-expression* specifies a number to convert, and *fmt$* is a string made up of symbols that show how to format the number. The most commonly used symbols are listed in the following table.

Symbol	Description
0	Digit placeholder; prints a trailing or a leading zero in this position if appropriate.
#	Digit placeholder; never prints trailing or leading zeros.
.	Decimal placeholder.
,	Thousands separator.
– + $ ( ) space	Literal character; displays each of these characters exactly as typed into the format string.

Named Formats

Visual Basic provides several standard formats you can use with the **Format$** or **Format** functions. Instead of using symbols, you specify these formats by name in the *fmt$* portion of the **Format$** or **Format** syntax. Always enclose the format names in double quotation marks ("").

The following table lists the format names you can use.

Named format	Description
General Number	Displays numbers as entered.
Currency	Uses the symbol, symbol placements, and negative number format specified in the [INTL] section of WIN.INI. Displays negative numbers inside parentheses.
Fixed	Shows at least one digit.
Standard	Uses a thousands separator.
Percent	Multiplies the value by 100 with a percent sign at the end.
Scientific	Uses standard scientific notation.
General Date	Displays date and time if *numeric-expression* contains both. If the *numeric-expression* is only a date or a time, the missing information is not displayed. Examples: 4/3/93 05:34 PM 4/3/93 05:34 PM
Long Date	Uses the Long Date format specified in the International dialog of the Windows Control Panel. Example: Saturday, April 4, 1993
Medium Date	Uses the "dd-mmm-yy" format. Example: 03-Apr-93
Short Date	Uses the Short Date format specified in the International dialog of the Windows Control Panel. Example: 4/3/93
Long Time	Displays the hour, minute, second and "AM" or "PM" using the "ttttt" format. Example: 5:34:23 PM
Medium Time	Displays the hour, minute, and "AM" or "PM" using the "hh:mm AM/PM" format. Example: 05:34 PM
Short Time	Displays the hour and minute using the "hh:mm" format. Example: 05:34
Yes/No	Any nonzero numeric value (usually –1) is Yes. Zero is No.
True/False	Any nonzero numeric value (usually –1) is True. Zero is False.
On/Off	Any nonzero numeric value (usually –1) is On. Zero is Off.

Format$ and **Format** support many other special characters, such as the percentage placeholder and exponents.

For More Information For information on the **Format$** and **Format** functions, see the *Language Reference*, or search Help for *Format$* or *Format*.

The following number conversions assume that the country in the Windows Control Panel is set to "United States."

Format$ syntax	Result
Format$(8315.4, "00000.00")	08315.40
Format$(8315.4, "#####.##")	8315.4
Format$(8315.4, "##,##0.00")	8,315.40
**Format$**(315.4,"$##0.00")	$315.40

The symbol for the decimal separator is a period (.), and the symbol for the thousands separator is a comma (,). But the character that's actually displayed as a decimal separator or a thousands separator depends on the country set in the Windows Control Panel.

Country	Format$ syntax	Result
Sweden	**Format$**(7300, "0,000.00")	7.300.00
United Kingdom	**Format$**(7300, "0,000.00")	7,300.00
Germany	**Format$**(7300, "0,000.00")	7.300,00

Date and Time Formats

To print date and time formats, use the **Format$** function with symbols representing date and time. These examples use the **Now** and **Format$** functions to identify and format the current date and time. The following examples assume that the country in the International dialog of the Windows Control Panel is set to "United States."

Format$ syntax	Result
Format$(Now, "m/d/yy")	1/27/93
Format$(Now, "dddd, mmmm dd, yyyy")	Wednesday, January 27, 1993
Format$(Now, "d-mmm")	27-Jan
Format$(Now, "mmmm-yy")	January-93
Format$(Now, "hh:mm AM/PM")	07:18 AM
Format$(Now, "h:mm:ss a/p")	7:18:00 a
Format$(Now, "d-mmmm h:mm")	3-January 7:18

By using the **Now** function with the format "ddddd" and "ttttt," you can print the current date and time in a format appropriate for the country specified in the International dialog of the Windows Control Panel.

Country	Format$ syntax	Result
Sweden	**Format$(Now**, "ddddd ttttt")	1992-12-31 18.22.38
United Kingdom	**Format$(Now**, "ddddd ttttt")	31/12/92 18:22:38
Canada (French)	**Format$(Now**, "ddddd ttttt")	92-12-31 18:22:38
United States	**Format$(Now**, "ddddd ttttt")	12/31/92 6:22:38 PM

The TimeCard Application

The TimeCard application uses two different time formats, one to calculate the hours worked each day and another to display clock-in and clock-out times.

To subtract clock-in times from clock-out times, TimeCard needs to use time in a 24-hour format that specifies only hours and minutes. The Declarations section of the Card form specifies this format as a constant:

```
Const HrFormat = "hh:mm"
```

Note Using a constant makes it easier to change all the instances where TimeCard uses this format by changing the format in the Declarations section.

To display clock-in times on the Card form, TimeCard uses one of the named formats, as shown in the following statement:

```
lblInTime(Index).Caption = Format$(InTime, "Medium Time")
```

For More Information For information on date and time functions, see the *Language Reference*, or search Help for *Now*.

The Fundamentals of Printing

Printing from your application involves these three components:

- The code in your application that starts the printing process
- The Windows printer drivers installed on both your system and the systems of users of your application
- The capabilities of the printers available to users of your application

The code in your application determines the type and quality of print output available from your application. But the users' printer drivers and printers also contribute to print quality.

Getting the Best Results

Printing is one of the most complex tasks a Windows–based application performs. Good results depend on all parts of the process working together. Poor results can arise from problems in your application, variations in printer drivers, or limited printer capabilities. While it is a good idea to test your application with commonly used printers and printer drivers, you can't test all the possible combinations users may have.

Printing from an Application

Visual Basic provides two techniques for printing text and graphics to the default printer (specified by the user in the Control Panel):

- You can send text and graphics to the Printer object and then print them using the **NewPage** and **EndDoc** methods.
- You can produce the output you want on a form and then print the form using the **PrintForm** method.

This section examines the advantages and disadvantages of these two techniques.

Using the Printer Object

The *Printer object* is a device-independent drawing space that supports the **Print**, **PSet**, **Line**, and **Circle** methods to create text and graphics. You use these methods on the Printer object just as you would on a form or picture box. The Printer object also has all the font properties described earlier in this chapter. When you finish placing the information on the Printer object, you use the **EndDoc** method to send the output to the printer. When applications close, they automatically use the **EndDoc** method to send any pending information on the Printer object.

The TimeCard application demonstrates one way to use the Printer object to redraw a data entry form and its contents on the Printer object and then print the duplicate of the form on a printer.

Advantages of the Printer Object

The Printer object provides the best print quality across a variety of printers because Windows translates text and graphics from the device-independent drawing space of the Printer object to best match the resolution and abilities of the printer.

The Printer object also allows you to print multiple-page documents with the **NewPage** method. The Printer object tracks page numbers, which you can use to create numbered page headers or footers.

Disadvantages of the Printer Object

The main drawback to the Printer object is the amount of code required to get the best results. For example, most of the general procedures in the TimeCard application are devoted to getting text and graphics onto the Printer object. Printing bitmaps on the Printer object also takes time and can therefore slow the performance of the application.

Using the PrintForm Method

The **PrintForm** method sends a pixel-by-pixel image of the specified form to the printer. To use **PrintForm** to print information from your application, you must first display that information on a form and then print that form with the **PrintForm** method.

The **PrintForm** method is by far the easiest way of printing from your application. Because it sends information to the printer at the resolution of the user's screen (typically 96 dots per inch), however, results can be disappointing on printers with much higher resolutions (typically 300 dots per inch for laser printers).

Printing with the Printer Object

There are several ways to get text and graphics on the Printer object. This section explores these methods and provides examples from the TimeCard application.

▶ **To print with the Printer object**

1. Put text and graphics on the Printer object.
2. Print the contents of the Printer object with the **NewPage** or **EndDoc** method.

Characteristics of the Printer Object

The characteristics of the Printer object match those of the default printer setup in the Windows Control Panel. These characteristics include:

- Page size
- Page orientation
- Font selection

The size and orientation of the page is given by the Height and Width properties of the Printer object. The ScaleHeight and ScaleWidth properties define the size of the printable area of the page.

Changing the default printer in the Control Panel causes the Printer object to adapt to the characteristics of the default printer.

Scale Properties

The Printer object has these scale properties:

- ScaleMode
- ScaleLeft and ScaleTop
- ScaleWidth and ScaleHeight

The ScaleLeft and ScaleTop properties define the x- and y-coordinates, respectively, of the upper-left corner of the printable page. By changing the values of ScaleLeft and ScaleTop, you can create left and top margins on the printed page. The TimeCard application changes ScaleLeft and ScaleTop to center the printed time card on the page using these statements:

```
Printer.ScaleLeft = -((Printer.Width - PFrm.Width) / 2)
Printer.ScaleTop = -((Printer.Height - PFrm.Height) / 2)
```

Positioning Text and Graphics

You can set CurrentX and CurrentY properties for the Printer object, just as you can for forms and picture boxes. With the Printer object, these properties determine where to position output on the current page. The following statements set drawing coordinates to the upper-left corner of the current page:

```
Printer.CurrentX = 0
Printer.CurrentY = 0
```

You can also use the **TextHeight** and **TextWidth** methods to help position text on the Printer object. See the section "The TextHeight and TextWidth Methods," earlier in this chapter, for more information on using these text methods.

Printing Forms on the Printer Object

You may want your application to print one or more forms along with information on those forms, especially if the design of the form corresponds to a printed document like an invoice or a time card. The easiest way to do this is to use the **PrintForm** method. For the best quality on a laser printer, however, you should use the Printer object. Keep in mind that the Printer object takes more planning, since you must recreate the form on the Printer object before you print.

Recreating a form on the Printer object may also require recreating:

- The outline of the form, including title and menu bars.
- The controls and their contents, including text and graphics.
- The output of graphics methods applied directly to the form, including the **Print** method.

The extent to which you recreate these elements on the Printer object depends on your application and how much of the form you need to print. For example, the TimeCard application includes a series of general procedures that recreate selected portions of the data entry form on the Printer object. These procedures draw:

- The outline and title bar, but not the menu bar.
- All the label controls.
- The Print Card command button.
- Lines that separate areas of information.
- Icons that serve as column headings, but not the holiday, vacation, or sick time icons.

Figure 16.6 shows how the Card form of the TimeCard application looks when printed.

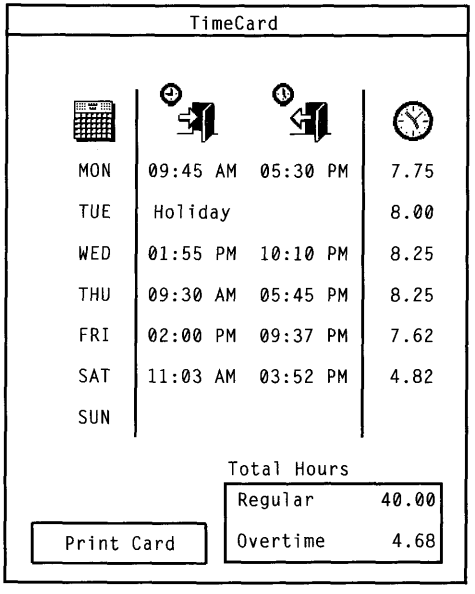


Figure 16.6 The Card form of the TimeCard application

The code needed to print the time card from the TimeCard application is divided into the following general procedures.

Procedure name	Description
PrintFrm	Takes a form as an argument. Centers the form on the Printer object, draws the outline and title bar, and then finds and prints all command buttons, labels, and picture box controls on the form.
DrawCmd	Takes a command button control as an argument. Draws a box with the same size and position as the command button passed to it. Centers and prints the caption of the command button in the box.
DrawLbl	Takes a label control as an argument. Prints the caption using the same position, font, and alignment as the label passed to it.
DrawPic	Takes a picture box control as an argument. Prints the pixels inside the picture box in the same location on the Printer object.
LinesOnPrinter	Contains code (similar to the **Line** methods used on the form) that redraws those form lines on the Printer object.
SetClientPrintOrigin	Changes the ScaleLeft and ScaleTop properties of the Printer object to position the (0, 0) coordinates at the upper-left corner of the Card form's client area.

The next few sections of this chapter present some of these procedures in detail.

The PrintFrm Procedure

Printing in the TimeCard application begins when the user clicks the Print Card command button. The Click event procedure for this command button calls the PrintFrm procedure, which consists of the following statements.

```
Sub PrintFrm (PFrm As Form)
    ' Change cursor to hourglass.
    PFrm.MousePointer = 11
    ' Set font size for printer.
    Printer.FontSize = 8.25
    ' Move the (0, 0) on the Printer object to center form on the page.
    Printer.ScaleLeft = -((Printer.Width - PFrm.Width) / 2)
    Printer.ScaleTop = -((Printer.Height - PFrm.Height) / 2)
    ' Set line thickness.
    DrawWidth = 2
    ' Draw form border.
    Printer.Line (0, 0)-Step(PFrm.Width, PFrm.Height), , B
    ' Draw bottom of title bar.
    Printer.Line (0, BarHgt)-Step(PFrm.Width, 0)
    ' Move X and Y to center; then print "TimeCard" in the title bar.
    Printer.CurrentX = (PFrm.Width - Printer.TextWidth("TimeCard")) / 2
    Printer.CurrentY = (BarHgt - Printer.TextHeight("TimeCard")) / 2
    Printer.Print "TimeCard"
```

```
    ' Move the (0, 0) on the Printer object to the upper-left corner of
    ' the form's client area.
    SetClientPrintOrigin Card
    ' Redraw form lines on Printer.
    LinesOnPrinter
    ' Cycle through the controls collection of the form.
    For CtlCnt = 0 To PFrm.Controls.Count - 1
        If TypeOf PFrm.Controls(CtlCnt) Is CommandButton Then
            DrawCmd PFrm.Controls(CtlCnt)
        ElseIf TypeOf PFrm.Controls(CtlCnt) Is PictureBox Then
            DrawPic PFrm.Controls(CtlCnt)
        ElseIf TypeOf PFrm.Controls(CtlCnt) Is Label Then
            DrawLbl PFrm.Controls(CtlCnt)
        End If
    Next CtlCnt
    ' Print contents of Printer object.
    Printer.EndDoc
    ' Change cursor back to default.
    PFrm.MousePointer = 0
End Sub
```

This procedure uses the controls collection of a form to identify the controls that will appear on the Printer object. Only command buttons, picture boxes, and labels are redrawn for printing. This lets you ignore controls that you don't want to print. For example, the holiday, vacation, and sick time icons in the TimeCard application appear in image controls. The PrintFrm procedure ignores image controls, so they are not printed. For more information about the controls collection of a form, see Chapter 8, "Objects and Instances."

Recreating Text and Graphics on a Form

When creating text and graphics on a form using the **Print**, **Line**, **Circle**, or **PSet** methods, you may also want a copy of this output to appear on the Printer object. The easiest way to accomplish this is to use two statements whenever you print or draw on the form—one statement for the form and another for the Printer object. For example, these statements print the same text on the form and the Printer object:

```
SampleText = "Bold Modern type appears on the form and the printer."
MyForm.Print SampleText
Printer.Print SampleText
```

Applications with Multiple Forms

Duplicating statements for the form and Printer object can cause problems, however, if your application prints more than one form. While an application may have several forms, it can have only one Printer object. Duplicating your **Print** and graphics methods on the Printer object each time you use them on a form can cause information from different forms to overprint each other.

A better approach in this case is to isolate the **Print** and graphics methods in general procedures. Then you duplicate only these statements on the Printer object at print time. Since the **EndDoc** or **NewPage** methods clear the current page, the Printer object is left clean for the next form.

It isn't necessary to isolate **Print** and graphics methods for the form, but doing so makes it easier to track changes to your code so you can copy changes to the duplicate Printer object statements. For example, with the following statements, the TimeCard application draws two lines and a box on the form in the Form_Paint event procedure:

```
Sub Form_Paint ()
    DrawWidth = 2
    Line (1450, 360)-Step(0, 4040), QBColor(8)
    Line (4090, 360)-Step(0, 4040), QBColor(8)
    Line (2500, 4950)-Step(2532, 922), QBColor(8), B
End Sub
```

The LinesOnPrinter procedure draws the same lines and box on the Printer object with these statements:

```
Sub LinesOnPrinter ()
    DrawWidth = 2
    Printer.Line (1450, 360)-Step(0, 4040), QBColor(0)
    Printer.Line (4090, 360)-Step(0, 4040), QBColor(0)
    Printer.Line (2500, 4950)-Step(2532, 922), QBColor(0), B
End Sub
```

Printing Controls on a Form

The Printer object can receive the output of the **Print** method and the graphics methods (such as the **Line** or **PSet** methods). But you cannot place controls directly on the Printer object. If your application needs to print controls, you must write procedures that redraw each type of control you use on the Printer object.

Writing Generic Procedures

If possible, keep the procedures generic enough to reuse in other applications. This may mean writing procedures that account for variations in properties. For example, the DrawLbl procedure in the TimeCard application can position text with any of the three values of the Alignment property (left, center, and right alignment).

The DrawCmd Procedure

The DrawCmd procedure in the TimeCard application is called by the PrintFrm procedure. As the PrintFrm procedure cycles through all the controls on the time card, it calls the DrawCmd procedure to redraw command buttons on the Printer object. The following procedure takes a control as an argument; however, you should pass only command buttons to DrawCmd.

```
Sub DrawCmd (cmdCtrl As Control)
    ' Save the control width and height in local variables.
    dX = cmdCtrl.Width
    dY = cmdCtrl.Height
    ' Save the control x and y positions in local variables.
    cmdX = cmdCtrl.Left
    cmdY = cmdCtrl.Top
    ' Save the caption of the command button.
    capTxt = cmdCtrl.Caption
    ' Set the outline width of the redrawn button.
    DrawWidth = 2
    ' Move the x- and y-coordinates of the Printer object to
    ' the control's location on the form.
    Printer.CurrentX = cmdX
    Printer.CurrentY = cmdY
    ' Draw a box to represent the button outline.
    Printer.Line -Step(dX, dY), , B
    ' Move the x- and y-coordinates of the Printer object
    ' to center the caption of the command button horizontally
    ' and vertically in the box on the Printer object.
    Printer.CurrentX = cmdX + ((dX - Printer.TextWidth(capTxt)) / 2)
    Printer.CurrentY = cmdY + ((dY - Printer.TextHeight(capTxt)) / 2)
    ' Print the caption on the Printer object.
    Printer.Print cmdCtrl.Caption
End Sub
```

DrawCmd saves the position of the command button in variables to determine the position of the redrawn button on the Printer object. Using this approach assumes the application does not use command buttons inside a container other than a form.

If the command button is inside a frame, the position coordinates define the location of the command button within the frame, not the form. Using DrawCmd to print command buttons inside containers other than a form will cause those command buttons to print in the wrong locations. In this case, you need to modify DrawCmd (or create a different procedure) to draw the command button on the Printer object.

Printing Bitmaps on a Form

Printing a control that displays a bitmap presents a special problem. For picture box or image controls, the bitmap exists as the value of the control's Picture property. But the Printer object doesn't have a Picture property, so you cannot assign the value of the Picture property to a variable as you would with a Caption property.

You can recreate the bitmap on the Printer object by drawing the contents of the bitmap pixel by pixel. This works only with forms and picture boxes; image controls do not have the necessary properties. Because this process is time-consuming, printing the contents of a bitmap on the Printer object can reduce the performance of your application.

The DrawPic Procedure

The DrawPic event procedure in the TimeCard application is also called by the PrintFrm procedure. As the PrintFrm procedure cycles through all the controls on the time card, it calls the DrawPic procedure to redraw bitmaps in picture boxes on the Printer object. The following procedure takes a control as an argument; however, you should pass only picture boxes to DrawPic.

```
Sub DrawPic (picCtrl As Control)
    ' Save horizontal screen resolution.
    ScrX = Screen.TwipsPerPixelX
    ' Save vertical screen resolution.
    ScrY = Screen.TwipsPerPixelY
    ' Change ScaleMode to pixels.
    picCtrl.ScaleMode = 3
    ' For each row of pixels...
    For YRd = 0 To (picCtrl.ScaleHeight - 1)
        ' ...store pixel's vertical position in local variable.
        PelY = picCtrl.Top + (YRd * ScrY)
        ' For each pixel in the row...
        For XRd = 0 To (picCtrl.ScaleWidth - 1)
            ' ...store pixel's horizontal position in local variable...
            PelX = picCtrl.Left + (XRd * ScrX)
            ' ...and store pixel color.
            PelC = picCtrl.Point(XRd, YRd)
            ' If the current pixel isn't white...
            If PelC <> QBColor(7) Then
                ' ...draw and fill a box on the Printer object whose
                ' position/size/color match those of the current pixel.
                Printer.Line (PelX, PelY)-Step(ScrX, ScrY), PelC, BF
            End If
            ' Go to next pixel in the row.
        Next XRd
        ' Yield system control briefly.
        DoEvents
```

```
        ' Go to next row in the picture box.
    Next YRd
    ' Reset (0, 0) point to form's client area.
    SetClientPrintOrigin Card
End Sub
```

This procedure changes the ScaleMode of the picture box control to pixels so the color values of individual pixels in the control can be read more easily. The x- and y-coordinates of the picture box and the horizontal and vertical screen resolutions are saved in variables to increase speed and save space later in the procedure.

The procedure then scans each pixel in each row of the picture box. For each nonwhite pixel, the procedure draws a filled box on the Printer object using the **Line** method. White pixels are not redrawn to save time and thereby improve the performance of the application.

TwipsPerPixelX and TwipsPerPixelY

The keys to this procedure are the TwipsPerPixelX and TwipsPerPixelY properties of the Screen object. These properties allow conversion of pixels into the number of twips needed to size and place the filled boxes on the Printer object. For example, a pixel on a VGA display converts to 15 twips in the horizontal and vertical directions. By using the TwipsPerPixelX and TwipsPerPixelY properties, the DrawPic procedure accurately converts pixels into printed output for all screen resolutions.

Printing the Contents of the Printer Object

Once you have placed text and graphics on the Printer object, use the **EndDoc** method to print the contents. The **EndDoc** method advances the page and sends all pending output to the spooler. For example:

```
Printer.Print "This is the first line of text in a pair."
Printer.Print "This is the second line of text in a pair."
Printer.EndDoc
```

Note Visual Basic automatically issues an **EndDoc** call if your application ends or terminates without issuing an **EndDoc** call.

Creating Multiple-Page Documents

When printing longer documents, you can specify where you want a new page to begin by using the **NewPage** method. For example:

```
Printer.Print "This is page 1."
Printer.NewPage
Printer.Print "This is page 2."
Printer.EndDoc
```

After you use the **EndDoc** method to print one object, you can start printing another. The first page of the new document is assigned page number 1. This page number doesn't print automatically, but Visual Basic tracks the page number internally. You can locate this number with the Page property. For example, if you want to print a header at the top of every page of a budget report, use these statements:

```
Header$ = "Monthly Budget Summary "
Printer.Print Header$ + Printer.Page
Printer.NewPage
Printer.Print Header$ + Printer.Page
PageNo = Printer.Page
MsgBox "Your document contains " + PageNo + " pages."
Printer.EndDoc
```

Printing with the PrintForm Method

You can also send output to the printer by placing the output on a form and then printing the entire form using the **PrintForm** method. The syntax is:

[*form.*]**PrintForm**

If you omit the form name, Visual Basic prints the current form. **PrintForm** prints the entire form, even if part of the form is not visible on the screen. If a form contains graphics, however, the graphics print only if the form's AutoRedraw property is set to **True**.

The **PrintForm** method always sends pixels on the form directly to the printer. As a consequence, you may not get the highest resolution when you use **PrintForm**. For example, you could send text to a printer by printing it on a form and then calling **PrintForm** with the following statements:

```
Print "Here is some text."
PrintForm
```

Form1.PrintForm

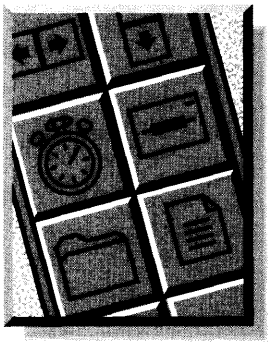
If the printer has higher resolution than your monitor, however, this statement prints the same text with better clarity:

```
Printer.Print "Here is some text."
```

For More Information For information on the **PrintForm** method, see the *Language Reference*, or search Help for *PrintForm*.

CHAPTER 17

Interacting with the Environment



Visual Basic allows you to take advantage of operating environment features such as the Clipboard and timers, and to process individual keystrokes in any control or for all controls on a form at once. Use these features to give your applications the extra polish that professional applications often require.

Contents

- Working with Selected Text
- Using the Clipboard Object
- Using Timer Controls
- Writing Idle Loops with **DoEvents**
- Writing Low-Level Keyboard Handlers

ALARM.MAK

Many of the code examples in the "Using Timer Controls" section of this chapter are taken from the ALARM.MAK sample application. If you installed the sample applications, you will find this application in the \ENVIR subdirectory of the main Visual Basic directory (\VB\SAMPLES\ENVIR).

Working with Selected Text

Text boxes and combo boxes have a series of properties for selected text that are especially handy when working with the Clipboard, as you'll see in the next section. These properties refer to the block of text selected (highlighted) inside the control. They allow you to create cut-and-paste functions for the user. The following properties can all be changed at run time.

Property	Description
SelStart	A **Long** integer that specifies the starting position of the selected block of text. If no text is selected, this property specifies the position of the insertion point. A setting of 0 indicates the position just before the first character in the text box or combo box. A setting equal to the length of the text in the text box or combo box indicates the position just after the last character in the control.
SelLength	A **Long** integer that specifies the number of characters selected.
SelText	The **String** containing the selected characters (or an empty string, if no characters are selected).

You can control what text is selected by setting the SelStart and SelLength properties. For example, these statements highlight all the text in a text box:

```
Text1.SelStart = 0          ' Start highlight before first character.
Text1.SelLength = Len(Text1.Text)           ' Highlight to end of text.
```

If you assign a new string to SelText, the new string replaces the selected text and the insertion point is placed just after the end of the newly inserted text. For example, the following statement replaces the selected text with the string "I've just been inserted!":

```
Text1.SelText = "I've just been inserted!"
```

If no text was selected, the string is simply pasted in at the insertion point.

For More Information For information on the SelStart, SelLength, and SelText properties, see the *Language Reference*, or search Help for *SelStart*, *SelLength*, and *SelText* properties.

Using the Clipboard Object

The *Clipboard object* is one of five special objects predefined in Visual Basic. (The other four are App, Screen, Printer, and Debug). The Clipboard object has no properties or events, but it has several methods that allow you to transfer data to and from the environment's Clipboard.

Two of the most useful Clipboard methods are **SetText** and **GetText**. These two methods transfer string data to and from the Clipboard, as shown in Figure 17.1.

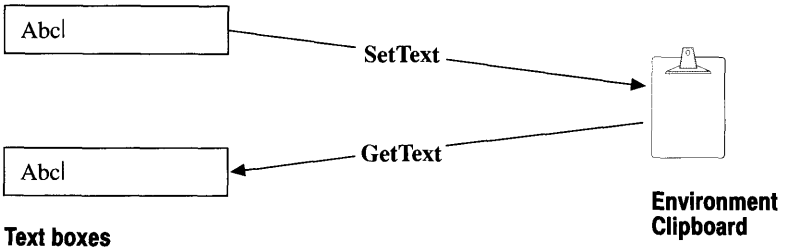


Figure 17.1 Moving data to and from the Clipboard with SetText and GetText

SetText copies text onto the Clipboard, replacing whatever text was stored there before. You use **SetText** like a statement, and its syntax is:

Clipboard.SetText *data*[, *format*]

GetText returns text stored on the Clipboard. You use it like a function:

destination = **Clipboard.GetText**()

By combining the **SetText** and **GetText** methods with the selection properties introduced in the previous section, you can easily write Copy, Cut, and Paste commands for a text box. The following event procedures implement these commands for controls named mnuCopy, mnuCut, and mnuPaste:

```
Sub mnuCopy_Click ()
    Clipboard.Clear
    Clipboard.SetText Text1.SelText
End Sub

Sub mnuCut_Click ()
    Clipboard.Clear
    Clipboard.SetText Text1.SelText
    Text1.SelText = ""
End Sub

Sub mnuPaste_Click ()
    Text1.SelText = Clipboard.GetText()
End Sub
```

Note The example works best if these are menu controls, because you can use menus while Text1 has the focus.

Notice that both the Copy and Cut procedures first empty the Clipboard with the **Clear** method (the Clipboard is not cleared automatically because you may want to place data on the Clipboard in several different formats, as described later in this section). Both the Copy and Cut procedures then copy the selected text in Text1 to the Clipboard with the statement:

```
Clipboard.SetText Text1.SelText
```

In the Paste command, the **GetText** method returns the string of text currently on the Clipboard. An assignment statement then copies this string into the selected portion of the text box (`Text1.SelText`). If no text is currently selected, Visual Basic places this text at the insertion point in the text box:

```
Text1.SelText = Clipboard.GetText()
```

This code assumes that all text is transferred to and from the text box Text1. This may sound limited at first, but the user actually can copy, cut, and paste between Text1 and controls on other forms.

Because the Clipboard is shared by the entire environment, the user can also transfer text between Text1 and any application using the Clipboard.

If you want the Copy, Cut, and Paste commands to work with any text box that has the focus, use the ActiveControl property of the Screen object. The following code provides a reference to whichever control has the focus:

```
Screen.ActiveControl
```

You can use this fragment just like any other reference to a control. If you know that the control is a text box, you can refer to any of the properties supported for text boxes, including Text, SelText, and SelLength. The following code assumes that the active control is a text box, and uses the SelText property:

```
Sub mnuCopy_Click ()
    Clipboard.Clear
    Clipboard.SetText Screen.ActiveControl.SelText
End Sub

Sub mnuCut_Click ()
    Clipboard.Clear
    Clipboard.SetText Screen.ActiveControl.SelText
    Screen.ActiveControl.SelText = ""
End Sub

Sub mnuPaste_Click ()
    Screen.ActiveControl.SelText = Clipboard.GetText()
End Sub
```

You can actually place several pieces of data on the Clipboard at the same time, as long as each piece is in a different format. This is useful because you don't know what application will be pasting the data, so supplying the data in several different formats enhances the chance that you will provide it in a format that the other application can use. The other Clipboard methods—**GetData**, **SetData**, and **GetFormat**—allow you to deal with data formats other than text by supplying a number that specifies the format. These formats are described in the following table, along with the corresponding number and symbolic name defined in CONSTANT.TXT.

Name in CONSTANT.TXT	Value	Description
CF_LINK	&HBF00	Dynamic-data-exchange link. Use this data format to give the user dynamic-link capability. For more information, see Chapter 21, "Communicating with Other Applications."
CF_TEXT	1	Text. Examples earlier in this section all use this format.
CF_BITMAP	2	Bitmap.
CF_METAFILE	3	Metafile.
CF_DIB	8	Device-independent bitmap.
CF_PALETTE	9	Color palette.

The last four formats (bitmap, metafile, device-independent bitmap, and palette) are all used for transferring graphic images. You can use these formats when cutting and pasting data from picture box controls. The following code provides generalized Cut, Copy, and Paste commands that work with any of the standard controls:

```
Sub mnuCopy_Click ()
    Clipboard.Clear
    If TypeOf Screen.ActiveControl Is TextBox Then
        Clipboard.SetText Screen.ActiveControl.SelText
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        Clipboard.SetText Screen.ActiveControl.Text
    ElseIf TypeOf Screen.ActiveControl Is PictureBox Then
        Clipboard.SetData Screen.ActiveControl.Picture
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Clipboard.SetText Screen.ActiveControl.Text
    Else
        ' No action makes sense for the other controls.
    End If
End Sub

Sub mnuCut_Click ()
    ' First do the same as a copy.
    mnuCopy_Click
    ' Now clear contents of active control.
    If TypeOf Screen.ActiveControl Is TextBox Then
        Screen.ActiveControl.SelText = ""
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        Screen.ActiveControl.Text = ""
    ElseIf TypeOf Screen.ActiveControl Is PictureBox Then
        Screen.ActiveControl.Picture = LoadPicture()
```

```
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Screen.ActiveControl.RemoveItem Screen.ActiveControl.ListIndex
    Else
        ' No action makes sense for the other controls.
    End If
End Sub

Sub mnuPaste_Click ()
    If TypeOf Screen.ActiveControl Is TextBox Then
        Screen.ActiveControl.SelText = Clipboard.GetText()
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        Screen.ActiveControl.Text = Clipboard.GetText()
    ElseIf TypeOf Screen.ActiveControl Is PictureBox Then
        Screen.ActiveControl.Picture = Clipboard.GetData()
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        Screen.ActiveControl.AddItem Clipboard.GetText()
    Else
        ' No action makes sense for the other controls.
    End If
End Sub
```

You can use the **GetFormat** method to determine whether the data on the Clipboard is in a particular format. For example, you can disable the Paste command depending on whether the data on the Clipboard is compatible with the currently active control:

```
Sub mnuEdit_Click ()
' Click event for the Edit menu.
    mnuCut.Enabled = True
    mnuCopy.Enabled = True
    mnuPaste.Enabled = False
    If TypeOf Screen.ActiveControl Is TextBox Then
        If Clipboard.GetFormat(CF_TEXT) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is ComboBox Then
        If Clipboard.GetFormat(CF_TEXT) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is ListBox Then
        If Clipboard.GetFormat(CF_TEXT) Then mnuPaste.Enabled = True
    ElseIf TypeOf Screen.ActiveControl Is PictureBox Then
        If Clipboard.GetFormat(CF_BITMAP) Then mnuPaste.Enabled = True
    Else
        ' Can't cut or copy from the other types of controls.
        mnuCut.Enabled = False
        mnuCopy.Enabled = False
    End If
End Sub
```

Note You might also want to check for other data formats with the constants CF_PALETTE, CF_DIB, and CF_METAFILE. If you want to replace a picture's palette using Clipboard operations, you should request CF_BITMAP rather than CF_DIB from the Clipboard. See the section "Using 256 Colors" in Chapter 15, "Creating Graphics for Applications," for more information on working with the color palette.

For More Information For more information on the Clipboard object, see the *Language Reference*, or search Help for *Clipboard*.

Using Timer Controls

Timer controls respond to the passage of time. They are independent of the user, and you can program them to take actions at regular intervals. A typical response is checking the system clock to see if it is time to do some task. Timers also are useful for other kinds of background processing.

Each timer has an Interval property, which specifies the number of milliseconds that pass between one timer event and the next. Unless disabled, a timer continues to receive an event (appropriately named the Timer event) at roughly equal intervals of time.

The Interval property has a few limitations to consider when you're programming a timer:

- The interval can be between 0 and 64,767, inclusive, which means that even the longest interval can't be much longer than one minute (about 64.8 seconds).
- The interval is not guaranteed to elapse exactly on time. To ensure accuracy, the timer should check the system clock when it needs to, rather than try to keep track of accumulated time internally.
- The system generates 18 clock ticks per second, so even though the Interval property is measured in milliseconds, the true precision of an interval is no more than one-eighteenth of a second.
- If your application or another application is making heavy demands on the system—such as long loops, intensive calculations, or disk, network, or port access—your application may not get timer events as often as the Interval property specifies.

Every timer control must be associated with a form. Therefore, to create a timer application, you must create at least one form (though of course you don't have to make the form visible if you don't need it for any other purpose).

Note The word "timer" is used in several ways in Visual Basic, each closely related to the workings of the timer control. In addition to the control name and control type, "timer" is used in the Timer event and the **Timer** function. (The latter returns the number of seconds elapsed since midnight.)

Placing a Timer Control on a Form

Placing a timer control on a form is like drawing any other control: Click the timer tool in the Toolbox and drag it onto a form. Or double-click the timer tool to create a default-sized timer on the current form, as shown in Figure 17.2. The latter technique often works best, because the size and position of the timer control don't matter at run time.

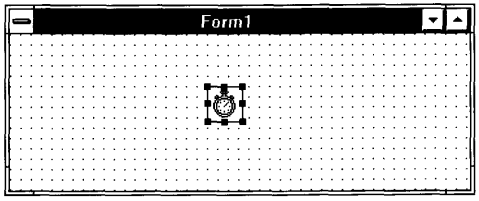


Figure 17.2 A timer control

The timer appears on the form at design time only so you can select it, view its properties, and write an event procedure for it. At run time, a timer is invisible and its position and size are irrelevant.

Initializing a Timer Control

A timer control has two key properties.

Property	Setting
Enabled	If you want the timer to start working as soon as the form loads, set to **True**. Otherwise, leave this property set to **False**. You may choose to have an outside event (such as a click of a command button) start operation of the timer.
Interval	Number of milliseconds between timer events.

Note that the Enabled property for the timer is different than for other objects. With most objects, the Enabled property determines whether the object can respond to an event caused by the user. With the timer control, setting Enabled to **False** suspends timer operation.

Remember that the Timer event is periodic. The Interval property doesn't determine "how long" as much as it determines "how often." The length of the interval should depend on how much precision you want. Because there is some built-in potential for error, make the interval one-half the desired amount of precision.

Note The more often a timer event is generated, the more processor time is eaten up in responding to the event. This can slow down overall performance. Don't set a particularly small interval unless you need it.

Responding to the Timer Event

When a timer control's interval elapses, Visual Basic generates the Timer event. Typically, you respond to this event by checking some general condition, such as the system clock.

A digital clock is a very simple but highly useful application involving a timer control. Once you understand how the application works, you can enhance it to work as an alarm clock, stopwatch, or other timing device.

The Digital Clock application includes a timer and a label with a border. At design time, the application looks like Figure 17.3.

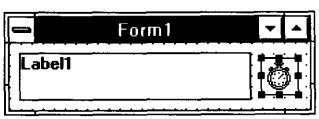


Figure 17.3 The Digital Clock Application

At run time, the timer is invisible.

Control	Property	Setting
Label1	BorderStyle	Fixed Single
Timer1	Interval	500 (half a second)
Timer1	Enabled	**True**

The sole procedure in the application is an event procedure for the timer:

```
Sub Timer1_Timer ()
    If lblTime.Caption <> CStr(Time) Then
        lblTime.Caption = Time
    End If
End Sub
```

The procedure displays the system time by calling the intrinsic **Time** function. This function returns a **Variant** containing the current time as a date/time value (**VarType** 7). When you assign it to a string variable or property, such as the Caption property in this case, Visual Basic converts it to a string using the time format specified in the Control Panel. If you want to display it using a different format, you can use the **Format** function.

For More Information For information on the **Format** function, see the *Language Reference,* or search Help for *Format.*

The Interval property for the timer is set to 500, following the rule of setting the Interval to half of the shortest period you want to distinguish (one second in this case). This may cause the timer code to update the label with the same time twice in one second. This is wasteful and can cause some visible flicker, so the code tests to see if the time is different from what is displayed in the label before it changes the caption.

You can customize the look of the Digital Clock without having to write any additional statements. For example, you might want to select a different font for the label or change the BorderStyle property of the form.

Building an Alarm Clock

Building an alarm clock involves only a little more programming than the Digital Clock. The Alarm Clock application is given an alarm time, and on each clock tick, the clock checks to see if the time has been reached. If so, it informs the user.

The alarm time is stored in a **Variant** declared in the Declarations section of the form:

```
Dim AlarmTime As Variant
```

To set the alarm time, the user clicks the blank portion of the form. (Alternatively, this can be implemented as a menu command or as a command button.) The procedure prompts the user for a time by using the **InputBox** statement. If the user presses the Cancel button on the input dialog box, the **InputBox** function returns an empty string. Therefore, the procedure assigns a new value to the alarm time only if the string returned is not empty. The procedure also determines if the user entered a valid time and converts it from a string representation (**VarType** 8) to a date (**VarType** 7).

Both the IsDate and the **CVDate** functions work correctly regardless of whether the user entered the time in a 24-hour format or the 12-hour A.M./P.M. format.

```
Sub Form_Click ()
    AlarmTime = InputBox("Enter alarm time", "VB Alarm", AlarmTime)
    If AlarmTime = "" Then Exit Sub
    If Not IsDate(AlarmTime) Then
        MsgBox "The time you entered was not valid."
    Else
        AlarmTime = CVDate(AlarmTime)
    End If
End Sub
```

The Timer event now compares the current time to the time stored in `AlarmTime`. After updating the time display, the Timer event checks to see if the alarm time has been reached yet. If so, it alerts the user:

```
Sub Timer1_Timer ()
    If lblTime.Caption <> CStr(Time) Then
        lblTime.Caption = Time
        If Time = AlarmTime Then
            Beep
            MsgBox "Alarm at " & Time
            AlarmTime = Null
        End If
    End If
End Sub
```

This code works fine as long as the clock is not interrupted at the wrong time. The application can run in the background and will beep and display the message box even though it doesn't have the focus. However, execution is suspended any time the user is moving the form.

If the application is not active on the same second that the alarm time is reached —for example, if some other application is doing disk access or tying up the CPU —then the alarm is never sounded. The following variation of the Timer event enables the Alarm Clock to report the alarm time as soon as it can, even if execution was suspended when the time was reached:

```
Sub Timer1_Timer ()
Static AlarmSounded As Integer
    If lblTime.Caption <> CStr(Time) Then
        lblTime.Caption = Time
        If Time >= AlarmTime And Not AlarmSounded Then
            Beep
            MsgBox "Alarm at " & Time
            AlarmSounded = True
        ElseIf Time < AlarmTime Then
            AlarmSounded = False
        End If
    End If
End Sub
```

In this version, the procedure tests to see if the current time is greater than or equal to alarm time. This way the application can detect whether the alarm should go off, even if it is returning from a suspended state and was not active at the alarm time. The static variable `AlarmSounded` ensures that the alarm goes off only once in each 24-hour cycle.

There is one more piece of code to add. The `AlarmTime` variable has the Empty value when the application starts, so the alarm goes off when the application starts (the current time is greater than the Empty value, which is always considered to be 0 when compared to numeric or date/time values). The solution is to make `AlarmTime` an empty string when the application starts:

```
Sub Form_Load ()
    AlarmTime = ""
End Sub
```

Writing Idle Loops with DoEvents

Normally, Visual Basic executes statements only in response to events. The time between events is referred to as "idle time."

You can write a loop that executes statements only when nothing else in the environment requires immediate attention. Such a loop is sometimes called an *idle loop*. In most cases, you should design your applications around events rather than write idle loops.

But if you do need certain statements or calculations to be performed continually, you definitely should use an idle loop. Unless restricted to idle time, an indefinite loop takes over the processor and doesn't permit other parts of the environment to respond to events. This makes the whole environment unresponsive.

Important Multitasking in Microsoft Windows is not preemptive. This means that Windows depends on each individual application to yield time voluntarily before it can service another application.

▶ **To write an idle loop**

1. From the Options menu, choose Project.
2. In the Project Options dialog, specify Sub Main as the Start Up Form. (As a result, no form is automatically loaded on startup, so you may need to use the **Show** or **Load** statement.)
3. Add a module to the project (or use an existing module if your project already contains one).

4. In the module, create a **Sub** procedure called Main.
5. Write a **Do While** loop that uses the **DoEvents** function:

```
Sub Main ()
    Form1.Show      ' Optionally, you can load and display a form.
    Do While DoEvents()
        ' Place idle-loop code here. These statements are processed
        ' whenever the system has free time.
    Loop
End Sub
```

An idle loop might be useful in a communications program (for example, a stock market ticker) that continually monitors data.

The **DoEvents** function switches control over to the operating environment kernel. It returns as soon as all other applications in the environment have had a chance to respond to pending events. This doesn't cause the current application to give up the focus, but it does enable background events to be processed. For example, if the Alarm Clock application from the previous section is running in the background, calling **DoEvents** gives the clock a chance to update its display.

The **DoEvents** function returns the number of forms in the application that are currently loaded. In the preceding code example, the **DoEvents** loop terminates when no forms are loaded. Also note that if any procedure anywhere in your application executes the **End** statement, all code in your application terminates unconditionally.

To call the **DoEvents** function, you must store the return value somewhere. If you don't want to use the value, use the **DoEvents** statement instead:

```
Do
    ' Perform processing here.
    DoEvents            ' Relinquish time to other applications.
Loop
```

DoEvents is useful if you have perform a time-intensive operation, such a long mathematical calculation, and don't want to slow down the responsiveness of the environment to the user. However, be careful when using **DoEvents** throughout your application. A procedure that has given up control with **DoEvents** must not be executed again before **DoEvents** returns. Otherwise, the procedure may end up being called endlessly, until system resources are exhausted.

For example, this procedure tests for prime numbers and uses **DoEvents** to periodically enable other applications to process events:

```
Function PrimeStatus (TestVal As Long) As Integer
    Dim Lim As Integer

    PrimeStatus = True
    Lim = Sqr(TestVal)
    For I = 2 To Lim
        If TestVal Mod I = 0 Then
            PrimeStatus = False
            Exit For
        End If
        If I Mod 200 = 0 Then DoEvents
    Next I
End Function
```

This code calls the **DoEvents** statement once every 200 iterations. This allows the PrimeStatus **Function** procedure to continue calculations as long as needed while the rest of the environment responds to events.

With this example, you must ensure that PrimeStatus is not called again before **DoEvents** returns. Consider what happens during a **DoEvents** call. Execution of application code is suspended while other forms and applications process events. One of these events might be a button click that launches the PrimeStatus procedure again. This would cause PrimeStatus to be re-entered before **DoEvents** could return. The results of PrimeStatus would then be unpredictable.

The solution is to prevent this from happening, either by disabling the control that calls the function or by setting a static "flag" variable while the function is running and not calling the function while the flag is set.

One of the effects of **DoEvents** is to give forms and windows a chance to respond to Paint events, so **DoEvents** can be used to update the display. However, because of the complications just described, **DoEvents** is not the safest way to cause screen updates. Use the **Refresh** method instead.

An alternative to using **DoEvents** in a calculation like this is to use a timer and process one (or more) iterations of the calculation on each timer event. Using a timer has several advantages over using **DoEvents**: You have finer control over the relative amount of processor time the calculation consumes (by adjusting the Interval property of the timer), and you can protect the calculation from re-entrancy simply by checking to see if the timer is currently enabled. The primary disadvantage to this approach is the necessity of making most of the variables in the timer event static to preserve their values from one timer event to the next.

For More Information For information on the **DoEvents** function and statement, see the *Language Reference*, or search Help for *DoEvents*. For information on the **Refresh** method, see the *Language Reference*, or search Help for *Refresh*.

Writing Low-Level Keyboard Handlers

Visual Basic provides three events that are recognized by forms and by any control that accepts keyboard input.

Table 17.1 Keyboard Events

Keyboard event	Occurs
KeyDown	As any key on the keyboard is pressed.
KeyPress	When a key corresponding to an ASCII character is pressed.
KeyUp	As any key on the keyboard is released.

Only the object that has the focus can receive a keyboard event. For the purpose of keyboard events, a form has the focus only if the form is active and no control on that form has the focus. This happens only on blank forms and forms on which all controls have been disabled. However, if you set the KeyPreview property on a form to **True**, the form receives all keyboard events for every control on the form before the control recognizes them. This is extremely useful when you want to do the same thing whenever a certain key is pressed, regardless of which control has the focus at the time.

The KeyDown and KeyUp events provide the lowest level of keyboard response. Use these events to detect a condition that the KeyPress event is unable to detect, including:

- Special combinations of SHIFT, CTRL, and ALT keys.
- Arrow keys. Note that some controls (command buttons, option buttons, check boxes) do not receive arrow key events. Instead, arrow keys cause movement to another control.
- PAGEUP and PAGEDOWN.
- Distinguishing the numeric keypad from numbers on the typewriter keys.
- Responding to a key being released as well as pressed (KeyPress responds only to a key being pressed).
- Function keys not attached to menu commands.

The keyboard events are not mutually exclusive. When the user presses a key, both the KeyDown and KeyPress events are generated, followed by a KeyUp event when the user releases the key. When the user presses one of the keys that KeyPress does not detect, only a KeyDown event occurs, followed by a KeyUp event.

Before using the KeyUp and KeyDown events, make sure that the KeyPress event isn't sufficient. This event detects keys that correspond to all the standard ASCII characters, including all alphanumeric keys and a number of others, including BACKSPACE, ENTER, and TAB. It's generally easier to write code for the KeyPress event.

You also should consider using shortcut and access keys, described in Chapter 4, "Menus and Dialogs." Shortcut keys must be attached to menu commands, but they can include function keys (including some function-key – shift-key combinations). These keys have the virtue of requiring no additional code.

The KeyPress Event

The KeyPress event occurs when any key that corresponds to an ASCII character is typed. This includes all the number and letter keys, punctuation keys, and a few special keys such as ESC (unless there is a command button with Cancel set to **True**), ENTER (unless there is a command button with Default set to **True**), and BACKSPACE. Use the KeyPress event whenever you want to process these keys. For example, if you want to force all the characters in a text box to be uppercase, you can use this event to change the case of the keys as they are typed:

```
Sub Text1_KeyPress (KeyAscii As Integer)
    KeyAscii = Asc(UCase(Chr(KeyAscii)))
End Sub
```

Notice that when a keypress causes this event to occur, the key is supplied to the event procedure as a numeric character code. The preceding code uses **Chr** to convert the code into the corresponding character, **UCase** to make the character uppercase, and **Asc** to turn the result back into a character code.

For More Information For a complete list of character codes, see the *Language Reference*, or search Help for *ASCII*.

You might notice that some keystrokes cause the system to beep. For example, pressing ENTER when there is no Default button on the form causes a beep. You can avoid this beep by intercepting the ENTER key (character code 13) in the KeyPress event:

```
Sub Text1_KeyPress (KeyAscii As Integer)
    If KeyAscii = 13 Then KeyAscii = 0
End Sub
```

The KeyDown and KeyUp Events

The KeyDown and KeyUp events provide information in a form different from the KeyPress event. This is because KeyDown and KeyUp report the exact physical state of the keyboard itself. These events do not interpret the keyboard state; you have to do that yourself.

The KeyPress event, in contrast, does not report the state of the keyboard directly. Instead, it supplies a generated character.

An example helps to illustrate the difference. When the user types uppercase "A," the KeyDown event gets the ANSI code for "A." The KeyDown event gets the same code when the user types lowercase "a." In contrast, the KeyPress event treats these two keys as two separate characters.

The KeyDown event returns information on the character typed by providing the following two arguments.

Argument	Description
KeyCode	Indicates the physical key pressed. In this case, "A" and "a" are returned as the same key. They have the identical *KeyCode* value. But note that "1" on the typewriter keys and "1" on the numeric keypad are returned as different keys, even though they generate the same character.
Shift	Indicates the shift-key state. Only by examining this argument can you determine whether an uppercase or lowercase letter was typed.

The KeyCode Argument

The argument *KeyCode* identifies a key by a numeric code. These codes are defined in the CONSTANT.TXT file. For example, some of the function-key codes are defined as:

```
Const KEY_F1 = &H70
Const KEY_F2 = &H71
Const KEY_F3 = &H72
Const KEY_F4 = &H73
Const KEY_F5 = &H74
```

Key codes for letter keys are the same as the ANSI codes of the uppercase character of the letter. So the *KeyCode* for both "A" and "a" is the value returned by **Asc**("A").

Key codes for the number and punctuation keys are the same as the ANSI code of the number on the key. So the *KeyCode* for both "1" and "!" is the value returned by **Asc**("1"). Again, numeric keypad keys are separate. They have their own special codes, as indicated in the CONSTANT.TXT file.

For example, this procedure quits the application when the user presses F2:

```
Sub Text1_KeyDown (KeyCode As Integer, Shift As Integer)
    Const KEY_F2 = &H71
    If KeyCode = KEY_F2 Then End
End Sub
```

For More Information For information on ANSI codes, see the *Language Reference*, or search Help for *ANSI*.

The Shift Argument

The argument *Shift* is nearly the same as the *Shift* introduced in Chapter 12, "Responding to Mouse Events." This argument returns the state of three keys—SHIFT, CTRL, and ALT—in the three least-significant bits. The three keys correspond to the bit patterns 001, 010, and 100, respectively. To test for a specific shift-key state, declare the following constants:

```
Const SHIFTKEY = 1
Const CTRL = 2
Const ALT = 4
```

You then can test for each shift-key state that you're interested in:

```
ShiftDown = (Shift And SHIFTKEY) > 0
CtrlDown = (Shift And CTRL) > 0
AltDown = (Shift And ALT) > 0
```

This procedure responds to the combination SHIFT+ALT+F2 by quitting the application:

```
Sub Text1_KeyDown (KeyCode As Integer, Shift As Integer)
Dim ShiftDown As Integer, AltDown As Integer, F2Down As Integer
    Const SHIFTKEY = 1
    Const CTRL = 2
    Const ALT = 4
    Const KEY_F2 = &H71

    ShiftDown = (Shift And SHIFTKEY) > 0
    AltDown = (Shift And ALT) > 0
    F2Down = (KeyCode = KEY_F2)

    If ShiftDown And AltDown And F2Down Then End
End Sub
```

Note To improve readability and maintenance of your code, make all the *Shift* and *KeyCode* global constants.

Writing a Form-Level Keyboard Handler

Each KeyDown and KeyUp event is attached to a specific object. To write a keyboard handler that applies to all objects on the form, set the KeyPreview property of the form to **True**. When the KeyPreview property is set to **True**, the form recognizes the KeyPress, KeyUp, and KeyDown events for all controls on the form before the controls themselves recognize the events. This makes it very easy to provide a common response to a particular keystroke. For example, if you want to make your application end when the user types SHIFT+ALT+F2 regardless of which control has the focus, you set KeyPreview to **True** and add this code:

```
Sub Form_KeyDown (KeyCode As Integer, Shift As Integer)
Dim ShiftDown As Integer, AltDown As Integer, F2Down As Integer
    Const SHIFTKEY = 1
    Const CTRL = 2
    Const ALT = 4
    Const KEY_F2 = &H71

    ShiftDown = (Shift And SHIFTKEY) > 0
    AltDown = (Shift And ALT) > 0
    F2Down = (KeyCode = KEY_F2)

    If ShiftDown And AltDown And F2Down Then End
End Sub
```

Notice that you have to write code like this only to handle keys that are not already defined as shortcut keys on menu items. If you have defined a shortcut key for a menu control, the Click event for that menu control occurs automatically when the user types that key, and no key event occurs. Similarly, if there is a command button on the form with the Default property set to **True**, the ENTER key causes the Click event for that command button to occur instead of a key event. If there is a command button with the Cancel property set to **True**, the ESC key causes the Click event for that command button to occur instead of a key event. Also notice that the TAB key moves the focus from control to control and does not cause a key event unless every control on the form is disabled or has TabStop set to **False**.

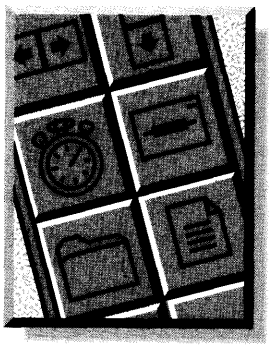
When the KeyPreview property of the form is set to **True**, the form recognizes the keyboard events before the controls, but the events still occur for the controls. However, the control key events do not occur if you set the *KeyAscii* or *KeyCode* arguments in the form key event procedures to 0. For example, if there is no default button on the form, you can use the ENTER key to move the focus from control to control:

```
Sub Form_KeyPress (KeyAscii As Integer)
Dim NextTabIndex, i
    If KeyAscii = 13 Then
        If Screen.ActiveControl.TabIndex = Count - 1 Then
            NextTabIndex = 0
        Else
            NextTabIndex = Screen.ActiveControl.TabIndex + 1
        End If
        For i = 0 To Count - 1
            If Me.Controls(i).TabIndex = NextTabIndex Then
                Me.Controls(i).SetFocus
                Exit For
            End If
        Next i
        KeyAscii = 0
    End If
End Sub
```

Because this code sets *KeyAscii* to 0 when it is 13, the controls never recognize the ENTER key being pressed, and their key event procedures are never called.

CHAPTER 18

Using the File-System Controls



Many applications must present information about disk drives, directories, and files. To allow users of your applications to explore the file system, Visual Basic includes three specialized controls: the drive list box, the directory list box, and the file list box.

You can write code to synchronize these controls so they display a particular drive with its directories and files. This chapter explains the file-system controls and shows you how to use them so users can investigate and choose among available disk files in your applications.

Contents

- Examining the File System
- A File Seeker Application

WINSEEK.MAK

Many of the code examples in this chapter are taken from the sample application WINSEEK.MAK. If you installed the sample applications, you will find this application in the \FILECTLS subdirectory of the main Visual Basic directory (\VB\SAMPLES\FILECTLS).

Examining the File System

The tools you use to draw the file-system controls appear in the middle of the Visual Basic Toolbox, as shown in Figure 18.1.

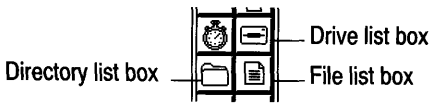


Figure 18.1 The tools for the file-system controls

Each of these tools has been carefully designed to combine flexible and sophisticated file-system inspection capabilities with easy programming. Each control performs its file-data retrieval tasks automatically, but you can write code both to customize their appearance and to specify which information they display.

File-system controls can be used singly or in combination. With combinations, you can write code in each control's event procedures to determine how they interact. Or you can let them act independently. Figure 18.2 shows the three controls used together.

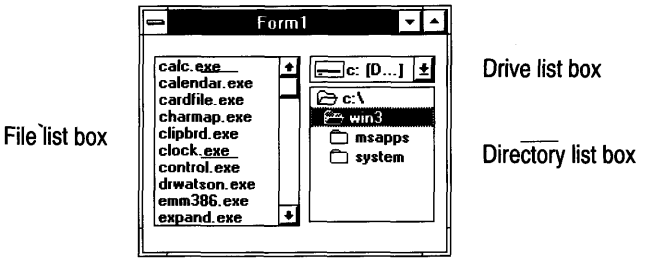


Figure 18.2 The file-system controls used together

File-system controls obtain all their information from the operating system automatically; you can access this information or determine what each control displays through its properties. For example, by default the file list box displays the contents of the current working directory (that is, the directory from which the application was launched, or what became the current directory as the result of a **ChDir** statement).

Your application can also display a list of the files with names matching a pattern, such as *.FRM. Simply draw a file list box on the form and set its Pattern property to *.FRM. You can specify the Pattern property at run time with the following code:

```
File1.Pattern = "*.FRM"
```

The Drive List Box

The drive list box is a drop-down list box. By default, it displays the current drive on the user's system. When this control has the focus, the user can type in any valid drive designation or click the arrow at the right of the drive list box. When the arrow is clicked, the list box drops down to list all valid drives. If the user selects a new drive from the list, that drive appears at the top of the list box.

You can use code to examine the Drive property of the drive list box to determine which drive is currently selected. Your application can also specify which drive appears at the top of the list box by this simple assignment:

```
Drive1.Drive = "c:\"
```

The drive list box displays valid available drives. Choosing a drive from the list box doesn't automatically change the current working drive; however, you can use the Drive property to change drives at the MS-DOS level by specifying it as an argument to the **ChDrive** statement:

```
ChDrive Drive1.Drive
```

The Directory List Box

The directory list box displays the directory structure of the current drive on the user's system, beginning with the top-level directory. Initially, the name of the current directory appears highlighted and indented from directories above it in the hierarchy, back to the root. Its subdirectories are indented beneath it. As a user moves up or down the list, each of the items is highlighted in turn.

Each directory in the box has an integer identifier associated with it. The directory specified by the Path property (Dir1.Path) always has the ListIndex value of −1. The directory immediately above it has the ListIndex value of −2, the one above that of −3, and so on up to the root. The first subdirectory of Dir1.Path has the ListIndex 0. If there are multiple directories at the first subdirectory level, the next has ListIndex 1, then 2, and so on, as shown in Figure 18.3.

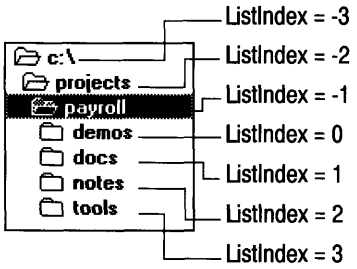


Figure 18.3 A directory structure displayed in the directory list box

Use the Path property of the directory list box to set or return the current directory in the box (ListIndex = −1). For example, if you assigned "c:\projects" to Drive1.Path in the preceding illustration, the \PROJECTS directory would be selected as the current working directory.

Similarly, you can assign the Drive property of the drive list box to the Path property of the directory list box:

```
Dir1.Path = Drive1.Drive
```

When this assignment is executed, the directory list box displays all the available directories and subdirectories on that drive. By default, the directory list box also displays all directories above, and any subdirectories immediately below, the current directory of a drive assigned to the Dir1.Path property. The directory list box doesn't *set* the current directory at the MS-DOS level; it merely highlights the directory and gives it the ListIndex value of –1.

To set the current working directory, use the **ChDir** statement. For example, the following statement changes the current directory to the one displayed by the directory list box:

```
ChDir Dir1.Path
```

In an application that uses file controls, you may want to set the current directory to the directory where the application's executable (.EXE) file is located. You do this using the Application object:

```
ChDir App.Path
```

For More Information For information on the Application object, see the *Language Reference*, or search Help for *applications*.

When a user clicks an item in a directory list box, that item is highlighted. When an item is double-clicked, it is assigned to the Path property, its ListIndex property gets the value –1, and the directory list box is redrawn to show its immediate subdirectories.

Note The Path property is available only at run time, not at design time.

The ListCount property returns the number of directories below the currently expanded directory, not the total number of items in the directory list box. Since the ListIndex value of the currently expanded directory is always –1, you can write code to determine how far down from the root the currently expanded directory is in the hierarchy. For example:

```
GoHigher = 0            ' Initialize for currently expanded directory.
Do Until Dir1.List(GoHigher) = ""   ' Dir1.List(x) returns empty string
    GoHigher = GoHigher - 1         ' if the directory doesn't exist.
Loop
LevelsAbove = Abs(GoHigher)         ' Convert to + number if desired.
```

The File List Box

The file list box displays files contained in the directory specified by the Path property at run time. You can display all the files in the current directory on the current drive using the following statement:

```
File1.Path = Dir1.Path
```

A subset of these files can be displayed by setting the Pattern property—for example, *.FRM displays only files with that extension. The Pattern property can also accept a list delimited by semicolons. For example, a line with the following code displays all files with the extensions .FRM and .BAS:

```
File1.Pattern = "*.FRM; *.BAS"
```

Visual Basic supports the ? wildcard character. For instance, ???.TXT displays files that have base names of only three characters, plus the extension .TXT. The attributes of the currently selected file (Archive, Normal, System, Hidden, and ReadOnly) are also available through file list box properties, so if you want to display only read-only files in the list box, simply set that property to **True** and the other Attribute properties to **False**:

```
File1.ReadOnly = True
File1.Archive = False
File1.Normal = False
File1.System = False
File1.Hidden = False
```

When Normal = **True**, those files without the System or Hidden attribute are displayed. When Normal = **False**, you can still display files with ReadOnly and/or Archive attributes by setting these attributes to **True**.

The default value for the System and Hidden attributes is **False**. The default value for the Normal, Archive, and ReadOnly attributes is **True**. If you set System to **True** and all others to **False**, files that have only the System attribute set will be displayed; those with both System and Hidden attributes set would not be displayed. If you want files with both System and Hidden attributes displayed, you must set both the System and Hidden properties to **True**. However, if both System and Hidden properties are **True**, files that have only one of those attributes will also be displayed. To show all files, Normal, System, and Hidden must all be **True** (the Archive and ReadOnly attributes are ignored).

Note You cannot use the Attribute properties to set a file's attributes. To do this, use the **SetAttr** statement.

By default, the user can highlight only a single selection in a file list box. Use the MultiSelect property to let the user select multiple files.

For More Information For information on multiple selection in list boxes, see the *Language Reference*, or search Help for *MultiSelect*.

Using File-System Controls Together

If you use a combination of file-system controls, you can synchronize the information they display. For example, if you have a drive list box, directory list box, and file list box with the default names Drive1, Dir1, and File1, the sequence of events might work like this:

1. The user selects a drive in the Drive1 list box.
2. A Drive1_Change event is generated, and the display in Drive1 is updated to reflect the new drive.
3. Code in the Drive1_Change event procedure assigns the new selection (the Drive1.Drive property) to the Path property of the Dir1 list box with the following statements:

```
Sub Drive1_Change ()
    Dir1.Path = Drive1.Drive
End Sub
```

4. The assignment to the Path property generates a Dir1_Change event and updates the display in Dir1 to reflect the current directory of the new drive.
5. Code in the Dir1_Change event procedure assigns the new path (the Dir1.Path property) to the File1.Path property of the File1 list box:

```
Sub Dir1_Change ()
    File1.Path = Dir1.Path
End Sub
```

6. The assignment to the File1.Path property causes the display in the File1 list box to reflect the Dir1 path specification.

The event procedures you use and the properties you change depend on the way your application uses the combination of file-system controls. The code in the next section of this chapter illustrates the synchronization of controls described in this section.

For More Information For information on file-system controls, see the *Language Reference*, or search Help for *file-system controls*.

A File Seeker Application

Many applications provide capabilities for investigating the file system, since users often want to find a file or group of files available to an application quickly. The WINSEEK.MAK sample application (shown in Figure 18.4) helps the user browse drives and directories and displays any category of files.

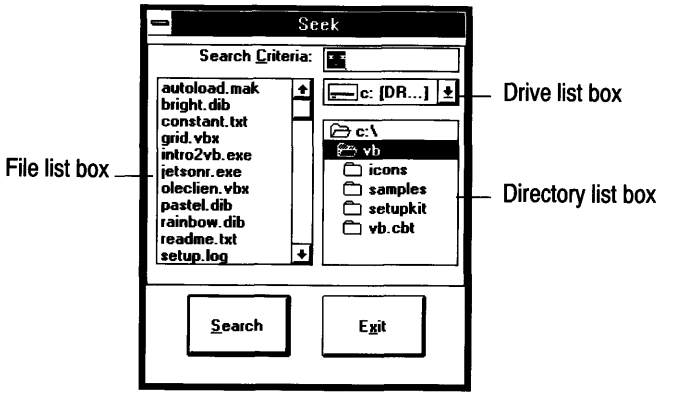


Figure 18.4 File controls in the WinSeek application

The following table summarizes the controls in SEEK.FRM from the WinSeek application.

Control	Property	Setting
Drive list box	Name	drvList
Directory list box	Name	dirList
File list box	Name	filList
	Pattern	*.*
Command button	Name	cmdSearch
	Caption	&Start Search
	Default	**True**
Command button	Name	cmdExit
	Caption	E&xit
List box	Name	lstFoundFiles

Note The file-system controls do not have caption properties, although you can label them and give them access keys. For more information on using labels this way, see Chapter 3, "Creating and Using Controls."

Writing Code for the WinSeek Application

It is important to understand that in the drive list box, a Change event is triggered by a single mouse click on an item; however, in the directory list box, a double-click event is necessary to generate a Change event. (In the drive list box, a Change event also occurs when the user selects an item and then changes the focus on the form.)

When users want to change directories without using a mouse, they typically use the arrow keys to select the desired directory and then press the ENTER key. Since ENTER is commonly associated with the default command button control, WinSeek needs a way to recognize when the user simply wants to change directories rather than conduct a search for files.

The WinSeek application resolves this ambiguity by determining if the path of the dirList box is different from the currently highlighted directory. This situation can occur when the user single-clicks an item in the directory list box or navigates the directory list box using the arrow keys. The following code determines whether the dirList.Path is different from the path of the highlighted directory. If the paths are different, the dirList.Path is updated. If the paths are the same, the search is performed.

```
' If the dirList.Path is different from the currently selected
' directory, update it; otherwise perform the search.
If dirList.Path <> dirList.List(dirList.ListIndex) Then
    dirList.Path = dirList.List(dirList.ListIndex)
    Exit Sub
End If
' Continue with search.
    .
    .
    .
```

The WinSeek application uses the following procedures to handle significant events:

- The drvList_Change procedure
- The dirList_Change procedure
- The cmdSearch_Click procedure

The drvList_Change Procedure

When the user clicks an item in the drive list box, its Change event is generated. The drvList_Change event procedure is invoked, and the following code is run:

```
Sub drvList_Change ()
    On Error GoTo DriveHandler
    dirList.Path = drvList.Drive ' If new drive was selected, the
    Exit Sub                     ' Dirl box updates its display.
DriveHandler:                       ' If there is an error,
    drvList.Drive = dirList.Path ' reset drvList.Drive with
    Exit Sub                     ' the drive from dirList.Path.
End Sub
```

Notice that the Change event in a drive list box occurs when a new drive is selected, either with a single mouse click or when the user moves the selection (for example, with an arrow key). The error handler is triggered by actions such as attempting to access a floppy disk drive while the drive door is open or selecting a network drive that has been inadvertently disconnected. Since the error prevents the original assignment, dirList.Path still contains the previous valid drive. Reassigning dirList Path to drvList.Drive corrects this error.

For more information on handling errors, see Chapter 10, "Handling Run-Time Errors."

The dirList_Change Procedure

If the user double-clicks an item in the directory list box, or if the Path property of dirList is changed in code (as in the drvList_Change procedure), the dirList_Change event is initiated. The following code responds to that event:

```
Sub dirList_Change ()
    ' Update file list box to synchronize with the directory list box.
    filList.Path = dirList.Path
End Sub
```

This procedure assigns the Path property of the dirList box to the Path property of the filList box. This causes a PathChange event in the filList control, which is redrawn; however, you don't need to add code to the filList_PathChange procedure, because in this application the event chain ends in the filList list box.

The cmdSearch_Click Procedure

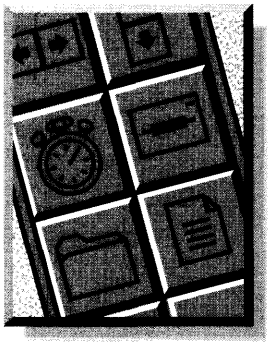
This procedure determines whether the highlighted item in the dirList list box is the same as the dirList.Path. If they are different, then dirList.Path is updated. If they are the same, then the search is performed.

```
Sub cmdSearch_Click ()
    .
    .
    .
    ' If the dirList.Path is different from the currently
    ' selected directory, update it; otherwise perform the search.
    If dirList.Path <> dirList.List(dirList.ListIndex) Then
        dirList.Path = dirList.List(dirList.ListIndex)
        Exit Sub
    End If
    ' Continue with search.
    .
    .
    .
End Sub
```

Note You can enhance the WinSeek application with additional features. For example, you might want to use a file control's Attribute properties. You could use check boxes to allow the user to set different combinations of file attributes so that the file list box displays files that are Hidden, System, and so on. This would in turn restrict a search to conforming files.

CHAPTER 19

Processing Files



This chapter describes how to create and use permanent (disk) files with Visual Basic. By processing files, your applications can create, manipulate, and store large amounts of data, access several sets of data at once, and share data with other applications.

Contents

- Overview of File Access Types
- Random Access
- Sequential Access
- Binary Access

RECEDIT.MAK and TEXTEDIT.MAK

The code examples in this chapter are taken from two sample applications, Record Editor and Text Editor. To view the files of the Record Editor, open RECEDIT.MAK, which appears in the \FILEIO subdirectory of the main Visual Basic directory (\VB\SAMPLES\FILEIO). To view the files of the Text Editor, open TEXTEDIT.MAK, which appears in the \MENUS subdirectory of the main Visual Basic directory.

Overview of File Access Types

In Visual Basic, there are three types of file access:

- Random
- Sequential (**Input**, **Output**, and **Append**)
- Binary

By itself, a *file* consists of nothing more than a series of related bytes located on disks. When your application accesses a file, it must make assumptions about what the bytes are supposed to represent (integers, strings, or other data types). Visual Basic provides functions and statements that allow you to process the file based on these assumptions.

The sections that follow provide a brief overview of the access types, their advantages and disadvantages, and detailed discussions of each access type.

Random Access

A file opened for *random access* is assumed to be composed of a series of *records* of identical length. Although a record must correspond to a single data type, user-defined types can be used to create records made up of numerous fields, each of which can have different data types.

The length of each record (as defined by its type declaration) must be supplied as a parameter in the **Open** statement used to open a file for random access. Otherwise, Visual Basic assumes that the record length is 128 bytes. This length is used to calculate the position of each record in the file. This position is called the *record number*. All you need to do to access a particular record is to specify its record number.

Random access provides significant advantages in accessing and dealing with records. On the other hand, data within a record is typically shorter than the fields allow, which can lead to wasted disk space.

Sequential Access

Sequential access is designed for use with plain text files. Each character in the file is assumed to represent either a text character or a text formatting sequence, such as a newline character (NL).

Sequential access lets you use procedures especially designed for writing and reading lines or strings of text. It makes it easy to deal with files produced by a text editor—that is, files in which data is *not* divided into a series of records.

However, sequential access may not be well suited for storing long series of numbers, because each number is stored as a character string. A four-digit number would require 4 bytes of storage instead of the 2 bytes it requires to store the same number as an integer. Sequential access may also be poorly suited for use with data that requires access to the file at any point other than the beginning.

Binary Access

Binary access allows you to use files to store data however you want; there are no assumptions made about data type or requirements for standard record length. But you must know precisely how the data was written to the file in order to retrieve it correctly.

For example, if you retrieve the data at file location 112 as an integer, bytes 112 and 113 are retrieved to make up an integer value. It doesn't matter that these two bytes may be part of 4 bytes previously stored as a **Long** value. It is up to your application to keep track of the contents of the file and make sure that such actions are correct.

While binary access provides the fewest functions and procedures to help manipulate data, it also provides the greatest flexibility. For example, you can conserve disk space by building variable-length records.

Random Access

As described earlier, the bytes in random-access files form identical records, each containing one or more fields. A record with one field corresponds to any standard type, such as an integer or fixed-length string. A record with more than one field corresponds to any user-defined type. For example, Figure 19.1 shows how the bytes of one file can form 19-byte records that consist of 10-byte, 7-byte, and 2-byte fields.

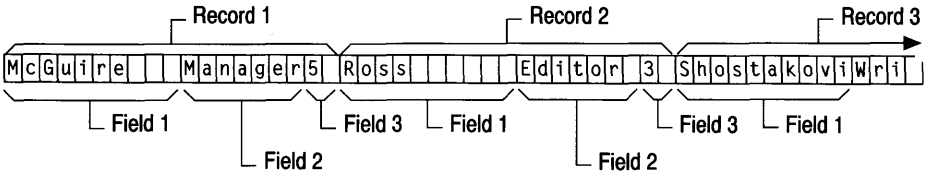


Figure 19.1 Records in a file

In a file opened for random access, all records must correspond to one type. For example, all records in Figure 19.1 correspond to the following type:

```
Type Worker
    LastName    String * 10
    Title       String * 7
    Rank        String * 2
End Type
```

With random access, you can:

- Define record types.
- Open files for random access.
- Edit files opened for random access.

The following functions and statements apply to files opened for random access.

Random-access functions	Random-access statements
Dir, Dir$	**Open**
EOF	**Close**
FileCopy	**Get**
FileDateTime	**Put**
FileLen	**Type...End Type**
FreeFile	
GetAttr	
Loc	
LOF	
Seek	
SetAttr	

Defining Record Types

Before opening a file for random access, define a type that corresponds to the records the file does or will contain. The Record Editor sample application declares a user-defined data type called `Person` with the following code:

```
Type Person
    ID                  As Integer
    MonthlySalary       As Double
    LastReviewDate      As Long
    FirstName           As String * 15
    LastName            As String * 15
    Title               As String * 15
    ReviewComments      As String * 150
End Type
```

Declaring Field Variables in a Type Definition

Because records in a file opened for random access must all have the same length, user-defined type elements declared as strings must have a fixed length. For example, the `Person` type declares the `FirstName` and `LastName` elements as strings with a fixed length of 15 characters.

If the actual string copied into a file's record contains fewer characters than the fixed length of a string, Visual Basic fills the trailing spaces in the record with blanks (character code 32)—often causing wasted storage space. Also, if the string is longer than the field size, it is truncated.

For more information about user-defined types, see Chapter 7, "Variables, Constants, and Data Types."

Declaring Variables

After defining a type that corresponds to a typical record, declare any other variables that your application needs to process a file opened for random access. For example, the following variables are declared in Record Editor:

```
Global Employee As Person          ' A record variable.
Global Position As Long            ' Tracks the current record.
Global LastRecord As Long          ' The number of the last record
                                   ' in the file.
```

Opening Files for Random Access

To open a file for random access, you use the following syntax for the **Open** statement:

Open *file* **For Random As** *filenumber* **Len** = *recordLen*

Len = *recordLen* is used to specify the size of each record. If this value is incorrect, access to the file will be incorrect and data will be corrupted.

You could use the following code to open a file:

```
Dim FileNum As Integer
Dim RecordLen As Long
Dim Employee As Person

' Calculate the length of each record.
RecordLen = Len(Employee)

' Get the next available file number.
FileNum = FreeFile

' Open the new file with the Open statement.
Open "MYFILE.FIL" For Random As FileNum Len = RecordLen
```

Note that random-access file mode is the default. If you omit **For Random** in the **Open** statement, Visual Basic assumes you want to open a file for random access:

```
Open NewFileName As FileNum Len = RecordLen
```

Editing Files Opened for Random Access

To edit a file opened for random access, copy records from the file into variables, change the values in the variables, and copy the program variables back into the file.

Reading Records into Variables

Use the **Get** statement to copy records into program variables. For example, the Record Editor sample application uses the **Get** statement in the ShowRecord procedure to copy a record from a file into the `Employee` variable:

```
Get FileNum, Position, Employee
```

In this line of code:

- The `FileNum` variable contains the number used to open the file with the **Open** statement.
- The `Position` variable contains the record number of the record to copy from the specified file.
- The `Employee` variable, declared as `Person` type, receives the contents of the record.

For More Information For information about the **Get** statement, see the *Language Reference*, or search Help for *Get*.

Editing Records

Once you've read a record into a variable, you can display the contents of the variable on a form. For example, the form used by the Record Editor (Figure 19.2) contains a control array of text boxes that corresponds to each field of an employee record. The form provides an ideal vehicle for viewing or editing record contents.

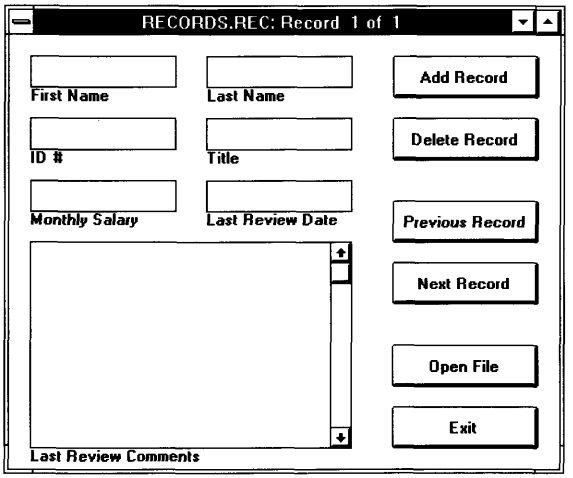


Figure 19.2 The form for the Record Editor sample application

Writing Variables to Records

Use the **Put** statement to copy records into files opened for random access. For example, the Record Editor uses **Put** in three instances: when replacing records, when adding new records, and when deleting records.

Replacing Records

The Record Editor replaces records with this line of code in the GetFields procedure:

```
Put #FileNum, Position, Employee
```

`Position` in this instance refers to the number of the record you want to replace (overwrite) with the data in the `Employee` variable.

Adding Records

To add new records to the end of a file opened for random access, use the **Put** statement shown in the preceding code fragment. Set the value of the `Position` variable equal to one more than the number of records in the file. For example, to add a record to a file that contains five records, set `Position` equal to 6.

In the Record Editor, the AddRecord_Click procedure uses the **Put** statement to add the record to the file:

```
Sub AddRecord_Click ()
    .
    .
    .
    LastRecord = LastRecord + 1
    Put #FileNum, LastRecord, Employee
    .
    .
    .
End Sub
```

For More Information For information about the **Put** statement, see the *Language Reference*, or search Help for *Put*.

Deleting Records

You could delete a record by clearing its fields, but the record would still exist in the file. Usually you don't want such gaps in your file, because they waste space and interfere with sequential operations.

A better way to delete a record is to copy all the records that follow it "down" one record position, which overwrites the unwanted record.

For example, you can use the following code to delete records:

```
Dim TempVar As Person
Dim Ind As Integer
    .
    .
    .
For Ind = Position To LastRecord - 1
    Get FileNum, Ind + 1, TempVar
    Put FileNum, Ind, TempVar
Next
```

This example uses a **For...Next** loop to overwrite the contents of the record number contained in `Ind` with the contents of the record immediately following it.

Notice that this procedure has one problem: The last iteration of the loop copies the last record of the file into the record immediately preceding it, leaving a duplicate of that record at the end of the file.

▶ **To remove a duplicate record in a random-access file**

1. Create a new file.
2. Copy all the valid records from the original file to the new file.
3. Close the original file and use the **Kill** statement to delete it.
4. Use the **Name** statement to rename the new file with the name of the original file.

For More Information For information about the **For...Next** statement, the **Name** statement, or the **Kill** statement, see the *Language Reference*, or search Help for the appropriate statement name.

Sequential Access

Sequential access works best when you want to process files consisting only of text, such as the files created with a typical text editor.

With sequential access, you can:

- Open files for sequential access.
- Edit files opened for sequential access.

The following functions and statements apply to files opened for sequential access.

Sequential-access functions	Sequential-access statements
Dir, Dir$	**Open**
EOF	**Close**
FileCopy	**Input$**
FileDateTime	**Input #**
FileLen	**Line Input #**
GetAttr	**Write #**
Loc	
LOF	
Seek	
SetAttr	

Opening Files for Sequential Access

When you open a file for sequential access, you open it to perform one of the following operations:

- Input characters to a file (**Input**)
- Output characters to a file (**Output**)
- Append characters to a file (**Append**)

After opening a file for one of these operations, you must close the file (using the **Close** statement) before reopening it for another type of operation.

To open a file for sequential access, you use the following syntax for the **Open** statement:

Open *file* **For [Input | Output | Append] As** *filenumber* **Len** = *buffersize*

Note When you use sequential access to open a file for **Input**, the file must already exist; otherwise, Visual Basic generates an error. When you try to open a nonexistent file for **Output** or **Append**, however, the **Open** statement creates the file first and then opens it.

Editing Files Opened for Sequential Access

To edit a file opened for sequential access, copy the contents of the file into variables, change the variables, and then copy the variables back into the file.

Copying Strings from Files

To retrieve the contents of a text file, open the file for sequential **Input**. Then use the **Line Input #, Input$,** or **Input #** statement to copy the file into program variables.

Normally, the entire contents of a text file are retrieved in one procedure. However, Visual Basic has statements that let you read and write sequential files one line at a time. For example, the following code fragment reads a file line by line:

```
Dim LinesFromFile, NextLine As String

Do Until EOF(FileNum)
    Line Input #FileNum, NextLine
    LinesFromFile = LinesFromFile + NextLine + Chr(13) + Chr(10)
Loop
```

The loop reads the contents of the file until the **EOF** function returns **True**, indicating that the end of file has been reached. With each iteration of the loop, the **Line Input #** statement places a single line from the file in the variable `NextLine`.

Line Input # recognizes the end of a line when it finds the carriage return-linefeed sequence in the file. However, it does not include the carriage return-linefeed when it reads the line into the variable. Therefore, if you want to retain the carriage return-linefeed in each line, your code must add them (as shown in the example).

You can also use the **Input** function to copy any number of characters from a file to a program variable. For example, the following code uses **Input$** to read an entire file at once into a variable:

```
LinesFromFile = Input$(LOF(FileNum), FileNum)
```

Visual Basic also supports the **Input #** statement. Use **Input #** to read a list of numeric and/or string expressions written to the file with the **Write #** statement.

For More Information For information about the **Input #** statement, see the *Language Reference*, or search Help for *Input #*.

After you have copied the file's contents into a variable, close the file with the **Close** statement.

Using the Text Box Control

Text box controls are often used in Visual Basic to display text files. This limits the maximum length of a file that can be edited to just under 32K, which is the maximum number of characters you can assign to a Visual Basic multiline text box control.

Before reading a file into a text box, you must confirm that the file will fit. This example uses 30000:

```
If LOF(FileNum) > 30000 Then
    Msg = "Sorry, your file is too large to edit."
    MsgBox Msg, 16, "File Too Big"
    Exit Sub
End If
```

The **LOF** (length of file) function requires that the file be open before its length can be determined. If you want to determine a file's length without opening it, use the **FileLen** function.

Writing Strings to Files

To store the contents of program variables in a sequential file, open it for sequential **Output** or **Append**, and then use the **Print #** statement. For example, a text editor might use the following line of code to copy the entire contents of a text box into a file:

```
Print #FileNum, TheBox.Text
```

Visual Basic also supports the **Write #** statement. The **Write #** statement allows you to write a list of string and/or numeric expressions to a file. It automatically separates each expression with a comma and puts quotation marks around string expressions. For example, consider the following lines of code:

```
Dim AnyString As String
Dim AnyNumber As Integer

AnyString = "AnyCharacters"
AnyNumber = 23445
Write #FileNum AnyString, AnyNumber
```

This code writes two expressions to the file specified by `FileNum`. The first expression contains a string, and the second expression contains the number 23445. Therefore, Visual Basic writes the following characters to the file:

```
"AnyCharacters",23445
```

Notice that the file actually contains quotation marks around the characters in `AnyString`.

Note If you are using **Write #** and **Input #** with sequential access, you may want to consider using random or Binary access instead. They are better suited to dealing with record-oriented data.

For More Information For information about **Write #**, see the *Language Reference*, or search Help for *Write #*.

Binary Access

Binary access allows you the greatest flexibility in dealing with a file, because the bytes in the file can represent anything. Use binary access when it is important to keep file size small.

The following functions and statements apply to files opened for binary access.

Binary-access functions	Binary-access statements
Dir, Dir$	**Open**
EOF	**Close**
FileCopy	**Get**
FileDateTime	**Put**
FileLen	**Input**
FreeFile	**Input$**
GetAttr	
Loc	
LOF	
Seek	
SetAttr	

Opening a File for Binary Access

To open a file for binary access, you use the following syntax for the **Open** statement:

Open *file* **For Binary As** *filenumber*

As you can see, **Open** for binary access differs from **Open** for random access in that **Len** = *recordLen* is not specified. If you include a record length in a binary-access **Open** statement, it is ignored.

You must specify **For Binary** in your **Open** statement. If you omit it, Visual Basic assumes you are opening the file for random access.

Storing Information in Variable-Length Fields

In order to understand how and when to use binary access, consider the Employee Records file from the discussion on random access earlier in this chapter. This example used records and fields of fixed length (as required by random access) to store information about employees. The typical record is described in a **Type...End Type** statement:

```
Type Person
    ID                  As Integer
    MonthlySalary       As Double
    LastReviewDate      As Long
    FirstName           As String * 15
    LastName            As String * 15
    Title               As String * 15
    ReviewComments      As String * 150
End Type
```

Every record in this file takes 209 bytes, regardless of the actual contents of the fields. In the `ReviewComments` field, the string "Good Job!" requires the same amount of storage space as "The employee has shown dramatic improvement in the past quarter and has a good attitude for the future."

Even 150 characters might be too small for `ReviewComments`. Perhaps you need to allow for as many as 500 characters, but the average entry takes only 20 characters. Using random access, you'd have to set the field to its maximum desired width, which could waste 480 bytes per record in this field alone.

This waste can be avoided using binary access. Since you no longer need to have fixed-length fields in all cases, the **Type** definition statement can be changed to:

```
Type Person
    ID                  As Integer
    MonthlySalary       As Double
    LastReviewDate      As Long
    FirstName           As String
    LastName            As String
    Title               As String
    ReviewComments      As String
End Type

Dim Empl As Person          ' Defines a record.
```

File Output Using Binary Access

Each employee record in the preceding example now stores only the exact number of bytes required. However, this poses a problem when you need to know how long the variable-length fields are. This information is required later to correctly retrieve the stored record.

A good way to solve this problem is to store an integer, which describes the length of the string, in the file before each variable-length string. This costs a little storage space (2 bytes per string), but still makes the average employee record considerably shorter.

The following Write_A_Record procedure writes a record to the Employee Records file.

```
Sub Write_A_Record (Start_Pos As Long)
    Dim StrSize As Integer

    ' Assumes the file is opened as #1.
    ' Start_Pos is the byte position in the file where you want
    ' the beginning of the record to be placed. You have to indicate
    ' it only on the first Put; the rest follow in order.

    Put #1, Start_Pos, Empl.ID                  ' Size is fixed.
    Put #1, , Empl.MonthlySalary                ' Size is fixed.
    Put #1, , Empl.LastReviewDate               ' Size is fixed.

    StrSize = Len(Empl.FirstName)               ' Size is variable, so get
    Put #1, , StrSize                           ' the size and output it
    Put #1, , Empl.FirstName                    ' before the string.

    StrSize = Len(Empl.LastName)                ' Same.
    Put #1, , StrSize
    Put #1, , Empl.LastName

    StrSize = Len(Empl.Title)                   ' Same.
    Put #1, , StrSize
    Put #1, , Empl.Title

    StrSize = Len(Empl.ReviewComments)          ' Same.
    Put #1, , StrSize
    Put #1, , Empl.ReviewComments

End Sub
```

One trade-off in using variable-length fields and binary access instead of fixed-length fields and random access is that the entire record could be written with a single function call using random access. While binary access provides greater flexibility, it also requires more code to handle I/O operations.

Reading Files with Binary Access

The **Get** statement reads the correct number of bytes into any variable of known length, such as a fixed-length string or an integer. When you use **Get** with a variable-length string, the number of bytes read from the file equals the *current length* of the string.

Consider the following example:

```
Comments = "How are you?"          ' String contains 12 characters.
Get #1, , Comments                 ' Retrieves 12 bytes from file #1
                                   ' into the Comments$ string.
```

To get the number of bytes you want, you must first make sure that the string field is the correct length. The following Get_A_Record procedure reads an employee record as saved earlier in this section. Just before reading each variable field, it reads the length of that field from the file. It then uses the **String$** function to set the receiving string to a string of blank characters of the correct size.

```
Sub Get_A_Record (Start_Pos As Long)
    Dim StrSize As Integer

    ' This Sub assumes the file is open as #1 and that you are
    ' passing the starting file position of the record as
    ' Start_Pos.

    Get #1, Start_Pos, Empl.ID          ' Fixed -- no need to adjust.
    Get #1, , Empl.MonthlySalary        ' Fixed -- no need to adjust.
    Get #1, , Empl.LastReviewDate       ' Also fixed-length.

    Get #1, , StrSize                       ' Gets the field size.
    Empl.FirstName = String$(StrSize, " ")  ' Sizes the field.
    Get #1, , Empl.FirstName                ' Reads the field.

    Get #1, , StrSize
    Empl.LastName = String$(StrSize, " ")
    Get #1, , Empl.LastName
```

```
    Get #1, , StrSize
    Empl.Title = String$(StrSize, " ")
    Get #1, , Empl.Title

    Get #1, , StrSize
    Empl.ReviewComments = String$(StrSize, " ")
    Get #1, , Empl.ReviewComments

    Position = Seek (#1)            ' Gets and saves the current file
                                    ' position to a global variable.
End Sub
```

Another way to retrieve the variable strings is with the **Input$** statement. The following example performs the same task as the Get_A_Record procedure—it retrieves a record containing variable-length records.

```
Sub Input_A_Record (Start_Pos As Long)
    Dim StrSize As Integer

    ' This Sub assumes the file is open as #1 and that you are
    ' passing the starting file position of the record as
    ' Start_Pos.

    Get #1, Start_Pos, Empl.ID          ' Fixed -- no need to adjust.
    Get #1, , Empl.MonthlySalary        ' Fixed -- no need to adjust.
    Get #1, , Empl.LastReviewDate       ' Also fixed-length.

    Get #1, , StrSize                       ' Gets the field size.
    Empl.FirstName = Input$(StrSize, #1)    ' Reads the field.

    Get #1, , StrSize
    Empl.LastName = Input$(StrSize, #1)

    Get #1, , StrSize
    Empl.Title = Input$(StrSize, #1)

    Get #1, , StrSize
    Empl.ReviewComments = Input$(StrSize, #1)

    Position = Seek (#1)        ' Gets and saves the current file
                                ' position to a global variable.
End Sub
```

As you can see from the preceding examples, **Put** is the primary output function you use to write data while using binary access. **Get** and **Input$** can both be used to retrieve data.

Keeping Track of Record Locations

No matter which access method you use, you must have some way of keeping track of where records are in a file. Otherwise, you'll have to read from the beginning of the file every time you want to locate and retrieve data, which can slow down an application considerably.

Random access has some advantage here, because Visual Basic keeps track of the locations of records based on their position in records from the beginning of the file. All you do is supply the appropriate reading or writing function with the record number, and the function takes care of finding where that record begins. However, this helps only if you are attempting to access a record by its order number. It doesn't help if you are trying to locate a record by a field value, like the employee's last name.

If you are using binary access, data is located at unpredictable (uneven) locations. In many cases, binary data won't even be organized into records. In any event, you'll probably want to use some mechanism to keep track of where various data elements are located in the file.

One way to do this is to create a *file location table* (a form of an index). For example, assume you need to have fast access to any employee record, and you will be using the employee's last name to locate the desired record. You begin by building a new data type and then use that data type to create a table:

```
Type Employee_Table
    Dim LastName As String * 15
    Dim FileLoc As Long
End Type

Const TABLESIZE = 100
Dim Emp_Table(1 To TABLESIZE) As Employee_Table
```

This creates a table that you can use to look up an employee by his or her last name. The table provides the byte location of the beginning of the appropriate record. In this example, it will hold up to 100 employees.

You can create and load this table each time you open the employee file. The following example reads the entire file, record by record, using the Input_A_Record procedure presented earlier in this chapter.

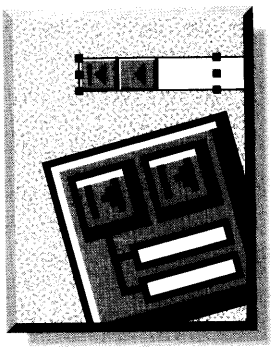
Just prior to reading each record, the current file pointer location is saved. Then, after a record is read, the current last name value is saved in the table.

```
Sub Build_Emp_Table ()
    Dim i As Integer

    Open "EMPLOYEE.FIL" For Binary As #1
    Position = 1                ' Initializes a global variable value.
    For i = 1 To TABLESIZE
        If EOF(#1) = True Then
            Emp_Table(i).LastName = "NO_MORE"
            Exit Sub
        End If
        Emp_Table(i).FileLoc = Position      ' Saves the file location.
        Input_A_Record (Position)            ' Reads the record.
        Emp_Table(i).LastName = Empl.LastName' Saves the last name
                                             ' from the record.
    Next
End Sub
```

You now have an index into the file and can use the `Position` value to directly access any employee record.

CHAPTER 20

Accessing Databases with the Data Control



With the data control, you can create applications to display, edit, and update information from many types of existing databases. Creating a data-aware application with Visual Basic can take just a few steps and doesn't require you to write any code.

To make an application data-aware, you first add the data control to a form and specify the database from which you want to get information. Next, you add "data-aware" controls such as text boxes to the form, and set properties to "bind" these controls to the data control. When you run the application, these "bound" controls will automatically display fields from the current record in the database.

Visual Basic implements data access by incorporating the same database engine that powers Microsoft Access™. Combined with Visual Basic, this technology gives you seamless access to many standard database formats, including Btrieve®, dBASE®, Microsoft FoxPro®, and Paradox®.

Contents

- Getting a Quick Start
- Using the Data Control
- Opening a Database
- Working with Bound Controls
- Manipulating Records with Code
- Controlling Transactions, Validation, and Data Updates
- Working with Database Design
- Working with Database Structure

BIBLIO.MAK and DATAMGR.EXE

Many of the code examples in this chapter are taken from the sample database application, BIBLIO.MAK. If you installed the sample applications, you will find this application in the \BIBLIO subdirectory of the main Visual Basic directory (\VB\SAMPLES\BIBLIO). The database used by the examples in this chapter is BIBLIO.MDB. This database also serves as a bibliography of current books on the subject of databases. The database itself is located in the \VB directory.

Visual Basic also includes DATAMGR.EXE, a program that allows you to create databases that you can use with the data control. Databases you create with this tool have the same format as Microsoft Access databases. You will find DATAMGR.EXE in the \DATAMGR subdirectory of the main Visual Basic directory (\VB\SAMPLES\DATAMGR).

Getting a Quick Start

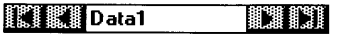
The following procedure gives you a brief overview of how to use the data control in a Visual Basic application. The example uses the BIBLIO.MDB sample database supplied with Visual Basic.

▶ **To use the data control in an application**

1. Select the data control in the Toolbox and draw a data control on a form.

   The data control icon looks like this:

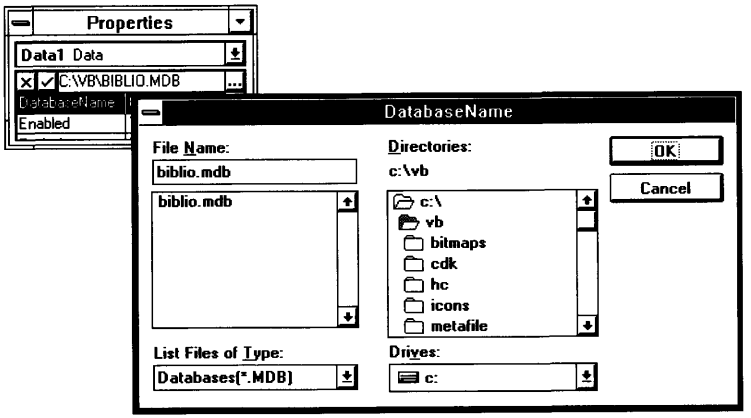
   After you draw the control on the form and size it, the caption appears as shown. The default name of the control is Data1.

2. In the Properties window, set the DatabaseName property to the file or directory name of the database to which you want to connect.

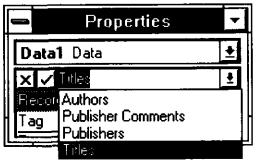
   If your database is not available during design time, you will need to fill in the DatabaseName and RecordSource properties at run time. The example uses the BIBLIO.MDB sample database.

Note To access one of the supported external database formats, be sure to set the Connect property first, as described later in the section, "Opening a Database."



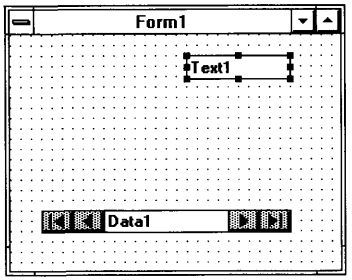
3. Set the RecordSource property to the name of the database table you want to access.

   If the database is currently available, Visual Basic provides a drop-down list of table names to select from. If your database could not be found, the drop-down list will not appear. The example uses the Titles table.

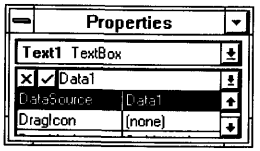


4. Draw a text box on the form to display the database information.

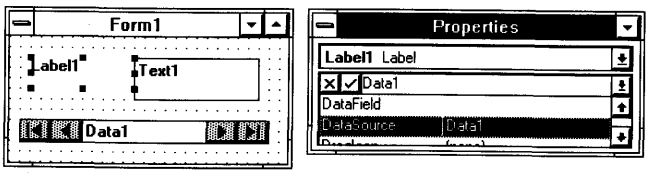
   This control will be used to display and edit a selected field from the database. You can also use other data-aware controls, including check boxes, picture boxes, image controls, and labels.



5. In the Properties window, set the DataSource property for Text1 to the name of the data control (Data1). This binds the text box to the data control.

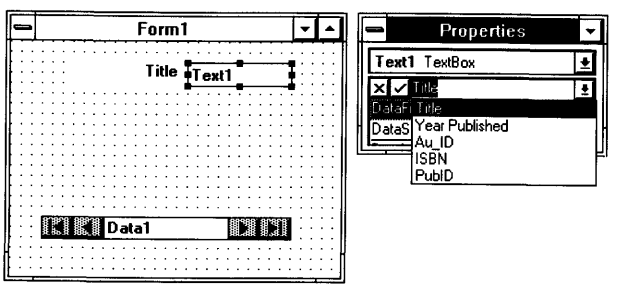


6. Add a label to indicate the name of the database field this text box will expose. Set the DataSource property for Label1 to Data1 to bind the label to the data control.

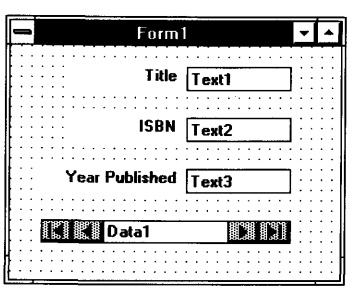


7. Set the DataField property for Text1 to the name of the field in the database table you want to view or modify.

   The example uses the Title field of the Titles table.



8. Repeat steps 4, 5, 6, and 7 for each additional field you want to access. In the example below, the Title, ISBN, and Year Published fields have been selected from the Titles table.



Now run the application. You can use the four arrow buttons on the data control to move to the beginning of the data, to the end of the data, or from record to record through the data. Figure 20.1 shows the action carried out by each button.

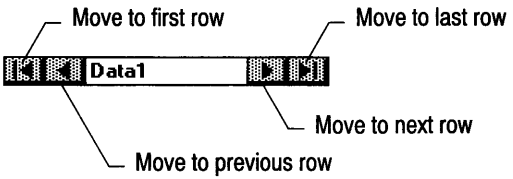


Figure 20.1 The data control and arrow buttons

You can modify the information in the database by changing the value displayed in one of the bound controls. When you click a button on the data control to move to a new record, Visual Basic automatically saves any changes you've made to the data.

Of course, you can add code to enhance your application further. The rest of this chapter shows you how to use the data control with bound controls to manipulate data, examine the structure of the database, and write event procedures to handle events that occur as data is accessed or updated.

Using the Data Control

Using the data control, you can connect a Visual Basic application to a database file and retrieve a set of records. The data control can perform the following tasks without the use of code:

- Connect to a local or remote database.
- Open a specified database table or define a set of records based on a Structured Query Language (SQL) query of the tables in that database.
- Pass data fields to bound controls, where you can display or change the values.
- Update a database based on any changes you make to data displayed in the bound controls.
- Capture Error events that occur as data is accessed.
- Close the database.

As you saw in the previous section, you add the data control to your forms just as you would any other Visual Basic control. You can have as many data controls on your form as you need. As a rule, you will use one data control for each database table that you need to manipulate.

Opening a Database

To access the information in a database, you need to specify the name of the database you want to open. This can be done at design time or while your program is running. You will be able to open Microsoft Access or external FoxPro, Btrieve, Paradox or dBASE databases by setting the DatabaseName and Connect properties of the data control. The following sections discuss how to set data control properties and how to choose the appropriate database format.

Setting Database Properties at Design Time

At design time, you specify the name of the database by setting data control properties. At a minimum, you need to set either the DatabaseName or Connect property. When you do this, Visual Basic immediately attempts to retrieve the names of all tables and available queries in the database. These names are displayed in the Settings list for the data control's RecordSource property. Once you choose a table or query from the list, you will be able to use the drop-down menus to select a valid DataField setting on each of your bound controls.

Setting Database Properties at Run Time

At run time, you can set or change the data control and bound control properties to indicate which database table and fields are to be manipulated. Once the correct properties are set, you can manually open the database using the **Refresh** method. Once your form is loaded, Visual Basic automatically attempts to retrieve the names of all tables in the named database. Once you choose a table from the list or enter a query, you can open the database specified by the DatabaseName or Connect property if it has not already been opened with the **Refresh** method.

If the database cannot be opened, the Error event will occur. For example, if the database has already been opened in Exclusive mode by another user, or the database cannot be located, Visual Basic generates an error. Your code will need to deal with each of these and other possible database errors.

Identifying Database Characteristics

The following table shows database-related properties you can set at design time or at run time (as noted).

Property	Determines	Remarks
Connect	The type of database.	Not needed for Microsoft Access databases. Specifies other types of standard databases.
DatabaseName	The data source name. Identifies the location of the database file.	Sets the path and file name or directory for the database file (.MDB or .DDF).

Property	Determines	Remarks
Exclusive	Single- or multi-user access to the database.	**True** (Single-user) or **False** (Multi-user). Defaults to **False**.
Options	Characteristics of the recordset created by the control.	See the section "The Options Property" later in this chapter.
ReadOnly	Read/write access to the database.	**True** (read-only) or **False** (read/write). Defaults to **False**.
RecordSource	The name of the database table to access or the text of a SQL query.	Must be a valid table name in the specified database or a valid SQL query.

The Connect Property

The Connect property specifies the type of database and, in some cases, additional parameters like the database password. Note that multiple parameters are separated by semicolons and that no Connect setting is needed to open Microsoft Access databases. Connect strings are not case-sensitive.

Database format	DatabaseName setting	Connect setting
Microsoft Access	drive:\path\file.MDB	—
FoxPro version 2.0	drive:\path\	FoxPro 2.0;
FoxPro version 2.5	drive:\path\	FoxPro 2.5;
dBASE III®	drive:\path\	dbase III;
dBASE IV®	drive:\path\	dbase IV;
Paradox	drive:\path\	paradox; pwd=password;
Btrieve	drive:\path\.DDF	btrieve;
ODBC (*Professional Edition only*)	*Registered data source name (server)*	*odbc;dsn=server;uid=user; pwd=password*

Note For information on accessing external databases, see the text file EXTERNAL.TXT, located in the \VB root directory. Access to Open Database Connectivity (ODBC) databases is not supported in the Visual Basic Standard Edition.

The DatabaseName Property

The data control locates the database using the DatabaseName property. Databases can be physically located almost anywhere. For example, your data can be:

- In a database file on your local disk.
- In a database file on a remote networked server. The database can also be shared by a number of other users.

It is easiest to work with a database if the database you want to open, or a copy of it, is available at design time. While you are setting properties for the data control, Visual Basic will try to access the database file specified in the DatabaseName property. If successful, Visual Basic will populate the *RecordSource list*—a list of all tables in the database. If the database is unavailable, Visual Basic will not be able to show you the names of the database's tables, but you can still type the name of the table or query you want to open at run time into the RecordSource property.

When using Microsoft Access databases, you may also find that the database itself has *attached* tables that can physically reside on other systems or be drawn from other databases at run time. Visual Basic automatically retrieves this data without any special handling on your part. Microsoft Access databases can also contain queries that you can manipulate as you would a table.

The Exclusive Property

You control access to the entire database by using the Exclusive property. Sometimes you may want exclusive access to a database. For example, you may want to temporarily control access while you build a report. You should set the Exclusive property to **True** whenever you expect to be the only user. Doing so will give you considerably faster access to your data.

If you want your application to have exclusive access to a database, set the Exclusive property to **True**. Then, when you then open the database, no other application will be able to open it. When you close the database, this lock will be released. If you attempt to open a database "exclusively," and the database has already been opened by another application, the open operation will fail.

If you want to open the database and share access to its information with other applications or users, the Exclusive property must be set to **False** *before* the database is opened—this is the default mode. If you change the Exclusive property after the database has been opened, no change will take place until you execute the data control's **Refresh** method, which closes and reopens the database.

The Options Property

Once your database is opened, the data control provides a **Recordset** object through the Recordset property. **Recordset** objects are used to manipulate the data in the database—they are discussed later in this chapter.

To control access to the database tables accessed by the recordset your data control creates, you can set one or more option bits with the Options property. These options permit you to deny access to the tables in the database containing the records you have selected.

Note Since the recordset records may reside on several tables, the options settings may have a far-reaching effect. For example, if you set the DenyRead or DenyWrite options, all affected tables will be locked until you close your recordset or database. Generally, the options that deny access to the tables should be used for administrative purposes only.

The following Visual Basic constants are valid settings of the Options property. You can add multiple Options values together and then provide them as a single parameter, as shown in the following table.

Constant	Value	Determines
DB_DENYWRITE	&H1	If other users can write to the tables in the recordset.
DB_DENYREAD	&H2	If other users can read the tables in the recordset.
DB_READONLY	&H4	If this application can write to the tables in the recordset.
DB_APPENDONLY	&H8	If this application can only add to, and not read from, the recordset. This option does not affect **Table** object operations.
DB_INCONSISTENT	&H10	How multiple table updates are made (search Help for *Inconsistent*).
DB_CONSISTENT	&H20	How multiple table updates are made (search Help for *Consistent*).
DB_SQLPASSTHROUGH (*Professional Edition only*)	&H40	If query is to be processed by external database server.

For example, if you want to open a recordset on a table and do not want other users to write to the table while you have the recordset open, you can set the Options property as follows:

```
Data1.DatabaseName = "BIBLIO.MDB"       ' Name the database.
Data1.RecordSource = "Titles"           ' Build a recordset on Titles.
Data1.Options = DB_DENYREAD             ' Others cannot read the table.
Data1.Refresh                           ' Open the database.
```

For More Information For information on any of the options listed in the preceding table, see the *Language Reference,* or search Help for *Options*.

The ReadOnly Property

If you want to open a database and do not plan to modify the data or structure, you can set the ReadOnly property to **True**. If you change the ReadOnly property, the database status will not change until the **Refresh** method is executed against the data control. The ReadOnly property does not affect the operation of other applications using the database, but it will ensure that your application does not write to the database.

The RecordSource Property

The RecordSource property indicates where in the database the data will be found. This property does not take effect until the database is opened. You can use the RecordSource property to specify one of three different sources:

- The name of a database table.
- The name of a Microsoft Access database query.
- The text of a SQL query.

At design time you can choose from a list of database tables and queries by first specifying the DatabaseName and (if required) Connect properties. At run time, you can also specify a table name or a query name, or you can use the RecordSource property to specify a SQL query that will return a set of records from your database. For example, the following SQL query returns all of the columns in the bibliography database for publishers based in New York:

```
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Select * from Publishers where state = 'NY' "
Data1.Refresh
```

Working with Bound Controls

Bound controls are the data-aware controls through which you access information in a database. Three new properties have been added to the text box, check box, image, label, and picture box controls to enable them to be "bound" to the data control. When a control is bound to the data control, Visual Basic applies field values from the current database record to that control. In turn, the control displays data to you and accepts your changes. If you change data in a bound control, those changes can be automatically written to the database as you move to another record.

Types of Bound Controls

Visual Basic supports five built-in controls that you can bind to the data control. Other data-aware controls are available from third parties and in the Professional Edition of Visual Basic. Check with your control vendor for more details. The following table lists these controls.

Control	Icon in Toolbox
Check box	
Image	
Label	
Picture box	
Text box	
Masked edit (*Professional Edition only*)	
3D panel (*Professional Edition only*)	
3D check box (*Professional Edition only*)	

Bound controls are characterized by three data-aware properties: DataChanged, DataField, and DataSource.

Property	Description
DataChanged	Indicates whether a value displayed in a bound control has changed.
DataField	Specifies the name of a field in the recordset created by the data control.
DataSource	Specifies the name of the data control to which the control is bound.

Note All data events are managed by the data control.

Adding Bound Controls to a Form

As you saw in earlier in this chapter, you must draw bound controls on the *same form* as the data control to which they are bound. After you draw a bound control on a form, you first set the DataSource property to bind it to the data control. Next you set the DataField property. If the database is available at design time, a list of valid fields will be available when you specify the DataField property. All valid fields in the specified table are displayed in the drop-down list in the Settings box in the Properties window. If the database is not available at design time, you'll need to provide a valid field name at run time before data values will be posted to the control from the database. Once a valid field name is provided for an opened database, you will see the contents of the database field and be able to edit the value immediately.

Tip You can have more than one bound control for a particular field, but you do not need to provide a bound control for each field in the table. Neither the data control nor the bound controls need to be made visible, so you can incorporate data access capabilities into any form you design.

Using Bound Controls at Run Time

When you run your application, the data control works together with the database to give you access to the current set of records, or *recordset*, with which you are working. Using the buttons on the data control, you can move from record to record, and using the bound controls, you can view or edit the data displayed from each field. Whenever you click a button on the data control, Visual Basic automatically updates any changes you've made to the recordset, as shown in Figure 20.2.

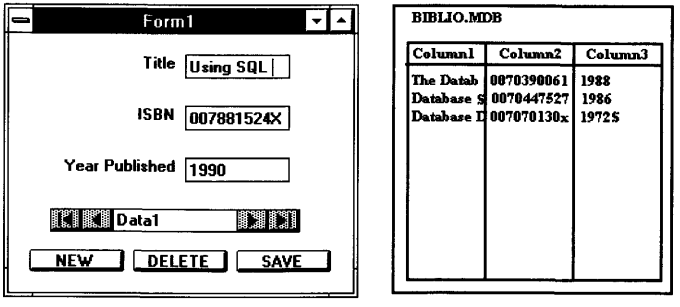


Figure 20.2 Displaying records on a form

If you want to add records to or delete records from the database, you'll need to write code.

Adding a New Record

To add a new record, you use the data control **Recordset** object's **AddNew** method. In the preceding illustration, for example, the New command button carries out this action. The code for the New button looks like this:

```
Data1.Recordset.AddNew
```

When you click the New command button, Visual Basic adds a new record to the end of the recordset. At this point, all bound controls are cleared, and you can fill in the new record values. After you add a new record, you need to save the information by using the **Update** method or by clicking one of the data control buttons to move to another record. Notice that using the buttons on the data control or one of the Move methods to move to another record will automatically save your added record.

Saving Data with the Update Method

After you edit a record, you need to save the changed information to the database by using the **Update** method on the recordset. You can also use the **Update** method after using the **AddNew** method, instead of using one of the Move buttons on the data control. Your attempt to add or change a record may fail if:

- Your table has a unique index and this record is already in the table.
- One of the index key fields in your record is Null.
- You do not have permission to write to the table or database.
- The table is not updatable.
- The record is on a page that is locked.
- The contents of the bound control do not match the Field definition in the database.
- The change violates one of the Microsoft Access database integrity rules.

Each of these conditions will generate a trappable error—in the data control's Error event—that your application will need to be able to deal with. For example, an error is triggered if the length of the text you supply in your bound text control is too long or if you define a numeric field and the number is too large. By default, the Error event displays an error message to indicate the problem.

The Save button in Figure 20.2 uses the following code:

```
Data1.Recordset.Update
```

Unless you end your application or have a pending transaction, any operation that moves the current record pointer to another record will automatically save any pending changes. Transactions are discussed later in this chapter.

If you want to abort the **AddNew** method, simply perform another **AddNew** method and move to another record. You may also perform the **UpdateControls** method to restore the previous values of the bound controls.

Deleting a Record

You can delete a record by using the **Delete** method. To do this, you first move the data control to the record you want to delete, and then execute the **Delete** method. There is no need to use the **Update** method after the **Delete** method. The **Delete** method deletes the current record from the database and makes the current record invalid. At this point, you must move to another record in your recordset. Any attempt to change the contents of the deleted record will result in a trappable error. Again, referring to Figure 20.2, the code for the Delete button looks like this:

```
Data1.Recordset.Delete
Data1.Recordset.MoveNext
```

Using Bound Picture Box and Image Controls

One of the most appealing aspects of Visual Basic applications is their ability to display graphics. Using the picture box or image controls, you can store graphics in a database and recall them with little or no coding.

To set up an application to manipulate graphics using a bound picture box control, follow these guidelines:

- Add a picture box or image control to your application and bind it to the data control.
- If you want to be able to paste images from the Clipboard into your picture box control, add the appropriate logic. Examples for Cut, Copy, and Paste operations can be found in Chapter 17, "Interacting with the Environment." You may also want to add a mechanism to indicate when the picture box is selected (that is, when it has the focus).

When you paste in a new picture, the DataChanged property of your picture box control will be set to **True.** When you move the current record pointer, the binary data in your picture will be saved to the database automatically.

Note You can also use the **LoadPicture** function to load a data file containing a graphic into your picture box control. You may also copy the image read in from your database into the Clipboard or use the **SavePicture** statement to save the binary data to a file.

Manipulating Records with Code

Visual Basic has implemented specific data objects, properties, methods and events that allow you to work with the recordsets that the data control creates. For example, if you want to write code to move to the last record in the recordset, you can treat the recordset as an object, and then apply the **MoveLast** method to it:

```
Data1.Recordset.MoveLast
```

Or, if you want to check the value of a specific field in the current record, your code might include the following line:

```
String$ = Data1.Recordset.Fields("Title").Value
```

You'll notice here how the syntax becomes a concatenation of object names that point to the specific value you want to assess. As you write code to work with data, you'll find that this extended syntax offers you a fine degree of control over how you choose to view or manipulate the recordset.

Using the Data Control Refresh Method

When working with the data control at run time, you'll need to use the **Refresh** method when you want to open a database, or when you want to reopen a database with a new set of data control properties. The **Refresh** method will attempt to open the database and build a new recordset based on the data control properties shown in the following table. If the database or table cannot be found, a trappable error will result. Once the database is opened, your bound controls will display the values from the first record of the recordset.

To open a new database, or to reopen a database with changed properties, Visual Basic:

- Closes an open database with the **Refresh** method applied to the data control.
- Opens or re-opens the database specified by the DatabaseName and Connect properties.
- Creates a new recordset from the selected database based on the RecordSource property.

Depending on the state of the data control properties (as shown in the following table),Visual Basic will either close or reopen the database and completely rebuild the recordset. In all cases, the **Refresh** method uses the current values of the data control properties.

Property change	Operation at Refresh
Connect	Database reopened; recordset rebuilt.
Exclusive	Database reopened; recordset rebuilt.
ReadOnly	Database reopened; recordset rebuilt.
DatabaseName	Database reopened; recordset rebuilt.
RecordSource	Recordset rebuilt.
Options	Recordset rebuilt.

If you simply want to recreate the recordset, and none of the other properties have changed, you can use the **Refresh** method to recreate the same recordset with fresh data from the database. In some cases, as with multi-user databases, it makes sense to do this periodically to make sure that your recordset has the latest information from the database.

If the **Refresh** method fails to open your database, a trappable error will result. However, you can also determine if a database is open by using the **Is** function to test if the Recordset or Database properties are valid. For example:

```
Data1.DatabaseName = "BIBLIO.MDB"       ' The right dbname file name...
Data1.RecordSource = "Foo"              ' An invalid table name
Data1.Refresh       ' This call will fail if the dbname is invalid.
If Data1.Database Is Nothing Then
MsgBox "Open Failed (Database invalid)"
End If
If Data1.Recordset Is Nothing Then
    MsgBox "Open Failed (Recordset invalid)"
End If
```

Managing the Current Record

Only one record is accessible to each data control at any one time, and this is the *current* record. As you work with records, you may need to ensure that the current record is always a valid record. It's possible to position the current record on a deleted record, for example, or to position it beyond either end of the recordset, thus making it invalid.

To make sure that the current record falls between the beginning of file (Data1.Recordset.BOF) and the end of file (Data1.Recordset.EOF), you can write code to check for the following states.

Recordset BOF/EOF Property	Meaning
BOF and EOF both **False**	The current record pointer is valid unless you have not moved after deleting the last record in the recordset.
BOF = **True**	The current record is positioned ahead of first record of data. Current record pointer is invalid.
EOF = **True**	The current record is position behind last record of data. Current record pointer is invalid.
BOF and EOF both **True**	There are no rows in the recordset. Current record is invalid.

Moving Around a Recordset

Moving around refers to changing the current record in a recordset. You have already seen how to move from record to record using the data control; this section discusses the Recordset methods that you can use to carry out these same actions.

The following table shows which Recordset methods are aligned with which buttons on the data control.

To move the current record to	Press	Use this Move method
The first record in the recordset		Data1.Recordset.**MoveFirst**
The previous record in the recordset		Data1.Recordset.**MovePrevious**
The next record in the recordset		Data1.Recordset.**MoveNext**
The last record in the recordset		Data1.Recordset.**MoveLast**

Moving to the First or Last Record

It is often necessary to jump to the first or last record in a recordset. For example, you may want to do a sequential search for a record containing a particular field value, and need to start at one end or the other of the recordset.

One way to move to the first record is to use the **Refresh** method. Each time you use **Refresh**, the data control automatically loads the first record and makes it current:

```
Data1.Refresh
```

You can also use the **MoveFirst** method to accomplish the same thing:

```
Data1.Recordset.MoveFirst
```

To move to the end of the recordset, use the **MoveLast** method:

```
Data1.Recordset.MoveLast
```

If the data control is positioned at either the first or last record of the recordset, any further movement toward the beginning or end will set the BOF (Beginning of File) or EOF (End of File) flags to **True**. At this point, you will not have a valid current record and the bound controls will be cleared. If you use a Move method that moves beyond BOF or EOF, Visual Basic triggers a trappable error. Because of this design, you can code the following routine that safely traverses the database from any point to the end:

```
Do While Data1.Recordset.EOF = False
    Data1.Recordset.MoveNext
    ' Insert your code to work with the current record...
Loop
```

At the end of this loop, the current record pointer is invalid.

Moving to the Next Record

The **MoveNext** method makes the next record in the recordset current. Generally, **MoveNext** is used to step through a recordset's rows to extract data on a record-by-record basis.

The following example loads a grid control with the first and last name of every customer:

```
Dim GridRow As Integer
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Titles"
Data1.Refresh
GridRow = 1                                    ' Point to first grid row.
Do While Data1.Recordset.EOF = False           ' Loop through recordset.
    Grid1.Rows = GridRow + 1                   ' Add a new record.
    Grid1.Row = GridRow                        ' Move to it.
    Grid1.Col = 0                              ' Point to first grid col.
    Grid1.Text = Data1.Recordset("Title")      ' Fill in data from Title.
    Grid1.Col = 1                              ' Point to next grid col.
    Grid1.Text = Data1.Recordset("ISBN")       ' Fill in data from ISBN.
    Data1.Recordset.MoveNext                   ' Move to next record.
    GridRow = GridRow + 1                      ' Point to next grid row.
Loop
```

Moving to the Previous Record

The **MovePrevious** method makes the previous record current. Generally, this method works like the **MoveNext** method, except that it moves the current record pointer toward the front of the recordset.

Using a Bookmark to Move to a Specific Record

Bookmarks allow you to save a current record pointer and reposition directly to a specific record. The Bookmark property contains a pointer to a record you specify; you can jump to that record by setting the Bookmark equal to the value of that pointer. This value can be saved in a **Variant** or **String** variable. The following code repositions the current record to a previously saved Bookmark:

```
Dim MyBookMark as Variant
MyBookMark = Data1.Recordset.Bookmark   ' Save current record pointer.
Data1.Recordset.MoveFirst               ' Move off the record...
Data1.Recordset.Bookmark = MyBookmark   ' Move back to saved location.
```

If you don't know the physical order number or Bookmark of the record you want to access, you can search for it by starting with the first record and loop through the recordset, comparing data from each record with the item you want to find.

Caution If more than one user is accessing the database, the record order can change when you use the **Refresh** method to rebuild the recordset. If you depend on the record number (counting from the top of the recordset) to locate records, you will find that this number will not consistently bring you back to the same record. Bookmarks save a pointer to the record that can be used to retrieve a specific record as long as that record remains in the recordset and you do not **Refresh** the recordset. If you or some other user deletes the record, the Bookmark becomes invalid, and Visual Basic generates a trappable error.

Updating a Recordset

To determine whether the data in a recordset can be changed, you examine the Updatable property of the recordset. If this property is **True**, the recordset will accept changes. For example, to see if a selected table is updatable, you could write the following code:

```
If Data1.Recordset.Updatable = True Then ...
```

Note that the recordset will not be updatable if:

- The data control was opened with the ReadOnly property set to **True.**
- Another user (or your own application) is referencing the recordset with the DenyWrite option bit set.
- The recordset was based on a query where not all columns are updatable.

Adding, Editing, and Deleting Records

Visual Basic provides a number of methods that you can use to add, edit, or delete records in a database. This section describes these methods and explains how they affect the contents of a recordset.

To change the information in a database, you must be using a database that is updatable. Generally, if the database is not updatable, the underlying recordset and fields are not updatable. If the recordset is updatable, all underlying fields are updatable. If you're not sure whether your database can be updated, you can check the following conditions:

- The Updatable properties of the database, recordset, and field must be **True**. If the Updatable properties are **False**, the recordset's data values are read-only.
- You must open the data control with the ReadOnly property set to **False**.

The following code checks to see if a database can be updated:

```
If Data1.ReadOnly = True or Data1.Database.Updatable = False Or
➥   Data1.Recordset.Updatable = False Then
    MsgBox "This data cannot be altered"
End If
```

To check the ability of a field to accept changes, you will need to examine the Attributes property and test the DB_UPDATABLEFIELD bit. For example:

```
If Data1.Recordset.Fields("First_Name").Attributes And
➥   DB_UPDATABLEFIELD = 0 Then
    MsgBox "This field cannot be altered"
End If
```

Adding New Records

Once you have determined that the database and recordset can accept changes, you are ready to add records.

To append new records to your recordset

1. Create a new (blank) record with the **AddNew** method. The current record pointer is saved and moved to the new record.
2. Assign new values to the fields in the new record.
3. Save the new record with the **Update** method. The current record pointer is restored to its original value.

The following code adds a new title to the Titles table of the BIBLIO.MDB database.

```
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Titles"
Data1.Refresh

Data1.Recordset.AddNew                    ' Create a new, empty record.
Data1.Recordset("Title") = "The Data Control"    ' Set the field values.
Data1.Recordset("Year Published") = "1993"
Data1.Recordset("AU_ID") = 37
Data1.Recordset("ISBN") = "2344456533"
Data1.Recordset("PubID") = 43
Data1.Recordset.Update                          ' Append the new record.
```

Note that Visual Basic makes no attempt to validate the accuracy of "foreign" keys. For example, in the preceding code example, the PubID field refers to the PubID field in the Publishers table. Your code must verify that this is a correct value in order to maintain database referential integrity. Validation of the fields is best done by use of the data control's Validation event. There you can check to see if the field values about to be written to the database are correct.

After using the **AddNew** method, you must update the recordset before using the **Close** methods. When using the data control, this is fairly easy, because any method that moves the current record pointer, including pressing any data control button, will save the new (or changed) record before moving the current record pointer. In the preceding example, the following code adds the record to the recordset:

```
Data1.Recordset.Update
```

Editing the Current Record

To change data in your database, you must first make the record you want to edit the current record. You can use any of the Move methods or data control buttons to position the record pointer to the record you want to change. Then make whatever changes are needed in the bound controls. Remember, the same restrictions apply here as far as updatable fields are concerned—not all fields are updatable. To save your changes, simply move the current record pointer to another record, or use the **Update** method as shown in the preceding code example. Records are edited "in place," so the current record pointer or Bookmark will not change after the edit.

▶ **To edit one or more field values of the current record**

1. Position the current record to the record you want to edit.
2. Assign new values to the fields you want to change.
3. Use the **Update** method or any of the Move methods

   –or–

   Press one of the buttons on the data control to save the changes and replace the existing field values.

The following code shows how you can edit the value contained in the Customer ID field in the first record. Note how you can change the data in the recordset or in the bound control itself:

```
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Titles"
Data1.Refresh                          ' Open the database.
Data1.Recordset("PubID") = "12345678"      'Change the value...
Text4.Text = "New title"              'or change it in the bound control.
Data1.Recordset.Update                 ' Save the changes.
```

Editing the Value of a Specific Field

A recordset can be broken down by fields, with each field representing a field of the database table. The set of all **Field** objects for a recordset is referred to as the **Fields** *collection*. The following code shows a number of ways to refer to the Value property of a particular field within a collection. As a shortcut, you can omit the .Field and .Value notations, since they are default properties. Generally, you will find that the last two forms are easiest to use.

```
Data1.Recordset.Fields(0).Value ' Refers to the value of the first field.
n = 0                           ' Indirect reference by field index number.
Data1.Recordset.Fields(n)       ' Refers to the value of the first field.
Data1.Recordset.Fields("FName") ' Refers to the value of the FName field.
string$ = "FName"               '`Indirect reference by field name.
Data1.Recordset.Fields(string$) ' Refers to the value of the FName field.
Data1.Recordset("FName")        ' Fields is the default collection of the
                                ' Recordset, so it can be omitted.
Data1.Recordset(n)              ' You can also reference collection
                                ' members by number as well.
```

Note that only one recor is current at any one time and that only one record from each recordset is visible to your application.

The `Data1.Recordset.Fields(n).Value` property, which can be abbreviated as `Data1.Recordset(n)` or `Data1.Recordset("Field name")`, contains the data retrieved from the database.

The following example shows how you can display the values of all of the fields in the selected recordset.

```
For i = 0 to Data1.Recordset.Fields.Count - 1
    MyList.AddItem Data1.Recordset(i)      ' Add the field to the list.
Next
```

Deleting Records

To delete an entire record, position the current record pointer to the record you want to remove and use the **Delete** method. For example, the following code will delete from the Titles table in the BIBLIO database every record whose "Year Published" field is less than 1889:

```
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Select * from Titles where
➥   [Year Published] < 1889;"
Data1.Refresh

Do While Data1.Recordset.EOF = False
    Print "Deleting title:"; Data1.Recordset("Title")
    Data1.Recordset.Delete
    Data1.Recordset.MoveNext
Loop
```

In the example, each iteration performs the **MoveNext** method. You need to use **MoveNext** to change the current record after a deletion, as a deleted record no longer contains valid data, and an attempt to access this data will result in an error.

Handling Data Errors

- Any error caused by user interaction with the data control triggers an Error event. Errors caused by other statements, methods, or functions will trigger a trappable run-time error.

The Error event is passed a *DataError* argument, which is a number associated with the error. A text string explaining the error is available through use of the **Error$(DataErr)** function. If you do not want Visual Basic to display the message (the default behavior), set the Response code set to 0.

Handling Large Data Fields

Memo field data types can handle strings that exceed the 64K size limitation of Visual Basic string variables. You can use three special Field methods to read and write to Memo fields: **FieldSize**, **GetChunk**, and **AppendChunk**.

If you are reading a Memo field, you need to know how large it is. The **FieldSize** method returns this value. If the size is greater than 64K, you need to use the **GetChunk** method to access the Memo field a piece at a time. For each portion you want to access, you specify the starting place in the Memo field as well as how many bytes you want to retrieve.

The **AppendChunk** method appends a string to the end of an existing Memo field.

For More Information For information on the exact syntax of these commands, see the *Language Reference* or search Help for *FieldSize*, *GetChunk*, or *AppendChunk*.

Closing a Recordset

The **Close** method closes the recordset and frees the resources allocated to it. Attempting to perform a method or access an element of a closed recordset results in a trappable run-time error. For example, the following code closes a recordset:

```
Data1.Recordset.Close
```

Databases and their respective recordsets are closed when:

- You use the **Close** method against a specific recordset.
- The form containing the data control is unloaded.
- The program executes an **End** statement.

The Validate event, which is discussed later in this chapter, is triggered when either the **Close** method is used or when the form is unloaded. Validate is not triggered when the program executes an **End** statement. Last-minute cleanup operations can be performed in the Validate event.

Controlling Transactions, Validation, and Data Updates

Transactions, field validation, and the **Update** and **Refresh** methods are key tools you can use to maintain the integrity of information in your database. The following section describes how these features are implemented for Visual Basic data access.

Controlling Transactions

A *transaction* is a recoverable series of changes you make to a recordset. You use transactions when you want to verify any changes you make before committing the new information to the database. For example, if you are dealing with a lengthy series of financial transactions, you might want to cancel your changes if the final totals are out of balance.

You use code to explicitly begin a transaction. While a transaction is open, any changes you make to data can be undone, or "rolled back." When you decide that your work is complete, you can save, or "commit," the changes to the database.

When you first open a database and no transactions are pending, the transaction state is *auto-commit*, which means that all changes made to a recordset are made immediately to the underlying table and are irreversible. For situations where this is not what you want to do, you can use , transactions to control when the changes will occur.

Visual Basic has three statements that support transaction processing: **BeginTrans**, **CommitTrans**, and **Rollback**. The database itself must also support transactions, however, or these commands will be ignored. You should always make sure that the Transactions property of the database is set to **True** before using these statements.

Transactions span databases. That is, when you use one of the transaction statements, it applies to all databases—even databases opened after the transaction was begun. When you use **CommitTrans** or **Rollback**, all pending transactions, regardless of the database, are either committed or rolled back.

Beginning a Transaction

The **BeginTrans** statement marks the beginning of a transaction and takes the next sequence of operations out of auto-commit mode. Once you begin a transaction, you must use **CommitTrans** or **Rollback** before you close the database or end the program. Uncommitted transactions are automatically rolled back when your program ends. If you attempt to close a database while a transaction is in progress, an error will occur.

Saving Changes to a Table

CommitTrans saves all recordset changes made since the transaction was opened with the **BeginTrans** command. When you execute **CommitTrans,** all changes are made permanent, the current transaction is ended, and the transaction state returns to auto-commit.

Undoing Changes

Rollback reverses, or undoes, all data changed in the current transaction. It also ends the transaction and returns the transaction state to auto-commit. The following example begins a transaction, deletes all records in a recordset if the user confirms the action:

```
Dim Message As String, DgDef As Integer, Response As Integer
Dim HitCount As Integer, Msg As String
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Titles"
Data1.Refresh

BeginTrans                                      ' Turn off AutoCommit.
Do While Not Data1.Recordset.EOF                ' Loop through recordset.
    If Data1.Recordset("Title") = "Bogus" Then   ' Is this a bad title?
        Data1.Recordset.Delete                  ' Yes, delete it.
        HitCount = HitCount + 1                 ' Count how many found.
    End If
    Data1.Recordset.MoveNext                    ' Loop through recordset.
Loop
    If HitCount = 0 Then Rollback : Exit Sub        ' No hits? Just
                                                    ' Rollback and exit.
Msg = " Do you really want to delete these " & HitCount & " records?"
DgDef = MB_YESNO + MB_ICONSTOP + MB_DEFBUTTON2      ' Describe dialog.
Response = MsgBox(Msg, DgDef)                       ' Get user response.
If Response = IDYES Then                            ' Evaluate response.
    CommitTrans                                     ' Delete the records.
Else
    Rollback                                        ' Undo the deletes.
End If
```

Working with Multiple Transactions

Transaction commands always apply to all recordsets. As the preceding code example shows, this allows you to simply state the transaction command you want (such as **BeginTrans**) without reference to the data control. If you're working with multiple transactions, however, you'll need to pay attention to the sequence of the transaction commands you issue.

Once you issue a **BeginTrans** command, you have a transaction pending, and all further transaction commands apply to that pending transaction. If you then begin a second transaction, without concluding the first, you start to build a series of nested transactions. The model here is much like that of nested control structures, where you need to close the innermost statements (or transactions) before proceeding to the outermost.

Validating Changes to the Database

The data control's Validate event allows you to check any changes made to the recordset before new information is written to the database. It also allows you to specify which record will become current after the Validate event is concluded. Validate is triggered when the current row is changed, except when it is changed by the **UpdateRecord** method. This means that Validate can be triggered whether or not you have changed data in the bound controls.

During a Validate event, you will not be able to execute any method that triggers another Validate event. For example, you will not be able to execute **AddNew** or any of the Move methods.

The Validate event is executed just before Visual Basic writes changes from the bound controls to the database and repositions the current record pointer to another row in the database. The following table summarizes the arguments you can use with Validate.

Argument	Determines
Save	Whether or not the **Update** method will be executed to save the current changes.
Action	What action triggered the event; also allows you to specify what operation takes place after the Validate event.

The Save Argument

In the Validate event, you can determine if any of the bound controls have changed by examining the *Save* argument. Visual Basic automatically checks the Changed property of each bound control to see if its value has changed since it was set by the last database action. If any values have changed, Visual Basic sets the *Save* argument to **True**.

If the *Save* argument is **True,** Visual Basic will save any bound control changes to the database. If you do not want to save the changes, you can set the *Save* argument to **False**.

The Action Argument

The *Action* argument tells you what caused the Validate event to be triggered and allows you to reposition the current record pointer after the Validate event is completed. In the Validate event, Visual Basic sets the *Action* argument with a value that indicates which action triggered the event in the first place. Data constants for the *Action* argument can be found in CONSTANT.TXT. The following table summarizes the *Action* argument values and the actions caused by the Validate event.

Constant	Value	Description
DATA_ACTIONCANCEL	0	Cancel the action that caused the event.
DATA_ACTIONMOVEFIRST	1	**MoveFirst** method.
DATA_ACTIONMOVEPREVIOUS	2	**MovePrevious** method.
DATA_ACTIONMOVENEXT	3	**MoveNext** method.
DATA_ACTIONMOVELAST	4	**MoveLast** method.
DATA_ACTIONADDNEW	5	**AddNew** method.
DATA_ACTIONUPDATE	6	**Update** (not **UpdateRecord**).
DATA_ACTIONDELETE	7	**Delete** method.
DATA_ACTIONFIND	8	**Find** method.
DATA_ACTIONBOOKMARK	9	The Bookmark property has been set.
DATA_ACTIONCLOSE	10	**Close** method.
DATA_ACTIONUNLOAD	11	The form is unloaded.

In some cases, you can set the *Action* argument to specify how Visual Basic will reposition the current row pointer after the event is completed. This is possible if the Validate event has been triggered by **AddNew** or one of the Move methods.

For example, suppose the Validate event was triggered because you clicked the MoveNext button on the data control. As Visual Basic then enters the Validate event, it sets the Action property to 3, indicating MoveNext. After the validation is completed, you want to reposition the current record pointer to the previous record instead of to the next record. To do this, you set the *Action* argument to 2, indicating MovePrevious. The repositioning routine will then use the *Action* argument you specify to indicate which row to set as the current row after the transaction.

You can specify any one of the Move or **AddNew** methods to be executed in place of any other set of Move or **AddNew** methods. If you attempt to change any action except one of the Move methods or **AddNew**, Visual Basic ignores your attempt and proceeds with the originally intended operation.

Canceling the Action

If you do not want the Validate event to reposition the current record pointer to another record, you can set the *Action* argument to 0. Setting the *Action* argument to 0 has no impact on whether or not data is saved to the database, it merely cancels the repositioning operation and leaves the current record active. If no repositioning takes place, the values displayed in the bound controls and the current record pointer remain unchanged.

Updating Your Data

Throughout this chapter, you have seen the **Update** method used to update the information in the recordset, based on changes made through bound controls or through code. In this section, you'll learn about three additional methods that you can use to update information: **UpdateRecord**, **UpdateControls**, and **Refresh**. The following table summarizes the actions of these three methods.

Method	Description
UpdateRecord	Updates database (recordset) with data from bound controls.
UpdateControls	Updates database (recordset) changes to bound controls.
Refresh	Creates a new recordset based on data control properties.

The UpdateRecord Method

The data control's **UpdateRecord** method updates the recordset specified by the data control with data from the bound controls. You use the **UpdateRecord** method in situations where the recordset does not currently reflect the data in the bound controls. No events (including Validate) are triggered when this method is executed, and the current row pointer is unaffected.

The UpdateControls Method

The data control's **UpdateControls** method updates the bound controls using the values in the recordset's current row. You use this method when the current row was changed, but the bound controls were not automatically updated with data from the current row.

Working with Database Design

If you are new to databases, the following information will help you understand some of the terminology and concepts associated with database structure and design.

Relational Database Objects Visual Basic provides a *relational* interface to database files. Basically, a relational database is one that stores data of *tables*, made up of *columns* and *rows* of data. In Visual Basic, columns are referred to as *fields*, and rows are referred to as *records*.

A Visual Basic database is a collection of information defined and organized by a set of relational tables, regardless of its database file format. This means that when you use data from external databases (like FoxPro, Paradox, dBASE, or Btrieve), you can use the same relational terms.

Tables A *table* is a logical grouping of related information. For example, a table might contain a list of all known authors. The Authors table in a database about books is where information about authors, such as their names, dates of birth, addresses, and perhaps pictures, is located.

Fields Each piece of information is confined to a single column or *field.* Tables are defined by the fields they contain, with each field describing the data it is to hold. When creating a database, you assign a data type, maximum length, and other attributes to each field. Fields can also contain characters, numbers, or even graphics. For example, the Authors table might have fields with the name and address as data type character, the date of birth as data type date, and the author's photograph as data type graphic.

Records Information about individual authors is kept in the rows or records of the table with a data value assigned to each column or field. Generally, database table records are created such that no two rows are the same. This way you would not have two authors by the same last name at the same address and with the same birthday, unless they were twins!

Indexes To make access to the data faster, most databases use *indexes.* Database table indexes are sorted lists that are faster to search than the tables. Each index entry points back to the database row it references. If the database (which does all of the searching) can look through an index first when looking for records (performing a query), its job is made easier and your data is returned faster. Without an index (or key) to assist in finding information in the database, the database must search each table sequentially, record by record. It would be like searching for the author "Shakespeare" in the card catalog and starting with "A."

Structured Query Language (SQL) Once the data is stored in the database, retrieving it is made easier by using an English-like language called *Structured Query Language*, or *SQL*. SQL has evolved into the most widely accepted means to "converse" with a database. Basically, the user asks questions in the SQL language; this is called a "query." The database engine "answers" by returning any database rows that meet the requirements of the query. The query usually contains the names of the tables to search, the names of the columns to return, and other information that sets the scope of the search. For example, a SQL query on our Authors table might look like this:

```
"Select Name, Picture from Authors where Date_of_Birth = '2/7/1947' "
```

This SQL query would return the name and picture of all authors whose birthday is February 7, 1947. If any rows were returned, you could use bound controls to display the values.

Because many of the external databases that Visual Basic recognizes are not relational in design, one of Visual Basic's jobs is to convert external database structures to a relational model. Your code will not have to provide any specific logic to support this translation once the database has been opened—it is all done automatically.

Note The sample database shipped with this version of Visual Basic, BIBLIO.MDB, consists of a list of books that explain the relational database model. You can study the list for additional titles on designing, creating, and tuning relational databases, and using SQL to access them.The database itself is located in the \VB directory.

Working with Database Structure

In order to manipulate your database, you may want to map or expose its structure. By mapping a database, you can discover the names and structure of its tables, fields, and indexes. Once you know these details, writing applications to extract database information becomes much simpler.

Behind the scenes, Visual Basic allows applications to manipulate data created either by Visual Basic or by other database applications. For example, you can use Microsoft Access to create and manage a database that your Visual Basic program will be able to manipulate. There are two categories of databases that the data control recognizes:

- **Microsoft Access-format databases.** These database files can be created with Microsoft Access and manipulated with Microsoft Access and Visual Basic. This is Visual Basic's *native* format—it provides the most flexibility and speed.
- **External databases.** This category includes database formats such as dBASE, Paradox, FoxPro, and Btrieve. Indexed sequential access method (ISAM) files created with Visual Basic for MS-DOS are not supported directly, but these files can be exported to ASCII format and re-imported into Visual Basic using Visual Basic code.

When the data control is used with Visual Basic Professional Edition, your applications will be able to create new databases and modify database structures. In addition, you will be able to access ODBC databases including Microsoft SQL Server and Oracle®.

Defining a Database

Visual Basic uses a specific set of data access objects to define or *describe* the database, tables, fields and records you work with. Figure 20.3 illustrates the database design of Visual Basic databases. All databases, regardless of their internal structure, will be expressed in terms of this design.

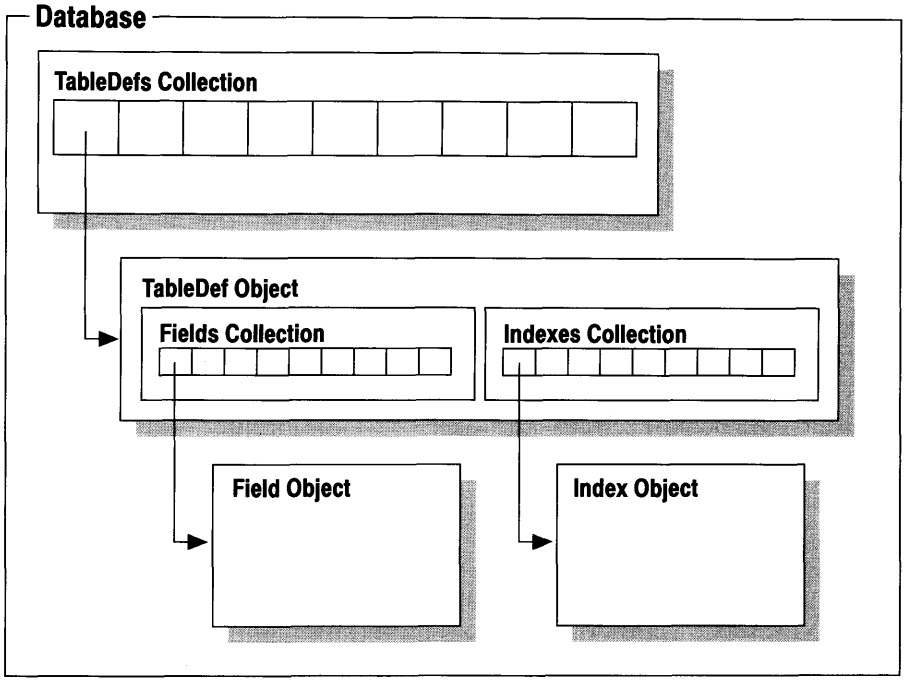


Figure 20.3 Visual Basic Database Structure

The Database Object

The data control's Database property can be represented by a **Database** object that *describes* the current database. It maintains information about the name of the database and its structure. For example, the database structure describes the tables in the database and the fields in each table. The **Database** object also supports several Visual Basic methods that you can use to manage the database.

Collections

A *collection* is a set of related objects. The objects in a collection are referred to as *members* of the collection. In Visual Basic, each member of a collection has a unique name so it's easy to identify. You can use the Count property to determine the number of members in a collection. Collections are managed like the items in a list box. Each member of the collection can also be referenced by its number, which starts with 0.

For example, to print the names of tables in your database you can examine the **TableDefs** collection:

```
Dim I As Integer
For I = 0 To Data1.Database.TableDefs.Count - 1
    Print Data1.Database.TableDefs(I).Name
Next I
```

Note In the Visual Basic Standard Edition, the data control can be used to open only existing database objects. The **OpenDatabase** function and the ability to **Dim** other database objects (like **TableDefs**, **Fields**, **Indexes**, **Dynaset,** and **Snapshot**) is not supported. Because of this, you cannot create new databases or modify existing database structures using Visual Basic procedures and data objects in the Standard Edition.

However, you *will* be able to create databases, tables, and indexes using the Visual Basic Data Manager application (supplied with Visual Basic) or other database administration applications designed to work with your specific database. In the Visual Basic Professional Edition, you will be able to create and modify database structures.

TableDef Objects and TableDefs Collection

Because a database can have several tables, and tables can have several fields, Visual Basic groups the definitions for these objects in *collections*.

A **Database** object can have one or many tables. The description of each table is maintained in a **TableDef** object (shown in Figure 20.4).

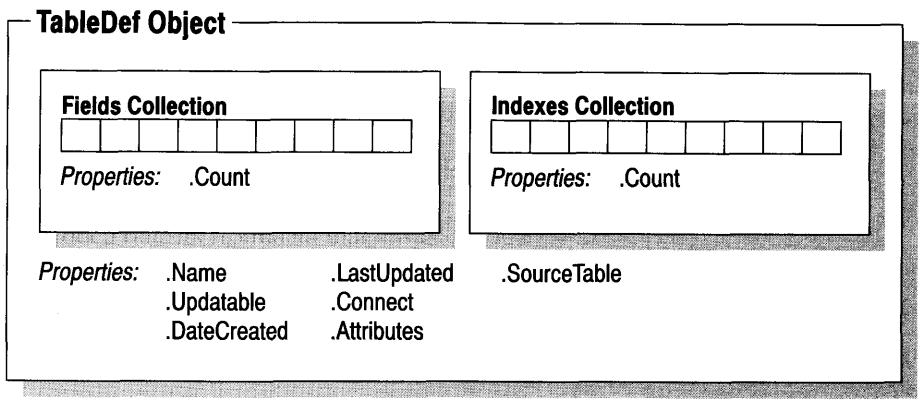


Figure 20.4 The TableDef Object

There is one **TableDef** for each table in the selected database. Each **TableDef** object contains the name of the table, a collection of **Fields**, and an **Indexes** collection. A **TableDef** does not contain data; rather, it describes it.

All **TableDef** objects are kept in the **Database** object's **TableDefs** collection. To find out how many tables are in a database, you can examine the Count property of the **TableDefs** collection, which contains the number of **TableDef** objects in the collection.

Field Objects and Fields Collection

Each **TableDef** object contains one or more **Field** objects. A **TableDef** object contains a **Fields** collection which contains one or more **Field** objects—one for each field (column) in the table. Data values are kept in **Field** objects (Figure 20.5) associated with Recordset objects. The Recordset property contains a **Dynaset** whose **Field** objects contain Value properties that contain data.

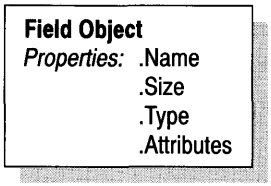


Figure 20.5 The Field Object

The set of all of the **Field** objects for a particular **TableDef** object is kept in the **Fields** collection. To find out how many fields are in a table, you can examine the Count property of the **Fields** collection, which contains the number of **Field** objects in the collection.

Index Objects and Indexes Collection

A table may have one or more indexes. **Index** objects (Figure 20.6) are used to expedite access to the table rows by maintaining internal pointers to data, referencing chosen fields as the key.

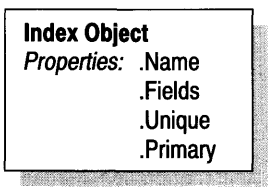


Figure 20.6 The Index Object

The set of all **Index** objects associated with a **TableDef** object is kept in the **Indexes** collection. To determine how many indexes have been established for a table, you can examine the Count property of the **Indexes** collection, which contains the number of **Index** objects in the collection.

Through use of a table's **Index** object, Visual Basic can retrieve rows directly; without having to search the table row by row. You can think of the data index like the index in a book. Instead of searching each page for a particular topic, you can use the index to quickly locate a page reference to a specified subject.

Usually, one of the table's indexes is defined as unique to ensure that no two rows are the same. When you define an index as "Primary," this implies that the index will be the principal test to determine if the record is a duplicate. Although unique indexes are not required, you will find it difficult to locate individual rows to update without them. Each index can specify several fields to make up the key.

All of the Visual Basic data objects are used to define the database and the data it holds. These database structures are initialized automatically when a database is opened, so your Visual Basic code will not need to execute any code to define the databases you want to access.

Mapping a Database

To *map* a database (show its structure or schema), you need to either use the Data Manager application or run a short Visual Basic procedure to map the database tables, fields, and indexes.

Note that all of the following examples assume that you have opened your database using the following code:

```
Dim I As Integer
On Error Resume Next        ' Without a RecordSource we get an error.
Data1.DatabaseName = "BIBLIO.MDB"
Data1.Refresh
```

Viewing the Database Properties

Once the database is open, can view the properties that indicate its status. The following table summarizes the properties of the **Database** object.

Property	Description
Name	The DatabaseName property.
Connect	The connect string used to open the database.
CollatingOrder	The sort method that is used in the database. Set by the *language* argument when the database is created.
Updatable	Indicates if the database is updatable.
Transactions	Indicates if the database supports transactions.

The following code displays the specified **Database** object's properties.

```
Print "Source of Data:", Data1.DatabaseName
Print "Recordset source table:", Data1.RecordSource
Print "Connect string parameters:", Data1.Connect
Print "Are transactions supported?:", Data1.Database.Transactions
Print "Is the database Updatable?:", Data1.Database.Updatable
Print "Is database opened in Exclusive mode?:", Data1.Exclusive
Print "Is database opened in Read-Only mode?:", Data1.ReadOnly
```

Mapping the Tables

The name of each database table is kept in the **TableDefs** collection along with other details. The **TableDefs** collection also contains system tables along with your data tables and queries. Each **TableDef** object can be referenced by number or by name. Remember that the Count property of the **TableDefs** collection contains the number of tables in the database.

Property	Description
Name	The table's name.
DateCreated	The date the table was created.
LastUpdated	The date the table's structure was last updated.
Attributes	The option bits for this table. Indicates the type of table and other options. For more information, search Help for *Attributes*.

This example shows how to list TableDef properties. (Notice the test for system tables.)

```
For i% = 0 To Data1.Database.TableDefs.Count - 1
If (Data1.Database.TableDefs(i).Attributes And DB_SYSTEMOBJECT) = 0 Then
    Print "Table Name:", Data1.Database.TableDefs(i).Name
    Print "Date Created:", Data1.Database.TableDefs(i).DateCreated
    Print "Last Updated:", Data1.Database.TableDefs(i).LastUpdated
    Print "Updatable?:", Data1.Database.TableDefs(i).Updatable
    Print "Attributes:", Data1.Database.TableDefs(i).Attributes
End If
Next i%
```

For More Information For information on any of the properties, search Help for the property name.

Mapping the Fields

The details of each field are kept in the **Fields** collection, which is a member of a **TableDef** object. The following example shows how this is done. Like the **TableDef**, each **Field** object can be referenced by number or by name. The Count property of the **Fields** collection contains the number of fields in the table. The **Field** properties you will be most interested in are listed in the following table.

Property	Description
Name	The field's name
Type	The data type for this field
Size	The maximum size for this field
Attributes	The option bits for this field

The following example shows how to list the Field properties for a named table. Notice that the Authors table is named to index the **TableDefs** collection, but you can also use the **TableDef** number, as in the previous example:

```
For i% = 0 To Data1.Database.TableDefs("Authors").Fields.Count - 1
    Print "Name:",Data1.Database.TableDefs("Authors").Fields(j%).Name
    Print "Type Code:",
    Print Data1.Database.TableDefs("Authors").Fields(i%).Type
    Print "Size:",
    Print Data1.Database.TableDefs("Authors").Fields(i%).Size
    Print "Attribute Bits:",
    Print Data1.Database.TableDefs("Authors").Fields(i%).Attributes
Next i%
```

Mapping the Indexes

The details of each table's index are kept in the **Indexes** collection, which is also a member of the **TableDef** object. Each **Index** object can be referenced by number or by name. The Count property of the **Indexes** collection contains the number of indexes defined on this table. The **Index** properties you will be most interested in are listed in the following table.

Property	Description
Name	The field's name.
Fields	A semicolon-delimited string of key field names that act as keys for the index.
Unique	Indicates whether duplicate records are permitted based on this index.
Primary	Indicates whether this is the (one and only) primary index for this table.

The following example shows how to print the **Index** properties for the Titles table:

```
Data1.Database.TableDefs("Titles").Fields.Refresh  ' Get the structure.
For i% = 0 To Data1.Database.TableDefs("Titles").Indexes.Count - 1
    Print "Name:",Data1.Database.TableDefs("Titles").Indexes(i%).Name
    Print "Fields:",
    Print Data1.Database.TableDefs("Titles").Indexes(i%).Fields
    Print "Unique:",
    Print Data1.Database.TableDefs("Titles").Indexes(i%).Unique
    Print "Primary:",
    Print Data1.Database.TableDefs("Titles").Indexes(i%).Primary
Next i%
```

Accessing the Recordset Object

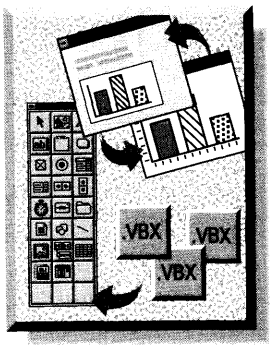
When you create a Recordset, the data is made available to your application in the **Field**.Value property. In the preceding examples, no **Recordset** objects were created; only the structure was exposed, so the Value property was not available. You can access the *value* of a **Field** object only when it is part of the current record of a **Recordset**. The following example shows eight different ways to access the value of the last name field in a **Recordset** created from the sample database:

```
Data1.DatabaseName = "BIBLIO.MDB"
Data1.RecordSource = "Titles"
Data1.Refresh       ' Open the database and create a Recordset object.

string1$ = Data1.Recordset(0)                    ' Direct-access methods
string2$ = Data1.Recordset("Title")

i% = 0                                           ' Indirect-access methods
string3$ = Data1.Recordset(i%)                   ' By number
f$ = "Title"
string4$ = Data1.Recordset(f$)                   ' By name

string5$ = Data1.Recordset!Title
string6$ = Data1.Recordset![Year Published]      ' For multi-word
                                                 '  field names
string7$ = Data1.Recordset.Fields("Title").Value
string8$ = Data1.Recordset.Fields(0).Value
```

All the string*n*$ variables in this example end up containing the same value for the Title field of the current record (except the reference to Year Published).

CHAPTER 21

Communicating with Other Applications



Because Visual Basic applications run in a multitasking operating environment along with other applications, you can easily exchange information by copying and pasting data. But your Visual Basic applications can automate this process, using dynamic data exchange (DDE) to extract data from other applications, automatically update them with new data, and even send commands or keystrokes to manipulate them by remote control.

Contents

- What Is Dynamic Data Exchange?
- Creating DDE Links at Design Time
- Link Properties
- Link Events
- Enabling Users to Create Their Own Automatic Links
- Using Methods to Perform DDE Operations
- Handling Errors
- Sending Keystrokes to Other Applications

DDE.MAK

Many of the code examples in this chapter are taken from the DDE.MAK sample application. If you installed the sample applications, you will find this application in the \DDE subdirectory of the main Visual Basic directory (\VB\SAMPLES\DDE).

What Is Dynamic Data Exchange?

Dynamic data exchange is a mechanism supported by the Microsoft Windows operating system that enables two applications to "talk" to each other by continuously and automatically exchanging data. DDE automates the manual cutting and pasting of data between applications, providing a faster vehicle for updating information.

Note Not all applications support DDE. Consult the documentation for your other applications to see if they support DDE.

Sources, Destinations, and Conversations

Two applications exchange information by engaging in a DDE *conversation*. This is similar to a conversation between two people. The application that initiates the conversation is called the *destination application*, or just the *destination*; the application responding to the destination is the *source application*, or just the *source*. An application can be engaged in several conversations at the same time, acting as the destination in some and the source in others. There is nothing special about an application that makes it a destination or source; these are simply roles that an application can adopt.

In Visual Basic, any text box, picture box, or label can be the destination in a conversation, while any form can be a source.

Note In the documentation for Microsoft Visual Basic 1.0, the destination was called the "client" and the source was called the "server." This terminology has been changed to conform to the new standard terminology for Windows–based applications.

Applications, Topics, and Items

When a destination application begins a DDE conversation, it must specify two things:

- The name of the source application it wants to talk to.
- The subject of the conversation, called the *topic*.

When a source application receives a request for a conversation concerning a topic it recognizes, it responds and a conversation is started. Once established, a conversation cannot change applications or topics. The combination of application and topic uniquely identifies the conversation, and it remains constant for the duration of the conversation. If either the destination or the source changes the application or the topic, the conversation is terminated.

During the conversation, the destination and source may exchange information concerning one or more items. *Items* are references to data that is meaningful to both applications. Either the destination or the source can change the item without affecting the state of the conversation.

Together, the application, topic, and item serve to uniquely identify the data that is being passed between the applications. Each of these is discussed in detail in the sections that follow.

Application Name

Every application that can be a DDE source has a unique *application name*. Usually this is the executable file name without any extension. The names of a few Microsoft applications are shown in the following table.

Application	DDE application name
Microsoft Excel	Excel
Microsoft Word for Windows	WinWord
Microsoft Project	Project
Microsoft Access ™	MSAccess
Microsoft FoxPro® for Windows	FoxPro
Microsoft Windows Program Manager	ProgMan

When a form in a Visual Basic application is the source in a conversation, its application name is the name you choose for the application when you make it into an executable file. If you are running your application within Visual Basic, the application name is the name of the project without any file extension.

Note For the application name of other Windows–based applications, see the documentation for those applications.

When performing DDE, the application name is not case-sensitive: "excel," "Excel," and "EXCEL" are all equivalent.

Topic

The *topic* defines the subject of a DDE conversation and is usually some unit of data that is meaningful to the source application. Most applications recognize a document name as a topic for a DDE conversation. For example, Microsoft Excel recognizes a file name (including the path if necessary) ending in .XLS or .XLC, while Word for Windows recognizes a file name ending in .DOC or .DOT. As with the application name, the topic for a DDE conversation is not case-sensitive.

When a form in a Visual Basic application is the source in a conversation, you can specify the topic by setting the LinkTopic property for that form. The LinkTopic property is described in the section "Link Properties" later in this chapter.

Many applications that perform DDE support a topic called "System." You can use this topic to request information from the application (for example, what other topics it supports and what data formats it supports). Check the documentation for the application to see if it supports this topic. You can provide a System topic in your own Visual Basic applications by including a form with its LinkTopic property set to System and controls with names corresponding to various system items. (Making a form a DDE source is discussed in the section "Source Events" later in this chapter.)

Item

The item identifies the piece of data *actually being passed* during the DDE conversation. For example, Excel recognizes cell references (such as R1C1) as items in a conversation. Word for Windows recognizes any bookmark in the document specified by the LinkTopic as an item in a conversation. Word also recognizes the special bookmark \Doc as a valid LinkItem for any document. As with application names and topics, the item for a conversation is not case-sensitive.

When a control in a Visual Basic application is the destination in a conversation, it defines the item for the conversation by setting the LinkItem property for that control. The LinkItem property is described in the section "Link Properties" later in this chapter.

When a form in a Visual Basic application is the source in a conversation, the name of each of the text boxes, labels, and pictures on the form can be the item for a DDE conversation.

DDE Links

A DDE conversation is often called a *link* because the two applications in the conversation are linked by the data they are exchanging. There are three kinds of links, distinguished by how the source updates the destination when data in the source changes:

- *Automatic link*: The source supplies data to the destination every time the data defined by the LinkItem changes.
- *Manual link*: The source supplies data only when the destination requests it.
- *Notify link*: The source notifies the destination when the data changes, but supplies data only when the destination requests it.

Note In the documentation for Microsoft Visual Basic 1.0, automatic links were called "hot links" and manual links were called "cold links." This terminology has changed to conform to the new standard terminology for Windows–based applications.

Creating DDE Links at Design Time

While you are designing a Visual Basic application, you can create data links between it and other applications that support DDE. These links are saved as values in the various Link properties of the forms and controls in your application, so that they are automatically re-established whenever the application is run. This is convenient, because you don't have to write any code to allow your applications to perform DDE conversations with other applications.

This convenience comes at the expense of some flexibility, however, because you cannot change these links without writing code to do so. From a practical standpoint, you will probably want to establish links at run time so that you can handle errors and control things such as the order in which the links are established. Nevertheless, you can learn a lot about DDE by creating links at design time, as described in this section.

Obtaining Data Through Links

If you want your Visual Basic application to use data supplied by other applications, then you want to create a destination link. A *destination link* is one in which your application (the destination) is the one requesting the information, and the other application is the source. You can create such a link at design time and make it a permanent part of your application. To do this, you must have the project containing your application open in the Visual Basic environment, and the other application must be running.

▶ **To create a destination link between your application and another application**

1. In the other application, select the data item you want to link.
2. From the Edit menu in that application, choose Copy.
3. In your Visual Basic application, select the control (text box, label, or picture box) that you want to receive the data.
4. From the Edit menu in Visual Basic, choose Paste Link.

If the link is successful, the value of the control (the Text property of a text box, the Caption property of a label, or the Picture property of a picture box) changes to display the current state of the data in the source application. If the Paste Link command on the Edit menu is unavailable, then the source application you attempted to paste the link from does not support DDE or cannot provide data in a format that Visual Basic recognizes.

Remember that this is an *automatic* link; once the link is established, the value of the control will change whenever the data in the source application changes. This will happen while your application is running (causing a Change event) and while you're building the application, so don't be surprised if you see the value of controls changing at design time.

This link is saved with the form and is "permanent." Whenever that form is opened at design time or loaded at run time, it attempts to re-establish the conversation with the source application. If the source application is running and recognizes the topic, then the conversation is re-established. If the source application is not running or does not respond to the topic, then Visual Basic generates an error.

Providing Data Through Links

If you want your Visual Basic application to supply data to other applications, then you want to create a source link. A *source link* is one in which the other application (the destination) requests data from your application (the source). Like destination links, you can create source links at design time and make them a permanent part of your application. To do this, the project containing your application must be open in the Visual Basic environment, and the other application must be running.

▶ **To create a source link between your application and another application**

1. In your Visual Basic application, select the form that will be the source for the data. Set its LinkMode property to 1-Source.
2. Select the control on that form that contains the data you want to supply.
3. From the Edit menu in Visual Basic, choose Copy.
4. In the destination application, select the destination for the data.
5. From the Edit menu in the destination application, choose Paste Link.

If the link is successful, the value of the control (the Text property of a text box, the Caption property of a label, or the Picture property of a picture box) is displayed in the destination application. If there is no Paste Link command in the destination application, then it does not support DDE automatic links. If the Paste Link command exists but is unavailable, then the destination does not recognize the data you attempted to supply from your Visual Basic application.

Many applications do not recognize picture data, for example, and some restrict the kind of information that can appear in certain parts of their documents. Consult the documentation for the destination application if this is the case.

Once you have established an automatic link between a control in your application and a destination application, Visual Basic automatically supplies new data to the destination application every time the value of the control changes. You can change the value of the control at design time and watch as Visual Basic supplies the new data to the other application.

When you run your application, however, Visual Basic must break the links as it switches from design time to run time. Some applications handle this and automatically attempt to re-establish their DDE conversations, but you may find that the source links you established at design time no longer work at run time. Most applications that support DDE provide some way to refresh their links manually; consult the documentation for that application.

Link Properties

Using the Copy and Paste Link commands to establish DDE conversations is fine for learning about DDE or for quick programming tasks, but these commands aren't flexible enough for most sophisticated applications. In addition, they don't allow you to set up conversations with other applications that support DDE but do not provide Copy and Paste Link commands.

To go further with DDE in Visual Basic—to create and use DDE conversations from code, or to manually establish conversations without using Paste Link—you must understand and use the Link properties of forms and controls.

Destination Link Properties

Destination links are maintained by properties on the control that is the destination of the conversation. To explore these properties, look at what the Visual Basic Paste Link command actually does. When you perform a Paste Link operation on a control to make it the destination of a DDE conversation, Visual Basic actually sets the values of three of the properties for that control: LinkTopic, LinkItem, and LinkMode.

The LinkTopic Property

The LinkTopic property of a control specifies its DDE conversation. When a control is engaged in a DDE conversation with another application, the LinkTopic property contains the name of that application and the topic of the conversation using this convention:

application|*topic*

The name of the source application is separated from the topic by a vertical bar or "pipe" character (character code 124). For example, a link with a Microsoft Excel spreadsheet might look like this:

```
EXCEL|C:\EXCEL\PAY.XLS
```

You can set this property for any text box, label, or picture control. If you know the application name for an application and a topic it supports, you can establish a DDE conversation with that application.

The LinkItem Property

The LinkItem property of a control specifies the item for the conversation defined by the LinkTopic property for that control. The setting of this property varies depending on the source application. For example, if the topic of a conversation with Microsoft Excel is a spreadsheet, the LinkItem could be:

R1C1

The LinkMode Property

If you try setting the LinkTopic and LinkItem properties, you'll find that nothing happens. This is because you have to "activate" the link by setting the LinkMode property to 1-Automatic, 2-Manual, or 3-Notify. Normally, the LinkMode property for a control is set to 0-None. When it is set to Automatic, Manual, or Notify, Visual Basic immediately attempts to initiate the conversation specified in the LinkTopic property. If the source application specified in the LinkTopic property is not running, Visual Basic generates an error.

If you set the LinkMode property to 1-Automatic, then whenever the data specified by the combination of the LinkTopic and the LinkItem changes, the control receives the new data and a Change event is generated for that control. If you set the LinkMode property to 2-Manual or 3-Notify, the control is not updated automatically and you must use the **LinkRequest** method (described later in this chapter) to obtain new data from the source. You can terminate the conversation at any time by setting the LinkMode property to 0-None.

If the LinkTopic or LinkItem property is not set to a valid value, setting LinkMode to Automatic, Manual, or Notify causes an error.

If you change the value of the LinkTopic property, Visual Basic terminates the old conversation. Visual Basic changes LinkMode to 0-None when you change the LinkTopic, but it's good practice to explicitly set LinkMode to None before changing the LinkTopic property. Once you have set the new value for LinkTopic, you can establish the new conversation by setting LinkMode to 1-Automatic, 2-Manual, or 3-Notify again.

Note If the source application terminates a conversation, the LinkMode property is set to None and a LinkClose event (described later in "Link Events") occurs. This is a change from Visual Basic 1.0—the LinkMode property was not set to None when the source terminated a conversation.

In more recent versions of Visual Basic, once the conversation is terminated, performing any DDE operation on this link causes a run-time error.

Creating an Automatic Link

If the LinkTopic property for a control is set to a valid value (a combination of the name of a running application and a topic that application recognizes) and the LinkItem property is set to a valid value for that LinkTopic, then you can establish an automatic link by setting the LinkMode property for that control to Automatic.

The following code for a Form_Click event attempts to establish an automatic link between a text box on the form and Microsoft Excel when the form is clicked the first time. Each time the form is clicked after that, the item for the conversation is changed (notice that this does not terminate the conversation):

```
Sub Form_Click ()
    Const AUTOMATIC = 1, NONE = 0
    Static Row As Integer

    Row = Row + 1
    If Row = 1 Then
        Text1.LinkMode = NONE
        Text1.LinkTopic = "EXCEL|C:\EXCEL\AMORTIZE.XLS"
        Text1.LinkItem = "R1C1"
        Text1.LinkMode = AUTOMATIC
    Else
        Text1.LinkItem = "R" & Row & "C1"
    End If
End Sub
```

Note The constants AUTOMATIC, NONE, MANUAL, NOTIFY, and SOURCE are contained in the CONSTANT.TXT file that ships with Visual Basic. You can use these constants in your code if you load this file into one of the modules in your application.

Creating a Manual Link

You create a manual link by setting the LinkMode property to 2-Manual. When LinkMode is Manual, a conversation exists, but the control does not automatically receive new data whenever it changes. To obtain the most up-to-date data, you must use the **LinkRequest** method on the control.

The following code for a Form_Click event attempts to establish a manual link between a text box on the form and Microsoft Excel when the form is clicked the first time. Each time the form is clicked after that, new data is requested:

```
Sub Form_Click ()
    Const MANUAL = 2, NONE = 0

    If Text1.LinkMode = NONE Then
        Text1.LinkTopic = "EXCEL|C:\EXCEL\AMORTIZE.XLS"
        Text1.LinkItem = "R1C1"
        Text1.LinkMode = MANUAL
    End If
    Text1.LinkRequest
End Sub
```

Creating a Notify Link

The other possible value for LinkMode, 3-Notify, is similar to 2-Manual except that the source notifies your application when the data changes. This notification occurs in your application as a LinkNotify event. When this occurs, you can use the **LinkRequest** method to get the most up-to-date data.

Because bitmap data is often large and slow to transfer via DDE, you may not want to use an automatic link for a picture box. Instead, you could add a command button that users can choose when they want to see the latest data in a pie chart (PIE.XLC). Your code can make this button enabled when there is new data and disabled if there is no new data.

```
Sub Form_Click ()
    Const NOTIFY = 3, NONE = 0

    If picChart.LinkMode = NONE Then
        picChart.LinkTopic = "EXCEL|C:\EXCEL\PIE.XLC"
        picChart.LinkItem = "Chart"
        picChart.LinkMode = NOTIFY
        cmdUpdate.Enabled = True
    End If
End Sub

Sub cmdUpdate_Click ()
    picChart.LinkRequest
    cmdUpdate.Enabled = False
End Sub

Sub picChart_LinkNotify ()
    cmdUpdate.Enabled = True
End Sub
```

The LinkTimeout Property

Some applications take longer to respond in DDE conversations than others. If an application takes too long to respond, Visual Basic generates a trappable run-time error (error number 286). You can adjust how long Visual Basic waits for a response by setting the LinkTimeout property for a control. The values for the LinkTimeout property are tenths of seconds; the default value is 50 tenths, or five seconds. This should be adequate for conversations with most applications, but you can adjust it as necessary. This property is measured in absolute time, so you may need to increase this value on slower computers.

If you set the LinkTimeout property to –1, Visual Basic waits for 65,535 tenths of a second, or approximately one hour and 49 minutes, for a response from the other application before generating an error.

Note If Visual Basic seems to be "hung up" during a DDE conversation, and you don't want to wait for the time-out error to occur, you can recover by pressing the ESC key. Pressing ESC interrupts any pending DDE operations in Visual Basic and generates a trappable run-time error (error number 287).

Source Link Properties

Source links are more complex than destination links because they involve form properties as well as control properties.

When you copy a control from a form in Visual Basic and use Paste Link in another application to establish a conversation with the Visual Basic (source) application, the other (destination) application initiates the conversation using the values of two properties on the form: LinkTopic and LinkMode.

The LinkTopic Property

The LinkTopic property of a form specifies the topic of a DDE conversation to which the form will respond. The names of individual controls on the form define the items that other applications can use in DDE conversations. The application name is either the application name you specify when you create an executable file (in the Make EXE File dialog box) without the .EXE extension, or the project name (without the .MAK extension) if you are running the application in the Visual Basic development environment. This application name is the same as the EXEName property of the Application object.

For More Information For information on the EXEName property of the Application object, see the *Language Reference,* or search Help for *applications.*

For example, if you have a Visual Basic application called Mortgage that contains a form with a LinkTopic property set to Home, then the application will respond to destinations that attempt to establish this DDE conversation:

Mortgage|Home

If there are text box controls on the form called Interest, Principal, and Balance, then these are all valid items for this conversation (as are the names of any other text box, label, or picture box controls on the form).

Note By default, the LinkTopic property for a form has the value "Form*n*"where *n* is a number. This is the same as the default value for the Name and Caption properties for the form. All of these properties are independent, however; the LinkTopic property does not change its value when you change the Name or Caption properties.

If you change the LinkTopic property, Visual Basic terminates all conversations on that topic. The destination applications can then re-establish links with the new topic.

The LinkMode Property

Like the LinkMode property for controls, the LinkMode property for forms is normally set to 0-None. When it is None, no destination application can initiate a conversation with that form as the topic. If LinkMode is set to 1-Source, the controls on the form supply data to any destination that establishes a DDE conversation with them.

If you leave a form's LinkMode property set to 0-None at design time, then you cannot set it to Source at run time. If you set it to Source at design time, however, you can freely change its value between None and Source at run time. This is a change from Visual Basic 1.0, which allowed you to change the LinkMode property at run time regardless of what it was set to at design time.

With this change, Visual Basic can better optimize the size of your applications: Because any control on a form that is acting as a DDE source can be the item for a DDE conversation involving that form, Visual Basic must maintain the names of all the controls on each form that will act as a DDE source. If you leave the LinkMode property of a form set to 0-None at design time, Visual Basic assumes the form will never act as a DDE source and does not include the names of the controls on that form in the finished application, thus reducing the size of the .EXE file.

Tip To minimize the size of your finished applications, limit the number of forms with LinkMode set to Source, and limit the number of controls on each of these forms.

Some destination applications attempt to send data to the source application, even though the flow of data in DDE conversations is generally from source to destination. This is called *poking* data via DDE. If the LinkMode property for a form is set to Source, it enables destination applications to poke data to the controls on the form; if the LinkMode property is set to None, the form does not allow any poking of data to any of the controls on the form. If a destination application pokes data to a control on a form, a Change event occurs for that control.

The following code allows you to click the form to set whether it responds to DDE conversations with the topic "Home":

```
Sub Form_Click ()
    ' Assumes LinkMode was set to Source at design time.
    Const SOURCE = 1, NONE = 0

    If LinkMode = NONE Then
        LinkTopic = "Home"
        LinkMode = SOURCE
    Else
        LinkMode = NONE
    End If
End Sub
```

For More Information For information on the Link properties, see the *Language Reference*, or search Help for *link*.

Link Events

As you might expect, there are events associated with links, and you can write event procedures to respond to these events. The controls that can be destinations in a DDE conversation recognize these DDE-specific events:

- LinkOpen
- LinkClose
- LinkError
- LinkNotify

When forms are acting as DDE sources, they recognize these events:

- LinkOpen
- LinkClose
- LinkError
- LinkExecute

Destination Events

There are four events that can occur only when a control is engaged as the destination in a DDE conversation: LinkOpen, LinkClose, LinkError, and LinkNotify.

The LinkOpen Event

The LinkOpen event occurs when a control successfully initiates a DDE conversation. You can write this procedure to respond to the event:

```
Sub Text1_LinkOpen (Cancel As Integer)
    Dim Msg
    Msg = "DDE conversation initiated with LinkTopic " & Text1.LinkTopic
    Msg = Msg & " and LinkItem " & Text1.LinkItem
    MsgBox Msg
End Sub
```

This is useful for debugging purposes, as in the preceding example, or whenever you want to perform some operation (such as reading a file) every time the control successfully initiates a DDE conversation.

The LinkClose Event

If a control is engaged in a DDE conversation and the conversation is terminated for any reason—you set the control's LinkMode property to None or changed the control's LinkTopic property, or the source terminated the conversation—the LinkClose event occurs. You can write an event-handling procedure that responds to this event. For example:

```
Sub Text1_LinkClose ()
    MsgBox "DDE conversation terminated."
End Sub
```

The LinkError Event

If an error occurs when you are using code to manipulate a DDE conversation, it will be a run-time error that you can trap and handle in the usual way. For more information, see Chapter 10, "Handling Run-Time Errors."

When a DDE conversation is active, however, errors can occur when no code is executing: Data may arrive in a format that the destination doesn't understand, or a large amount of data may cause the operating environment to run out of memory. If DDE causes an error to occur when no code is running, Visual Basic generates a LinkError event. See the section "Handling Errors" later in this chapter, for more information.

The LinkNotify Event

If you set the LinkMode property of a control to 3-Notify, then the LinkNotify event occurs whenever the source has new data to supply to the control. As you saw earlier, you can use this event to notify the user that new data is available by enabling a button or displaying a message.

Another way to use this event is to set a flag in your code and obtain the data at a later time. For example, if the form is minimized or hidden, there is no need to get the data immediately because the user can't see it anyway: You can set a module-level variable and then request the new data in the Activate, Resize, or Paint events when the form becomes visible.

Source Events

There are four form events that can occur only when a form is engaged as the source in a DDE conversation: LinkOpen, LinkClose, LinkError, and LinkExecute.

The LinkOpen Event

The LinkOpen event occurs when a destination application initiates a conversation. The event procedure for this event accepts an integer argument, *Cancel*. The argument is always **False** when the event procedure is called.

If your event procedure code sets this argument to **True**, however, Visual Basic won't enable the destination application to establish the conversation. For example, the following code keeps track of how many destinations attempt to establish DDE conversations; once the number has exceeded 10, it denies all further attempts:

```
Sub Form_LinkOpen (Cancel As Integer)
    Dim DestinationCount As Integer          ' Module-level variable.
    If DestinationCount < 10 Then
        DestinationCount = DestinationCount + 1
    Else
        Cancel = True
    End If
End Sub
```

Code like this might be useful if you are creating large data structures or opening files for each DDE conversation and therefore need to limit the number of simultaneous conversations. To make this code work, you would also decrement the DestinationCount variable each time a conversation is terminated. You can detect the termination of a conversation with the LinkClose event.

The LinkClose Event

The LinkClose event occurs if a destination application terminates its conversation:

```
Sub Form_LinkClose ()
    If DestinationCount > 0 Then DestinationCount = DestinationCount - 1
End Sub
```

The LinkError Event

As with destination links, when code that establishes, terminates, or manipulates a DDE conversation is executing and an error occurs, the standard run-time error handler is invoked, if there is one. However, DDE can cause errors to occur when no code is executing. If this happens when a form is acting as the source in a DDE conversation, Visual Basic generates the LinkError event for the form. See the section "Handling Errors" later in this chapter, for more information.

The LinkExecute Event

Some applications send a command string to be executed by a source application; this causes a LinkExecute event for a Visual Basic form that is acting as a source. The event procedure for this event accepts the two arguments shown in the following table.

Argument	Description
CmdStr	A string containing the command to be executed.
Cancel	An integer argument. When a destination sends a command to a source to be executed, it expects to receive either a positive or a negative acknowledgement (that is, whether the source agrees to execute the command). If *Cancel* is zero when the event procedure exits, the destination receives a positive acknowledgement; if *Cancel* is nonzero, the destination receives a negative acknowledgement.

The contents of the command string vary from application to application; there is no predefined format or meaning for this argument. For example, this event procedure accepts a command to quit:

```
Sub Form_LinkExecute (CmdStr As String, Cancel As Integer)
    If CmdStr = "[Quit]" Then
        Cancel = False
        End
    Else
        Cancel = True
    End If
End Sub
```

If a form has no event procedure for the LinkExecute event, then when a destination attempts to send a command to the form, it always receives a negative acknowledgement.

For More Information For information on link events, see the *Language Reference*, or search Help for *link*.

Allowing Users to Create Their Own Automatic Links

Many applications that support DDE provide a way for the user to establish DDE conversations without writing code. As you have seen, Visual Basic does this with the Copy and Paste Link commands on the Edit menu. This is the standard method used by most applications that support automatic links. By writing a little code, you can supply this feature to the users of your Visual Basic applications.

Note While you are exploring this, you'll find the Clipboard application (CLIPBRD.EXE) invaluable. Supplied with your operating environment, it allows you to observe the actual data that your code is placing on the Clipboard.

The examples in this section assume that your application includes a menu control array called MenuBar that includes an Edit menu with the usual commands: Cut, Copy, Paste, and Paste Link.

Adding a Paste Link Command to Your Applications

In applications that support automatic links, the Paste Link command allows the user to create destination links that accept data from other applications. A Paste Link command in your application must obtain link information from the Clipboard and use that to set the Link properties of the active control. All applications that support DDE links provide link information using this convention:

application|*topic*!*item*

The *Application* and *Topic* are separated by a vertical bar or "pipe" character (character code 124), and the *Topic* and *Item* are separated by an exclamation point (character code 33). Notice that this is just an extension of the convention used in the LinkTopic property for controls (described earlier in this chapter).

▶ **To perform a Paste Link operation**

1. Obtain the link information from the Clipboard.
2. Split the information into a string for the application and topic, and a string for the item.
3. Use these strings to set the LinkTopic and LinkItem properties for the control that is accepting the Paste Link. You can determine the active control by using the ActiveControl property of the Screen object.

For More Information For information on the Screen object, see the *Language Reference*; or search Help for *Screen*.

Implementing a Paste Link Command

Example This event procedure code is assumed to be attached to a menu control array that provides the commands on the Edit menu, including Paste Link:

```
Sub mnuEdit_Click (Index As Integer)
    Dim item As Integer, Link
    Const AUTOMATIC = 1, CF_LINK = &HBF00

    Select Case Index
    Case CUT
        ' Code to support Edit Cut (omitted).
    Case COPY
        ' Code to support Edit Copy (omitted).
    Case PASTE
        ' Code to support Edit Paste (omitted).
    Case PASTELINK
        Link = Clipboard.GetText(CF_LINK)
        item = InStr(Link, "!")
        If item <> 0 Then
            Screen.ActiveControl.LinkMode = NONE
            Screen.ActiveControl.LinkTopic = Left(Link, item - 1)
            Screen.ActiveControl.LinkItem = Mid(Link, item + 1)
            Screen.ActiveControl.LinkMode = AUTOMATIC
        ElseIf InStr(Link, "|") Then
            ' Some DDE links have no item.
            Screen.ActiveControl.LinkMode = NONE
            Screen.ActiveControl.LinkTopic = Link
            Screen.ActiveControl.LinkItem = ""
            Screen.ActiveControl.LinkMode = AUTOMATIC
        End If
    End Select
End Sub
```

The code assumes that the constant PASTELINK is defined in the Declarations section of the form code and is set equal to the index of the Paste Link menu item.

Checking for Valid Link Data

Note that the preceding code fails if there isn't valid link data on the Clipboard, or if the active control is not one of the controls that supports DDE. You can modify the code to check this, but there is a better way. If there isn't valid link data on the Clipboard, or if the active control does not support DDE, then the Paste Link command should be made unavailable so the user can't choose it.

To take care of this, use the following code in the Click event procedure for the menu bar:

```
Sub MenuBar_Click (Index As Integer)
    Const CF_LINK = &HBF00, CF_DIB = 8, CF_BITMAP = 2, CF_TEXT = 1

    Select Case Index
    Case EDITMENU
        Edit(PASTELINK).Enabled = False
        If Clipboard.GetFormat(CF_LINK) Then
            If TypeOf Screen.ActiveControl Is PictureBox Then
                If Clipboard.GetFormat(CF_BITMAP) Then
                    Edit(PASTELINK).Enabled = True
                End If
            ElseIf TypeOf Screen.ActiveControl Is TextBox Then
                If Clipboard.GetFormat(CF_TEXT) Then
                    Edit(PASTELINK).Enabled = True
                End If
            End If
        End If
    End Select
End Sub
```

This code makes the Paste Link command unavailable if there isn't valid link data on the Clipboard, or if the active control is not a text box or picture box. You do not have to consider labels because the user can't select them; thus, a label will never be the active control.

In addition, this code ensures that the Paste Link command is enabled only if the data on the Clipboard is available in a format appropriate for pasting into the active control: CF_TEXT for a text box and CF_BITMAP for a picture box. You also may want to test for CF_DIB, CF_PALETTE, and CF_METAFILE for a picture box.

Creating Links with the Copy Command

In applications that support automatic links, the Copy command allows the user to create source links that supply data to other applications.

If you want to allow users of your application to supply automatic-linked data to other applications, modify the Copy command in your application to place link information on the Clipboard. In Chapter 17, "Interacting with the Environment," you saw code that copies the data from the active control onto the Clipboard. You need to add statements to this code that put link information on the Clipboard if the active control is a text box or a picture box (the controls that support links). Remember, although labels also support links, you don't need to add code for them because there's no way for the user to select a label.

The link information you place on the Clipboard must use the same convention you saw earlier:

application|*topic*!*item*

For the *application*, use the project name (if you are running your application in Visual Basic) or the application title you provide in the Make EXE File dialog box (if you are running your application as an executable file). *Topic* is the LinkTopic property for the form, and *item* is the name of the control that provides the data in the link. Because the Name property is not available at run time, you must store the name of each control in its Tag property.

Note that the following code assumes that the current name of the .EXE file is the same as the file name that was originally entered in the Make EXE File dialog box. This code will not work if the user later changes the name of the .EXE file (DDE requires the original file name given to the application in the Make EXE File dialog box, but App.EXEName always returns the current file name.

```
Sub Edit_Click (Index As Integer)
    Dim AppTopName

    Select Case Index
    Case CUT
        ' Code to support Edit Cut (omitted).
    Case COPY
        Clipboard.Clear
        If TypeOf Screen.ActiveControl Is PictureBox Then
            AppTopName = App.EXEName & "|" & LinkTopic
            AppTopName = AppTopName & "!" & Screen.ActiveControl.Tag
            Clipboard.SetText AppTopName$, CF_LINK
            Clipboard.SetData Screen.ActiveControl.Picture
        ElseIf TypeOf Screen.ActiveControl Is TextBox Then
            AppTopName = App.EXEName & "|" & LinkTopic
            AppTopName = AppTopName & "!" & Screen.ActiveControl.Tag
            Clipboard.SetText AppTopName$, CF_LINK
            Clipboard.SetText Screen.ActiveControl.Text
        Else
            ' Copy appropriate data from other types of controls.
            Clipboard.SetText Screen.ActiveControl
        End If
    Case PASTE
        ' Code to support Edit Paste (omitted).
    Case PASTELINK
        ' Code to support Edit Paste Link (omitted).
    End Select
End Sub
```

It is good practice to place the actual value of the control (the Text property of a text box or the Picture property of a picture box) on the Clipboard along with the link information. You can place multiple pieces of data on the Clipboard as long as you use a different format for each piece.

The preceding code assumes the LinkMode property for this form is set to Source at design time. It also assumes that the name of each text box and picture box control is stored in its Tag property.

Note If the control that provides data in the link is part of a control array, then *item* should include the array index enclosed in parentheses.

Using Methods to Perform DDE Operations

DDE gives you the ability to manipulate other applications by remote control. Once you have established a destination link with another application, you can send data, request data, and send it commands. When your application is the source in a conversation, you can also notify the destination when data has changed.

You can use these functions and methods to perform DDE operations:

- The **Shell** function
- The **LinkPoke** method
- The **LinkRequest** method
- The **LinkExecute** method
- The **LinkSend** method

Starting Other Applications

Often you'll want to establish a conversation with an application that is not currently running. If you attempt to establish a DDE conversation with an application that is not currently running, Visual Basic generates a run-time error (error number 282). You can trap this like any other run-time error, and start the application explicitly with the **Shell** function:

```
Sub StartApp (Link As Control, AppName As String, Topic As String)
    Dim t
    Const DDE_NO_APP = 282

    On Error GoTo StartUp
    Link.LinkMode = NONE
    Link.LinkTopic = AppName & "|" & Topic
    Link.LinkMode = MANUAL
    Exit Sub
```

```
StartUp:
    If Err = DDE_NO_APP Then
        t = Shell(AppName)
        Resume
    Else
        MsgBox "Unknown error."
        Stop
    End If
End Sub
```

Some applications provide a topic called System that you can use to request information about that application, such as what other topics the application supports. Microsoft Excel and Microsoft Word for Windows both support a System topic; for other applications, consult the documentation for that application. If you are creating a sophisticated Visual Basic application, you may want to provide a System topic by including a form with its LinkTopic property set to System and controls with names corresponding to various system items.

For More Information For information on the **Shell** function, see the *Language Reference*, or search Help for *Shell*.

Poking Data to Other Applications

Although the flow of data in a DDE conversation is usually from the source application to the destination application, the destination can also send data to the source. As mentioned earlier in this chapter, this is called poking data, and it is done with the **LinkPoke** method. **LinkPoke** sends the value of the control to the source application, updating the data referred to by the LinkItem property. For example, this procedure establishes a destination link with a cell on a Microsoft Excel worksheet and changes it:

```
Sub UpdateIt (Link As TextBox, filename As String, newVal As String)
    Const NONE = 0, MANUAL = 2

    Link.LinkMode = NONE
    Link.LinkTopic = "EXCEL|" & filename
    Link.LinkItem = "R1C1"
    Link.LinkMode = MANUAL
    Link.Text = newVal
    Link.LinkPoke
    Link.LinkMode = NONE
End Sub
```

When used with a picture box control, the **LinkPoke** method sends the contents of the Picture property (which could be a bitmap, a metafile, or an icon, depending on what was loaded into the picture) rather than the Image property (which is always a bitmap).

Requesting Data from Other Applications

When a control has initiated an automatic link, the source application automatically updates it whenever the data defined by the link changes. When a control has initiated a notify or manual link, however, the source does not automatically update it. You must explicitly ask the source to update the control by executing the **LinkRequest** method on the control:

```
Sub Picture1_Click ()
    Const NONE = 0, MANUAL = 2

    If Picture1.LinkMode = NONE Then
        Picture1.LinkTopic = "EXCEL|C:\EXCEL\CHART1.XLC"
        Picture1.LinkItem = "Chart"
        Picture1.LinkMode = MANUAL
        Picture1.LinkRequest
    Else
        Picture1.LinkRequest
    End If
End Sub
```

Notice that when you first establish a manual link, no data is immediately transferred. You must perform an explicit **LinkRequest** method to get any data in a manual link.

Sending Commands to Other Applications

You can also send commands to any application that supports DDE. Once you have established a destination link with another application, you can use the **LinkExecute** method on the control that is maintaining the link.

For example, you can send macro commands to Microsoft Excel. This code uses the **LinkExecute** method on a control named Link to command Microsoft Excel to close the active worksheet:

```
Link.LinkExecute "[FILE.CLOSE()]"
```

Every application accepts different command strings, so you should consult the documentation for the application to see what commands it will accept. Microsoft Excel, for instance, allows you to send a series of macro commands with a single **LinkExecute** statement by enclosing each command in square brackets. For example, this code tells Excel to save the current file and then quit:

```
Link.LinkExecute "[SAVE()][QUIT()]"
```

Notifying Other Applications When Data Has Changed

When a form in your application is acting as the source in a DDE conversation, Visual Basic handles most of the details for you. If a destination application initiates a conversation with a text box control on the form, for example, Visual Basic automatically updates the destination (in the case of an automatic link) or notifies the destination (in the case of a notify link) whenever the Text property of the text box changes.

However, Visual Basic does not do this with a picture box. Because the contents of a picture box can be so large, and because it often takes many graphics methods to complete a change to a picture, Visual Basic might waste time and resources updating all DDE destinations every time a pixel in the picture changes. Instead, with Visual Basic you explicitly update any DDE destinations by executing the **LinkSend** method on a picture box. This code draws random lines on a picture box and sends the contents of the Picture property to its DDE destinations (if any) when it's done:

```
Sub Picture1_Paint ()
    Const MaxPoints = 100
    Static Points(MaxPoints, 1) As Single, Count As Integer, i

    If Count <= MaxPoints Then
        Points(Count, 0) = Int(Rnd * Picture1.Width)
        Points(Count, 1) = Int(Rnd * Picture1.Height)
        Count = Count + 1
    End If
    Picture1.PSet(Points(0, 0), Points(0, 1))
    For i = 1 To Count - 1
        Picture1.ForeColor = QBColor(i Mod 16)
        Picture1.Line -(Points(i, 0), Points(i, 1))
    Next i
    Picture1.LinkSend
End Sub
```

DDE in Other Microsoft Applications

Microsoft Excel and Microsoft Word for Windows both support DDE and provide statements in their macro languages that allow users to write macros that perform DDE. If you have written macros in one of these Microsoft applications, you may find Table 20.1 helpful. It compares the DDE statements and functions in the Microsoft Excel and Microsoft Word macro languages with the equivalent DDE properties and methods in Visual Basic.

Table 20.1 DDE Statements in Microsoft Excel, Word, and Visual Basic

Microsoft Excel	Microsoft Word	Microsoft Visual Basic
INITIATE	DDEInitiate	LinkMode = Automatic LinkMode = Manual LinkMode = Notify
REQUEST	DDERequest	**LinkRequest** method
POKE	DDEPoke	**LinkPoke** method
EXECUTE	DDEExecute	**LinkExecute** method
TERMINATE	DDETerminate	LinkMode = None

Handling Errors

As with other activities, errors can sometimes occur when a Visual Basic application is performing DDE. The kinds of errors that occur can be divided into two groups:

- Errors that occur in statements, functions, or expressions within executing code in the same way that run-time errors occur in any executing code.
- Errors that occur when no code is being executed.

Errors of the first kind are no different than any other run-time errors and can be handled with standard error-handling statements and functions provided in Visual Basic.

For More Information For information on handling these kinds of errors, see Chapter 10, "Handling Run-Time Errors." For a list of all trappable run-time errors, including DDE errors, see the *Language Reference*, or search Help for *errors*.

The second group of errors may seem somewhat unusual. How can an error occur if no code is executing? To understand this, remember that Visual Basic is always performing a variety of operations, even when none of your Visual Basic code is executing. One thing Visual Basic does is update automatic links whenever data changes. Many errors can occur at this point; for example, the data may be in the wrong format, or it may be so large that copying it causes Visual Basic or the operating environment to run out of memory. Visual Basic deals with this by generating a LinkError event. You can write an event procedure to handle this event like any other event, as explained in the next sections.

Errors When a Visual Basic Application Is the Destination

If a Visual Basic application is engaged as a destination in a DDE conversation, and an error occurs when no Visual Basic code is executing, the LinkError event occurs. In Visual Basic applications, the destination in a DDE conversation is always a control, so in this case the LinkError event is a control event. As you might expect, the LinkError event occurs at the control involved in the conversation that caused the error.

To handle the LinkError event, you write an event procedure. The event procedure takes one integer argument that indicates the error that occurred. These DDE error numbers are completely different from the run-time error numbers, but you use them in the same way to distinguish between various errors.

For example, suppose you have a text box called CurrentPrice that is engaged as the destination of a DDE conversation. To handle errors that might occur during the DDE conversation, write an event procedure like this:

```
Sub CurrentPrice_LinkError (ErrNum as Integer)
    Dim Msg
    Const OUT_OF_MEMORY = 11, WRONG_FORMAT = 1, TOO_MANY_DESTINATIONS = 7
    Const UPDATE_FAILED = 8

    Select Case ErrNum
    Case OUT_OF_MEMORY
        Msg = "Not enough memory to perform DDE."
    Case UPDATE_FAILED
        Msg = "Could not update data via DDE."
    Case TOO_MANY_DESTINATIONS
        Msg = "DDE source cannot handle this many destinations."
    Case Else
        Msg = "Unexpected DDE Error: " & ErrNum
    End Select
    If ErrNum <> WRONG_FORMAT Then
        ' Ignore Wrong Format errors because several may be generated
        ' until the source and destination agree on a common format.
        MsgBox Msg, 48, App.EXEName & " DDE Failure"
    End If
End Sub
```

Of course, if the control is an element in a control array, there will be a second argument for the array index.

For More Information For a list of link errors, see the LinkError Event topic in the *Language Reference*, or search Help for *LinkError*.

Errors When a Visual Basic Application Is the Source

If a Visual Basic application is engaged as a source in a DDE conversation, and an error occurs when no Visual Basic code is executing, the LinkError event occurs. In Visual Basic applications, the source in a DDE conversation is always a form, so in this case the LinkError event is a form event.

The event procedure for a source error is exactly the same as the event procedure for a destination error, except that it is a form event procedure:

```
Sub Form_LinkError (ErrNum as Integer)
    Dim Msg
    Const OUT_OF_MEMORY = 11, WRONG_FORMAT = 1, LINKMODE_NONE = 6
    Const UPDATE_FAILED = 8

    Select Case ErrNum
    Case OUT_OF_MEMORY
        Msg = "Not enough memory to perform DDE."
    Case UPDATE_FAILED
        Msg = "Destination tried (unsuccessfully) to poke data."
    Case LINKMODE_NONE
        ' Source form LinkMode property set to NONE.
        Msg = "Destination trying (unsuccessfully) to connect."
    Case Else
        Msg = "Unexpected DDE Error: " & ErrNum
    End Select
    If ErrNum <> WRONG_FORMAT Then
        ' Ignore Wrong Format errors because several may be generated
        ' until the source and destination agree on a common format.
        MsgBox Msg, 48, App.EXEName & " DDE Failure"
    End If
End Sub
```

Catching and Avoiding Errors

Although you tend to think of your Visual Basic application as completely separate from all other running applications, remember that all applications are running on the same computer. In order to run simultaneously, they must share the computer's resources. When your Visual Basic application is waiting for an event to occur, it automatically shares the computer's resources with other running applications. Visual Basic also "yields" (allows other applications to run) whenever it performs a DDE operation.

Most applications can respond at this time, but some applications may be unable to complete their side of the DDE conversation. This causes a run-time (time-out) error in your code. To avoid this, you can call the **DoEvents** statement. For example, you might write code like this:

```
Dim i, retryCount
Const ERR_TIMEOUT = 286

On Error GoTo ErrHandler
    ' ExecuteStrings() is an array defined and filled elsewhere.
    For i = 0 To UBound(ExecuteStrings)
        Link.LinkExecute ExecuteStrings(i)
    Next i
Exit Sub

ErrHandler:
    If Err = ERR_TIMEOUT And RetryCount < 10 Then
        DoEvents
        retryCount = retryCount + 1
        Resume
    Else
        Error Err
    End If
```

Calling the **DoEvents** statement causes Visual Basic to yield briefly and allow other applications to execute. In the preceding example, calling **DoEvents** when a time-out error occurs enables the other application to handle its side of the DDE conversation. Some applications have to do more work to handle DDE than others; you may find that one call to **DoEvents** is not enough and that you must call **DoEvents** several times. If you get a lot of run-time errors indicating that the other applications are busy or not handling DDE properly, try placing a call to **DoEvents** in your error-handling code.

For More Information For information on **DoEvents**, see Chapter 17, "Interacting with the Environment"; the *Language Reference*; or search Help for *DoEvents*.

Sending Keystrokes to Other Applications

Some applications do not support DDE. Nevertheless, because all applications run in the same operating environment, some communication is always possible. The easiest way to communicate with applications that do not support DDE is by sending keystrokes.

Note You can send keystrokes to any Windows–based application running in the operating environment, whether or not it supports DDE. However, you cannot send keystrokes to an application running in an MS-DOS session running in Windows.

When you send keystrokes to another application, it appears to the other application that the user is typing at the keyboard. The application cannot distinguish between keystrokes sent from your Visual Basic application and keystrokes typed by the user. Thus you can make the other application perform any operation that the user, using keystrokes, can make it perform. For example, this statement sends the keys "ABC" as if they were typed at the keyboard:

```
SendKeys "ABC"
```

When **SendKeys** is used in this way, the keystrokes don't actually get sent until your Visual Basic application is "idle"—when none of your Visual Basic code is executing or when the **DoEvents** statement is called. However, you can force the keys to be sent immediately:

```
SendKeys "ABC", True
```

Passing **True** for the optional second argument on the call to **SendKeys** causes Visual Basic to yield, allowing the keystrokes to be processed, before continuing with the rest of your code.

But where do the keystrokes go? They go to the active application (the application that has the focus). If the currently active application is not the one to which you want to send keystrokes, however, then you'll need to know how to activate other applications.

Activating Other Applications

Keystrokes that you send can go to only one place: the active application. If you do not activate another application, your Visual Basic application will send keystrokes to itself. While this might be handy for testing your application, it is generally not very useful. To send keystrokes to another application, you have to activate that application with the **AppActivate** statement. For example, if the Windows Terminal application was running, this statement would activate it:

```
AppActivate "Terminal"
```

If the application is not already running, you must start it with the **Shell** function (described earlier) before you can activate it and send keystrokes to it.

For More Information For information on the **AppActivate** statement, see the *Language Reference*, or search Help for *AppActivate*.

Specifying Special Characters

You may want to send characters that you cannot simply type into a string, such as ENTER, F1, and TAB. To send these characters, you send the name of the key surrounded by brace characters ({ }). For example, to send the keystrokes for TAB, F1, and ENTER, use this statement:

```
SendKeys "{TAB}{F1}{enter}"
```

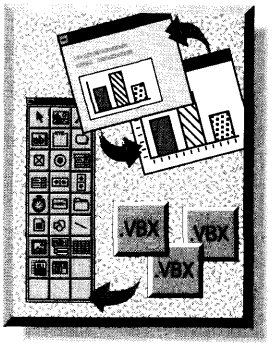
The names of the keys are not case-sensitive, so "enter," "Enter," and "ENTER" all mean the same thing.

Note You cannot send certain keys that generate interrupts rather than character codes, such as CTRL+ALT+DEL and PRINTSCRN.

For More Information For a complete list of the special key names, see the *Language Reference*, or search Help for *SendKeys*.

CHAPTER 22

Object Linking and Embedding (OLE)



Object linking and embedding (OLE) is a method for enhancing the interoperability between applications for Windows. Using OLE, your Visual Basic application can display and manipulate data from other Windows–based applications.

Contents

- Creating Front-End Applications
- OLE Fundamentals
- Using the OLE Control
- Working with Files
- Comparing OLE and DDE

OLE2DEMO.MAK

Many of the code examples in this chapter are taken from the sample application OLE2DEMO.MAK. If you installed the sample applications, you'll find this application in the \OLE2 subdirectory of the main Visual Basic directory (\VB\SAMPLES\OLE2).

Creating Front-End Applications

OLE is ideal for creating front-end applications. In this type of application, each OLE control provides a view of data from a different application. The OLE control lets you display data from another Windows–based application in your Visual Basic application, and allows the user to edit that data from within the application in which it was created.

In Figure 22.1, for example, the Memo form contains four OLE controls that display data from a drawing application, a word processor, a spreadsheet, and a graphing package.

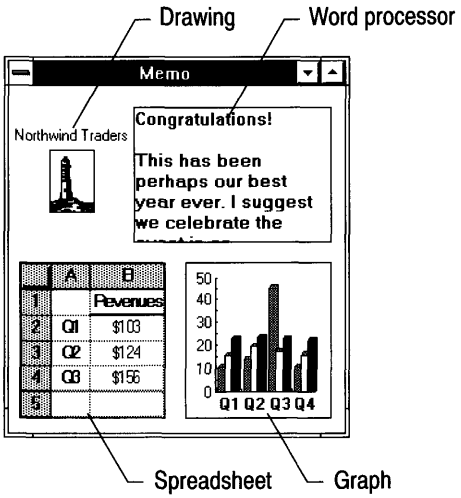


Figure 22.1 Each OLE control can display a view of data from a different application.

This chapter discusses the different ways in which you *link* and *embed* data in the OLE control, how you read and write that data to files, and how you program other characteristics of the control.

About Custom Controls

The OLE control is a custom control. *Custom controls* are separate Visual Basic files with the .VBX extension. To use the OLE control in your application, the file MSOLE2.VBX must be added to the project. The file AUTOLOAD.MAK, located in the Visual Basic startup directory, loads .VBX files automatically. If you ran SETUP.EXE to install Visual Basic on your system, MSOLE2.VBX is one of three custom controls that are added by default each time you open a new project. For more information on how to add a custom control to a project, see the section "Editing the AUTOLOAD.MAK File" in Chapter 5, "Managing Projects," or search Help for *custom controls*.

OLE Fundamentals

Suppose you want to use the OLE control to display data from a Microsoft Excel spreadsheet in your application. To do this, place an OLE control on a form and size it as you would any other control. When the OLE control is drawn on a form, the Insert Object dialog, shown in Figure 22.2, is displayed.

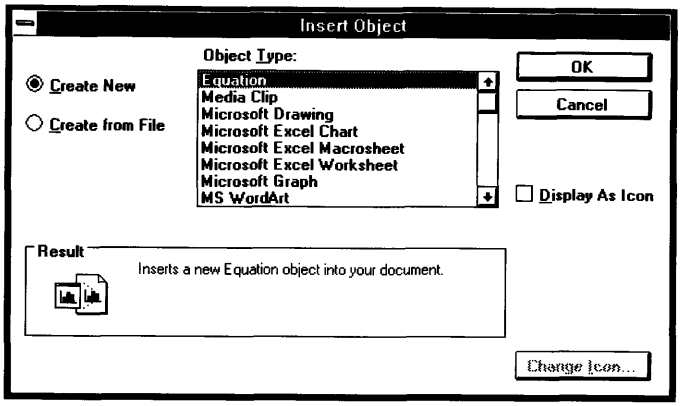


Figure 22.2 The Insert Object dialog

When you select the desired type of object (Excel Worksheet in this example) and click OK, Microsoft Excel is started and a new spreadsheet is opened for editing. After entering the data you want, you close Microsoft Excel and the data is displayed in the OLE control—exactly as it appears in the spreadsheet (see Figure 22.3).

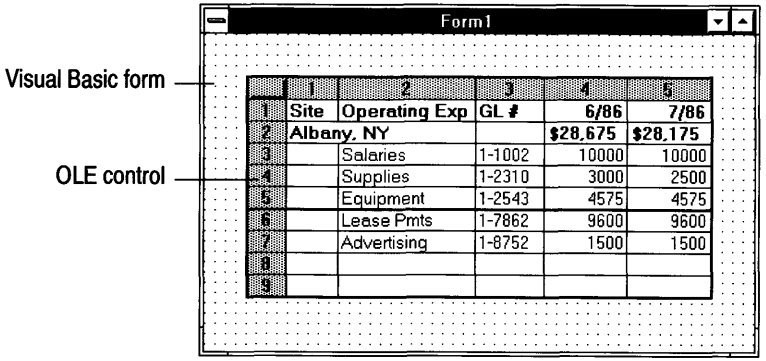


Figure 22.3 When an OLE control contains an object, its data is displayed exactly as it appears in the application that created the object.

At run time, when the user double-clicks the OLE control, Microsoft Excel starts up with the appropriate data automatically loaded, ready for editing. When the user closes Microsoft Excel, the focus is returned to the Visual Basic application, and the data is displayed in the OLE control.

To fully understand the OLE control, you should first be familiar with some specific terms and the concepts of object linking and embedding.

OLE Terms

OLE Object An *OLE object* refers to a discrete unit of data that has been supplied by an OLE application. An application can expose many types of objects. For example, a spreadsheet application may expose a worksheet, macrosheet, chart, cell, or range of cells—all as different types of objects.

You use the OLE control to create linked and embedded objects. When a linked or embedded object is created, it contains the *name* of the application that supplied the object, its data (or, in the case of a linked object, a *reference* to the data), and an *image* of the data. An OLE control can contain only one object at a time. You can create a linked or embedded object by:

- Using the Insert Object or Paste Special dialog (run time and design time).
- Setting the Action property of the OLE control (run time only).
- Using the OLE control's pop-up menus (design time only).

OLE Automation Some applications provide objects that support OLE Automation. You can use Visual Basic to programmatically manipulate the data in these objects. Some objects that support OLE Automation also support linking and embedding. The techniques for accessing OLE Automation objects are discussed in Chapter 23, "Programming Other Applications' Objects."

Note Applications using the OLE control or OLE Automation require Microsoft Windows version 3.1 or later.

You can access the following types of objects using Visual Basic.

Object type	Origin	Purpose
Linked	OLE control	Display an object in a Visual Basic form and allow the user to access the object's data. The object can be accessed by other users and applications. The object's data is maintained by the application that created the object.
Embedded	OLE control	Display an object in a Visual Basic form and allow the user to access the object's data. The object can be accessed only by the Visual Basic application in which its data is maintained.
OLE Automation	**CreateObject** function **GetObject** function	Manipulate an object remotely, from within a Visual Basic application.

Class An object's *class* determines the application that provides the object's data and the type of data the object contains. You can get a list of the class names available to your application by selecting the Class property in the Properties window and clicking the three dots (...) in the Settings box.

Note The Insert Object dialog does not display a list of class names. This dialog displays "user-friendly" names for each class of object, which are generally longer and more easily understood.

Container Application An application that can display other applications' objects is a *container application.* For example, a Visual Basic application that uses an OLE control to embed or link data from another application is a container application.

Note In Visual Basic 2.0, applications that provided objects were called *server* or *source* applications. Applications used to display objects (container applications) were referred to as *client* or *destination* applications.

Contrasting Linked and Embedded Objects

You use the OLE control to incorporate data into a Visual Basic application by linking or embedding that data from another application. Data associated with a *linked object* is stored in the application that created it. Data associated with an *embedded object* is contained in an OLE control in Visual Basic and can be saved by the Visual Basic application.

Linked Objects

When you link an object, you are inserting a *placeholder* (not the actual data itself) for the *linked object* into your application. For example, when you link a range of spreadsheet cells in a Visual Basic application, the data associated with the cells is stored in another file; only a link to the data and an *image* of the data are stored in the OLE control. While working with the Visual Basic application, the user can select the linked object (by double-clicking the object, for example), and the spreadsheet application starts automatically. The user can then edit those spreadsheet cells using the spreadsheet application.

When an object is linked to a Visual Basic application, the object's data can also be accessed from any other applications that contain links to that data, and the data can be changed from within any of them. For example, in Figure 22.4, a graph is linked to a Visual Basic application. The graph is also linked to a Microsoft Word document. If the graph's data is changed by either application, the modified graph will appear in both the Visual Basic application and the Microsoft Word document.

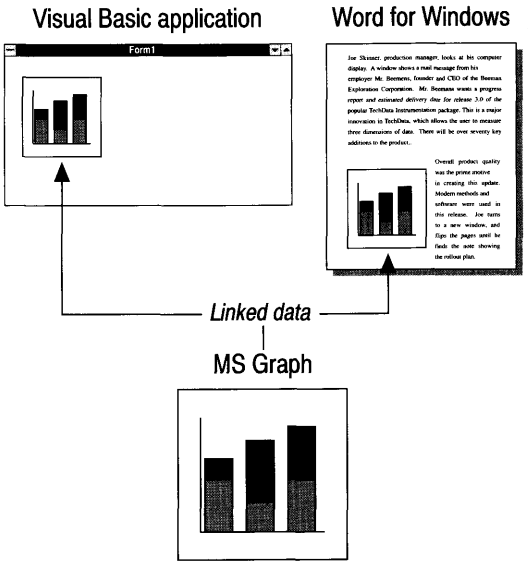


Figure 22.4 Data from a linked application can be accessed from many different applications.

As you can see, linking makes it easy to track identical information that appears in more than one application. Linking is useful when you want to maintain one set of data that is accessed from several applications.

Embedded Objects

When you use an OLE control to create an *embedded object*, all the data associated with the object is contained in the OLE control. When you save the contents of the control to a file, the file contains the name of the application that produced the object, the object's data, and a metafile image of the object.

Unlike linked objects, when an object is embedded in an application, no other application has access to the data in the embedded object. Embedding is useful when you want your application to maintain data that is produced and edited in another application, as shown in Figure 22.5. When the user activates the object (the graph), the application that created the object (MS Graph) is invoked by the container application (your Visual Basic application), and the object's data is opened for editing.

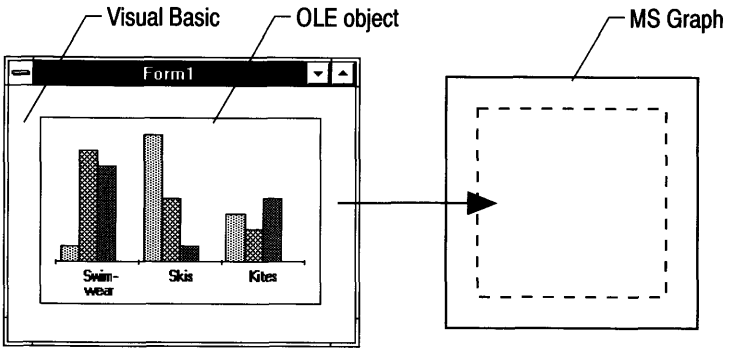


Figure 22.5 Your application maintains data for an embedded object.

Using the OLE Control

You can use the OLE control to display an OLE object on a form. You can create the object either at design time by using standard OLE dialogs or at run time by setting the appropriate properties.

Creating Objects—Design Time vs. Run Time

There are advantages and disadvantages to creating objects at design time and to creating them at run time. You must decide which technique is best for your application.

When you create an executable (.EXE) file, the data associated with an OLE object is incorporated in the executable file. Therefore, if you create an OLE object at design time, the resulting executable file will be larger than if you created the object in code at run time.

It can take several seconds to create an object, regardless of when the object is created. If you create an object at run time, the only logical place to put this code may be in the Form_Load event of the form containing the OLE control. Remember that when you create an object in code, the system suspends all other processing until the object is created. Therefore, if you create an object in a Form_Load event, the form will not be displayed until the object is created.

Creating Objects at Design Time

Each time you draw an OLE control on a form, the Insert Object dialog box is displayed. You use this dialog box, shown in Figure 22.2 earlier in this chapter, to create linked or embedded objects at design time.

When you create an object at design time, Visual Basic automatically sets the appropriate properties for the OLE control. You can view these settings in the Properties window when the OLE control is selected.

Determining How an Object Is Displayed

When creating an object, you use the DisplayType property to determine if the object is to be displayed as an icon (set DisplayType = 1), or if the object's data is to be displayed in the control (set DisplayType = 0). This property also determines the default setting of the Icon check box in the Insert Object and Paste Special dialogs. When you display these dialogs, either at design time or at run time (set Action = 14 or 15), the Icon check box is automaticaly selected if this property is set to 1 (Icon). Once an object is created you cannot change its display type.

You use the SizeMode property to determine how an object's image is displayed in the OLE control. You can either clip the image at the OLE control's borders (SizeMode = 0 - Clip), stretch the image to fill the OLE control (SizeMode = 1 - Stretch), or resize the OLE control to fit the image (SizeMode = 2 - AutoSize).

Caution When SizeMode = 2, the OLE control will change size at run time to accommodate the size of the object it contains.

Creating Linked Objects

When you create a linked object, the data displayed in the OLE control exists in one place—the linked file. Since the data exists in a file, other applications can view and modify the file (see Figure 22.4 earlier in this chapter). The OLE control maintains the object's link information, such as the name of the application that supplied the object, the name of the linked file, and an image of the linked data.

▶ **To create a linked object using the Insert Object dialog**

1. Draw an OLE control on the form.

   The Insert Object dialog is displayed. You can also display this dialog by clicking the OLE control with the right mouse button and choosing the Insert Object command.

2. In the Object Type list box, highlight the type of object you want to create.
3. In the Insert Object dialog, select the Create from File option button.

4. Choose the Browse button.

   A Browse dialog is displayed.

5. Use the Browse dialog to select the file you want to link.
6. Click OK to return to the Insert Object dialog.
7. In the Insert Object dialog, select the Link check box and click OK to create the object.

When you use a linked object, every user that runs your application must have access (a valid path) to the linked file. Otherwise, when your application is run, an image of the original data is displayed, but the user will not be able to modify the data, nor will the user see changes others may have made to the linked data.

If you allow the user to make changes to the linked data, you do not need to write any code to maintain these changes. The user can save the linked data using the application that created the object. For example, if you create an object linked to a Microsoft Excel spreadsheet, the user can activate Microsoft Excel at run time by double-clicking the object. When the spreadsheet is opened, the user makes any desired changes and saves the spreadsheet using Microsoft Excel's File Save command. When the user closes Microsoft Excel, the new data is displayed in the OLE control.

An object's image is not automatically updated when the form containing the object is loaded. If your application contains a linked object, it is possible for that object's data to be changed by another application when your application is not running. The next time your application is run, the new data will not be displayed in the OLE control. To update the OLE control to display the current data, set Action = 6 (OLE_UPDATE).

Linking to Data Within a File

When creating a linked object, you may not want to link an entire file. For example, you may want to link a range of cells in a spreadsheet or a paragraph in a word processor.

One way to do this is to use the SourceItem property to specify the data within the source document you want to link. The exact syntax used to set this property varies depending on the application supplying the object. For example, when linking to a Microsoft Excel spreadsheet, you specify SourceItem using an *R1C1* syntax. This type of syntax is commonly used by spreadsheet applications to indicate the rows and columns in a range of cells. Here's how to specify the range A1:D5 in a Microsoft Excel spreadsheet using R1C1 syntax:

```
R1C1:R5C4
```

Alternately, you specify a range of cells using a named range.

An easier way to link to data within a file is to copy the desired data to the Clipboard and choose Paste Special from the OLE control's pop-up menu (click the OLE control with the right mouse button). Pop-up menus are explained later in this chapter.

Creating Embedded Objects

When you create an embedded object, you can either create an object from a file or create a new object. When you create an object from a file, a copy of the specified file's data is displayed in the OLE control. When you create a new object, the application that created the object is invoked and the user is allowed to enter any desired data. New embedded objects are usually created at run time to allow the user to enter data in the object.

Typically, you create embedded objects that display existing data at design time. This allows you to view the object's data as it will appear to the user. You can then move and size the OLE control and the other controls on the form and create your application's user interface accordingly.

You display existing data in an embedded object by creating the object using an existing file as a template. The OLE control then contains an image of the file along with a copy of the embedded file's data. Since the OLE control contains the embedded data, an application that displays data using an embedded object will be larger than an application that displays the same data using a linked object.

▶ **To create an embedded object using an existing file**

1. Display the Insert Object dialog.
2. In the Object Type list box, highlight the type of object you want to create.
3. In the Insert Object dialog, choose the Create from File option button.
4. Choose the Browse button.

   A Browse dialog is displayed.
5. Use the Browse dialog to select the file you want to embed.
6. Click OK to return to the Insert Object dialog.
7. In the Insert Object dialog, click OK to create the object.

Data in an embedded object is not persistent. If the user modifies the object's data, you must write code to save the data if you want the changed data to appear the next time your application is run. Saving embedded data to a file is discussed in the section "Working with Files" later in this chapter.

Using the OLE Control's Pop-up Menus

At design time, you click the OLE control with the right mouse button to display a pop-up menu. The commands displayed on the pop-up menu depend on the state of the OLE control. If the control contains an embedded object, the pop-up menu shown in Figure 22.6 is displayed.

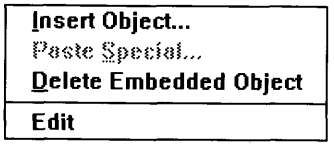


Figure 22.6 The pop-up menu contains these commands when the OLE control contains an object.

You use the Insert Object command to delete an existing object and create a new one. The Paste Special command allows you to paste an object from the Clipboard into the OLE control. The commands displayed below the separator bar are determined by the application that created the object. These commands reflect the *verbs* that the object supports. A verb is an action, such as play or edit, that the user can perform on an object.

For More Information For information on verbs, search Help for *Verb*.

When the OLE control is empty (that is, when it does not contain an object), the menu shown in Figure 22.7 is displayed.

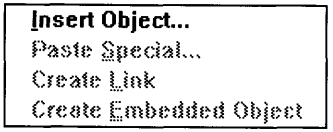


Figure 22.7 The pop-up menu contains these commands when the OLE control is empty.

In Figure 22.7, the Create Link and Create Embedded Object commands are grayed (disabled). The Paste Special command is grayed unless there is an OLE object on the Clipboard.

To activate the Create Link and Create Embedded Object commands, enter a valid class name in the Properties window. In addition, to activate the Create Link command, you must enter a value for the SourceDoc property. If the OleTypeAllowed property is set to a value other than 2 - Either, only the appropriate menu item is activated. When OleTypeAllowed is 0, Create Embedded Object is omitted from the menu. When OleTypeAllowed is 1, Create Link is omitted.

Tip To choose from a list of valid class names, select the Class property in the Properties window and click the three dots (...) in the Settings box.

Creating Objects Using Paste Special

The Paste Special dialog, shown in Figure 2.8, allows you to create linked and embedded objects from data that has been copied to the Clipboard. To display the Paste Special dialog at design time, click the OLE control with the right mouse button and select Paste Special from the pop-up menu.

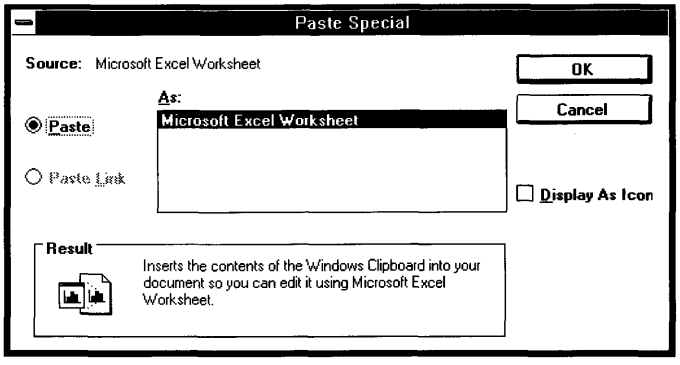


Figure 22.8 The Paste Special dialog

▶ To create an object using the Paste Special dialog

1. Run the application containing the data you want to link or embed.
2. Select the data you want to link or embed.
3. From the Edit menu, choose Copy.

   The data is copied to the Clipboard.
4. In Visual Basic, click the OLE control with the right mouse button and choose the Paste Special command.
5. Choose the Paste option button to create an embedded object.

   –or–

   Choose the Paste Link option button to create a linked object.
6. Click OK to create the object.

Creating Objects at Run Time

To create objects at design time, you use the Insert Object and Paste Special dialogs. To create an OLE object at run time, however, you set properties in code.

Setting Properties

To create an OLE object at run time, set the following properties.

Property	Description
Class	Identifies the application that created the object.
OleTypeAllowed	Determines the type of object you can create (linked, embedded, or either).
SourceDoc	When creating a linked object, this property determines the file to link to.
	When creating an embedded object, this property determines the file to use as a template.
SourceItem	Specifies data within a file to link to (linked objects only).
Action	Specifies the operation to perform on an OLE control.

Class Property

The Class property determines the *type* of data that the object contains and the *name* of the application that supplied the object. For example, to create an object that contains data from a Microsoft Excel spreadsheet, you set the Class property to "ExcelWorksheet."

You can get a list of the class names available to your application by highlighting the Class property in the Properties window and clicking the three dots (...) in the Settings box.

Important When you create an object, the executable file associated with the object's class must be either active or on the system's path.

OleTypeAllowed Property

The OleTypeAllowed property determines the type of object you can create. You set the OleTypeAllowed property to OLE_LINKED, OLE_EMBEDDED, or OLE_EITHER.

Note The constants OLE_LINKED, OLE_EMBEDDED, and OLE_EITHER, along with the other OLE constants used in this chapter, are defined in the CONSTANT.TXT file. You can use these constants in your code if you load this file into any of your application's code modules.

For most situations, setting OleTypeAllowed to OLE_EITHER (the default) will suffice. If your application allows users to create OLE objects, then you may want to restrict the type of objects they can create using this property. For more information on writing applications that allow the user to create objects, see the section "Displaying the OLE Dialogs" later in this chapter.

Note Use the OleType property to determine the state of an object in an OLE control (Linked, Embedded, None).

SourceDoc and SourceItem Properties

To create a linked object or display existing data in an embedded object, you must also set the SourceDoc property.

The SourceDoc property specifies the name of the document you want to link to (or the name of the file you want copied to an embedded object). When you set the SourceDoc property, you must include the path and the name of the file you want to link. For example:

```
Ole1.SourceDoc = "C:\REVENUES\QUARTER1.XLS"
```

Tip At design time, when you highlight the SourceDoc property in the Properties window and click the three dots (...) in the Settings box the Insert Object dialog is displayed. You can use this dialog to browse your disk and create a linked or embedded objecct from a selected file.

For linked objects, you can optionally specify data within the source document to link. You do this using the SourceItem property.

For More Information For information on the SourceDoc and SourceItem properties, search Help for *SourceDoc* and *SourceItem*.

Action Property

The Action property specifies the operation to perform on an OLE control. Setting this property serves the same function as a Visual Basic method—it performs an action on an object. For example, you can use the Action property to create, activate, or delete an object.

For More Information For a list of operations that can be performed with the OLE control, search Help for *Action*.

Creating a Linked Object at Run Time

Before creating a linked object, you must first specify the document you want to link to by setting the SourceDoc property. Optionally, you can specify the data within the document that you want to link by setting the SourceItem property. The document specified in the SourceDoc property must already exist.

The following code fragment creates a linked object at run time:

```
' Define the constant.
Const OLE_CREATE_LINK = 1
' Specify the application that will supply the object.
Ole1.Class = "ExcelWorksheet"
' Specify the file to link to.
Ole1.SourceDoc = "C:\EXCEL\TEST.XLS"
' Create the linked object (OLE_CREATE_LINK = 1).
Ole1.Action = OLE_CREATE_LINK
```

When you create a linked object, the data specified by the SourceDoc and SourceItem properties is displayed in the OLE control.

If the AutoActivate property is set to 2 - Double Click (the default), double-clicking an object at run time automatically activates the object. Exactly how an object behaves when it is activated depends on the object's default verb.

Note When AutoActivate is **True**, the DblClick event does not occur when the user double-clicks an OLE control.

For More Information For information on verbs, search Help for *Verb*.

Creating an Embedded Object at Run Time

When creating an embedded object at run time, you display existing data by creating the object from a file template, or you create a new object and let the user enter new data.

When you create an embedded object from a file, the specified file's data is displayed in the OLE control. This is typically done in a front-end application in which you want to display, and provide access to, data in another application.

▶ To create an embedded object from a file at run time

1. Set the Class property to determine the object's type.
2. Set the SourceDoc property to specify the file to use as a template.
3. Set Action = OLE_CREATE_EMBED.

The following code fragment creates an embedded object using an existing file as a template for the object.

```
' Define the constant.
Const OLE_CREATE_EMBED = 0
' Specify the type of object.
Ole1.Class = "ExcelWorksheet"
' Specify the file to be embedded.
Ole1.SourceDoc = "Q1PROFIT.XLS"
' Create embedded object (OLE_CREATE_EMBED = 0).
Ole1.Action = OLE_CREATE_EMBED
```

When you create an empty embedded object, it is a good idea to activate the application providing the object once the object is created. This allows the user to enter any desired data into the application. The user then displays this newly entered data in the OLE control by choosing the application's Update command (this menu command should appear on the application providing the object, not in your Visual Basic application).

▶ To create an empty embedded object at run time

1. Set the Class property to determine the object's type.
2. Set Action = OLE_CREATE_EMBED.
3. Set Action = OLE_ACTIVATE

The following code fragment creates an empty embedded object, then activates the application that created it.

```
' Specify the type of object.
Ole1.Class = "ExcelWorksheet"
' Create an embedded object (OLE_CREATE_EMBED = 0).
Ole1.Action = OLE_CREATE_EMBED
' Invoke the application that created the object.
Ole1.Action = OLE_ACTIVATE
```

This technique is useful when creating a document-centered application in which you allow the user to embed different types of data in a single document. This is discussed in more detail in the section "Displaying the OLE Dialogs" later in this chapter.

Activating an Object

When you activate an object, you perform an action on the object using the object's default verb (or the verb you specify using the Verb property). You activate an object either programmatically or by setting the AutoActivate property.

At run time you activate an object in code by setting Action = 7. By activating an object in this way, you maintain control over when an object is activated. Use the AutoActivate property to allow users to activate an object at their convenience.

In-Place Activation

Some objects can be activated from within the OLE control. When such an object is activated, the user can edit the object (or perform some other action) from inside the boundaries of the OLE control. This feature is called *In-Place Activation*. If an object supports In-Place Activation, you set AutoActivate to 1 (GetFocus) to activate an object when the OLE control gets the focus.

Note Only embedded objects support In-Place Activation.

You typically activate objects that don't support In-Place Activation when the user double-clicks an object. To do this automatically, set AutoActivate to 2. When AutoActivate is set to 2 (Double Click), at run time the object is activated (using its default verb) whenever the user double-clicks the OLE control or presses ENTER when the object has the focus.

Displaying an Object's Verbs

When the AutoVerbMenu property is set to **True**, a pop-up menu showing all the object's available verbs is displayed when the user clicks the OLE control with the right mouse button.

Note When the AutoVerbMenu property is set to **True**, Click events and MouseDown events do not occur when the user clicks the OLE control with the right mouse button.

The list of verbs an object supports may vary, depending on the state of the object. For example, it may make sense for an object to allow the user to *play* data, *edit* data, or both *play* or *edit* data. To update the list of verbs that an object supports, set Action = 17 (Fetch Verbs). Be sure to update the list of verbs before presenting the list of available verbs to the user.

When AutoVerbMenu is set to **True**, the verb list is updated automatically before the pop-up menu is displayed.

For More Information For information on verbs, search Help for *Verb*.

Displaying the OLE Dialogs

At run time, you use the Action property to display the Insert Object or Paste Special dialog to the user. The Insert Object dialog presents a list of available objects and creates an object based on user selection. The Paste Special dialog allows the user to paste an object from the Clipboard to an OLE control.

Using these dialogs, you allow the user to decide what type of object to create. You may do this, for example, when creating a document-centered application. In such an application, the user combines data from different applications to create a single document. This type of application is easy to visualize in terms of a word processor in which the user might enter some text and then embed a spreadsheet and a chart.

To create such an application in Visual Basic, you need to provide some sort of document into which the user can embed objects. For example, you could write a calendar application in which each month is maintained as a separate document. You allow the user to embed an object in the calendar by displaying the Insert Object or Paste Special dialog.

You display the Insert Object or Paste Special dialog at run time by setting the Action property. For example:

```
Sub InsObjectDlg_Click ()
    ' Display Insert Object dialog (OLE_INSERT_OBJ_DLG = 14).
    Ole1.Action = OLE_INSERT_OBJ_DLG
    ' Check to make sure object was created (OLE_NONE = 3).
    If Ole1.OleType = OLE_NONE Then
        MsgBox "Object Not Created."
    End If
End Sub

Sub PstSpecialDlg_Click ()
    ' Determine if Clipboard contents can be pasted into the
    ' OLE control.
    If Ole1.PasteOK Then
        ' Display Paste Special dialog (OLE_PASTE_SPECIAL_DLG = 15)
        Ole1.Action = OLE_PASTE_SPECIAL_DLG
        ' Check to make sure object was created.
        If Ole1.OleType = OLE_NONE Then
            MsgBox "Object Not Created."
        End If
    End If
End Sub
```

Once the dialog is displayed, you do not need to write any more code to create the object. The user makes choices in the dialogs and clicks OK to create an object. If the user cancels a dialog, an object is not created.

As shown in the above examples, you may want to query the value of the OleType property to check the state of the OLE control.

If OleType is...	Then...
OLE_LINKED	The OLE control contains a linked object.
OLE_EMBEDDED	The OLE control contains an embedded object.
OLE_NONE	The OLE control does not contain an object.

Working with Files

Data associated with an OLE object is not persistent; that is, when a form containing an OLE control is closed, any data associated with that control is lost. To save the data from an OLE object to a file, you use the Action property. Once this data has been saved to a file, you can open the file and restore the OLE object.

Objects in the OLE control can be saved only to open, binary files.

▶ To save the data from an OLE object

1. Open a file in binary mode.
2. Set the FileNumber property to the file number used in step 1.
3. Set the Action property to OLE_SAVE_TO_FILE (11).

The SaveObject_Click event procedure illustrates these steps:

```
Sub SaveObject_Click ()
    ' Get file number.
    FileNum = FreeFile
    ' Open file to be saved.
    Open "FOO.OLE" For Binary As #FileNum
    ' Set the file number.
    Ole1.FileNumber = FileNum
    ' Save the file (OLE_SAVE_TO_FILE = 11).
    Ole1.Action = OLE_SAVE_TO_FILE
    ' Close the file.
    Close #FileNum
End Sub
```

Once an OLE object has been saved to a file, it can be opened and displayed in an OLE control.

Note When you set Action = 11 (Save To File) or 12 (Read From File), the file position is located immediately following the object.

▶ To read data from a file into an OLE control

1. Open the file in binary mode.
2. Set the FileNumber property to the file number used in step 1.
3. Set the Action property to OLE_READ_FROM_FILE (12).

The OpenObject_Click event procedure illustrates these steps:

```
Sub OpenObject_Click ()
    ' Get file number.
    FileNum = FreeFile
    ' Open the file.
    Open "FOO.OLE" For Binary As #FileNum
    ' Set the file number.
    Ole1.FileNumber = FileNum
    ' Open the file (OLE_READ_FROM_FILE = 12).
    Ole1.Action = OLE_READ_FROM_FILE
    ' Close the binary file.
    Close #FileNum
End Sub
```

Tip The Updated event is invoked each time the contents of an object is changed. This event is useful for determining if an object's data has been changed since it was last saved. To do this set a global variable in the Update event indicating the object needs to be saved. When you save the OLE object, reset the variable.

Comparing OLE and DDE

OLE and dynamic data exchange (DDE) allow you to perform similar functions. Both allow you to display data from other applications in your Visual Basic application. There are, however, some fundamental differences between OLE and DDE.

The behavior of the OLE control differs from standard DDE operations in that program control is actually transferred to the other application *temporarily* for the purpose of editing the data (in some cases you can perform actions other than editing). In addition, the data always appears in the same view that it would in that application. For example, when the other application is Microsoft Excel, the OLE control displays spreadsheet data in the Visual Basic application as it would appear in Microsoft Excel itself.

Consider an application that uses OLE links to display another application's data, such as the application displayed in Figure 22.9.

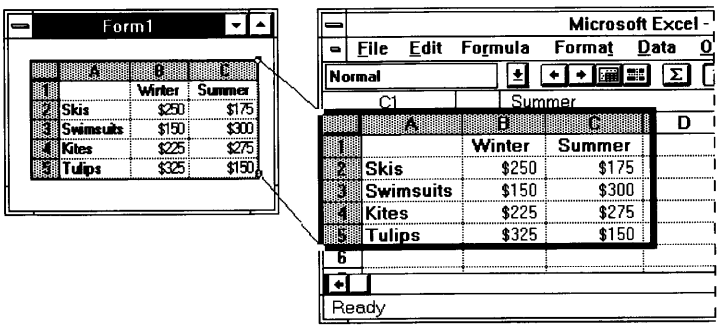


Figure 22.9 When you link an object using OLE, the linked data is displayed in the OLE control as it appears in the application that created the object.

One advantage of using OLE in this type of application is that you don't have to do any work to display the linked data's image. The image appears exactly the way it is in the original spreadsheet. This feature makes OLE especially useful when you link to an application whose image is difficult to recreate, such as a graphics or drawing package.

Warning The image that is displayed in a linked object is determined by the application providing the object. Some applications may not display linked data as you would expect. For example, when you link to a Microsoft Word for Windows 2.0 document, the Word icon is displayed in the OLE control, not the linked text.

Another advantage of linking with OLE is that you can allow the user to edit the linked data in its native environment when the user double-clicks an object by setting a single property:

```
Ole1.AutoActivate = 2        ' Double-click.
```

With previous versions of OLE, it was difficult to access linked or embedded data programmatically. For example, if you created a linked object from a Microsoft Excel spreadsheet, you could allow the user to edit the spreadsheet's data, but it was difficult (if not impossible) for you to access that data programmatically.

With the advent of OLE Automation, it is easy to programmatically access and manipulate data in an OLE object—as long as the application that supplied the object supports OLE Automation. Using OLE Automation to access data in OLE objects is discussed in Chapter 23, "Programming Other Applications' Objects."

The Data and DataText properties can be used to send data to and from an OLE object. However, use of these properties usually requires that you fully understand the format of the object's data. If you need programmatic access to data in an application that does not support OLE Automation, you'll probably still need to use DDE.

For More Information For information on using OLE to send data to and from an application that does not support OLE Automation, search Help for *DataText* and *Data*.

Distributing OLE Applications

When you create and distribute applications that use the OLE control, you should install the following files in the customer's Microsoft Windows \SYSTEM directory:

- COMPOBJ.DLL
- MSOLE2.VBX
- MSOLEVBX.DLL
- OLE2.REG
- OLE2.DLL
- OLE2CONV.DLL
- OLE2DISP.DLL
- OLE2NLS.DLL
- OLE2PROX.DLL
- STORAGE.DLL
- VBOA300.DLL

You also need to make sure OLE 2.0 is registered on the user's machine and that SHARE.EXE is installed in the user's AUTOEXEC.BAT file (SHARE.EXE is not required if the user is running Microsoft Windows for Workgroups). Use the SetupWizard to create distribution disks that do this.

▶ To install OLE 2.0 without using the SetupWizard or the Setup Toolkit

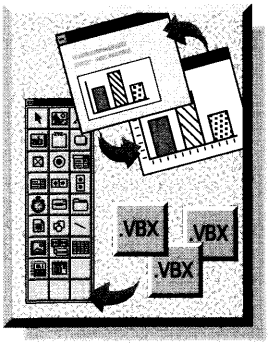
1. Check to see if OLE 2.0 has already been installed. If OLE2.DLL is on the user's machine, then OLE 2.0 has already been installed. If the user does not have the most current version of OLE2.DLL, you can install the new version using the following steps.
2. Copy the above files to the user's Microsoft Windows \SYSTEM directory.
3. Register OLE 2.0 using the following command line:

```
REGEDIT /S OLE2.REG
```

For More Information For information on the SetupWizard and on distributing applications, see Chapter 25, "Distributing Your Applications," or search Help for *Setup Kit*.

CHAPTER 23

Programming Other Applications' Objects



The ability to share data between applications has been one of the key features of the Microsoft Windows operating system. Early versions of Windows used the Clipboard to transfer data. In later versions, dynamic data exchange (DDE) allowed you to transfer data and even send commands to other applications. Using OLE Automation, you can do much more than share data. From within Visual Basic, you can remotely access and manipulate object linking and embedding (OLE) objects that are supplied by other applications. With OLE Automation, you can use Visual Basic to orchestrate custom solutions that utilize data and features from applications that support OLE Automation.

The previous chapter showed you how to create and display objects using the OLE control. This chapter shows how to create and manipulate objects in other applications using OLE Automation.

Note Applications using the OLE control or OLE Automation require Microsoft Windows version 3.1 or later.

Contents

- What Is OLE Automation?
- Accessing OLE Objects
- Creating Invisible Objects
- Using Existing Objects

What Is OLE Automation?

OLE Automation is an industry standard that applications use to expose their OLE objects to development tools, macro languages, and container applications that support OLE Automation. For example, a spreadsheet application may expose a worksheet, chart, cell, or range of cells—all as different types of objects. A word processor might expose objects such as applications, paragraphs, sentences, bookmarks, or selections.

When an application supports OLE Automation, the objects it exposes can be accessed by Visual Basic. You use Visual Basic to manipulate these objects by invoking methods on the objects, or by getting and setting the objects' properties —just as you would with Visual Basic's own objects.

Distribution Note When you create and distribute applications that use OLE Automation, you should install the files COMPOBJ.DLL, OLE2.REG, OLE2.DLL, MSOLE2.VBX, MSOLEVBX.DLL, OLE2CONV.DLL, OLE2DISP.DLL, OLE2NLS.DLL, OLE2PROX.DLL, STORAGE.DLL, and VBOA300.DLL in the customer's Microsoft Windows \SYSTEM directory.

You also need to make sure OLE 2.0 is registered on the user's machine and that SHARE.EXE is installed in the user's AUTOEXEC.BAT file (SHARE.EXE is not required if the user is running Microsoft Windows for Workgroups). Use the Setup Wizard to create distribution disks that do this.For more information on the Setup Wizard and on distributing applications, see Chapter 25, "Distributing Your Applications," or search Help for *Setup Kit*. For more information on distributing OLE applications, see Chapter 22 "Object Linking and Embedding."

Accessing OLE Objects

You can manipulate other applications' OLE objects remotely, without creating a linked or embedded object, using the **CreateObject** function.

Before creating an object, you define a variable that can be used to reference the object. You do this by dimensioning a variable of type **Object**. For example:

```
Dim Spread As Object
```

You then use the **CreateObject** function to create the object. This function requires a single parameter—a string that indicates the application name and the type of object you want to create. Use the following syntax to specify an object to create:

Application.ObjectType

For example, let's say there is a spreadsheet application named SPDSHEET.EXE that supports two objects: a worksheet and a chart. These objects would be defined as:

```
Spdsheet.Worksheet
Spdsheet.Chart
```

To get a list of objects that an application supports, consult that application's documentation.

Once you know the type of object you want to create, you use the **Set** keyword to assign the object returned by the **CreateObject** function to the object variable. Here's an example:

```
Set Spread = CreateObject("Spdsheet.Worksheet")
```

When this code executes, the application providing the object is started (if it is not already running) and an object is created. Unlike the image displayed when you create a linked or embedded object with the OLE control, the object's image is not displayed anywhere in Visual Basic, nor is the object's data maintained by Visual Basic. The object is part of the application that created it. This object can be referenced in Visual Basic code using the object variable you defined. For example, after creating the object, you could write code such as this to change the current cell, insert text, and save the object to a file:

```
Spread.Row      = 4
Spread.Column   = 2
Spread.Insert   = "Hello, world."
Spread.SaveAs   "C:\SPD\OLETEST.SPD"
```

Tip When creating an object, some applications require that the application providing the object is either active or on the system's path.

Creating Invisible Objects

Some applications may provide objects that are never displayed to the user. For example, a word processing application may expose its spelling checker engine as an object. Let's say this object supports a method called *CheckWord* that takes a string as an argument. If the string is spelled correctly, **True** is returned; otherwise, the method returns **False**. If the string is spelled incorrectly, then you could pass it to another (hypothetical) method called *SuggestWord* that takes a misspelled word as an argument and returns a suggestion for its correct spelling. The code might look something like this:

```
Sub CheckSpelling ()
    Dim ObjVar As Object
    Dim MyWord, Result

    MyWord = "potatoe"

    ' Create the object.
    Set ObjVar = CreateObject("WordProc.SpellCheck")

    ' Check the spelling.
    Result = ObjVar.CheckWord (MyWord)

    ' If False, get suggestion.
    If Not Result Then
        MyWord = ObjVar.SuggestWord (MyWord)
    End If
End Sub
```

In the above example, the spelling checker is never displayed to the user. Its functionality is exposed through the properties and methods of the spelling checker object.

As shown in the example, you create and reference invisible objects the same way as any other type of object. Use the **CreateObject** function to create the object; then reference the object using an object variable.

Using Existing Objects

Objects can exist on your system in several ways. Chapter 22, "Object Linking and Embedding (OLE)," shows how you create linked and embedded objects using the OLE control. Applications that create OLE Automation objects can save objects in files on disk. This section shows you how to access objects that already exist on your system.

Accessing Linked and Embedded Objects

You use the OLE control to create and display linked and embedded objects in a Visual Basic application. Some applications that supply objects support both linking and embedding and OLE Automation. If you use the OLE control to create a linked or embedded object and that object supports OLE Automation, you can access that object's properties and methods in Visual Basic using the Object property. The Object property returns the object in the OLE control. This property refers to an OLE object in the same way an object variable created using the **CreateObject** or **GetObject** functions refers to an object.

For example, let's say an OLE control named Ole1 contains an object that supports OLE Automation. Let's also say that this object has an **Insert** method, a **Select** method, and a Bold property. In this case, you could write the following code to manipulate the OLE control's object:

```
' Insert text in the object.
Ole1.Object.Insert "Hello, world."
' Select the text.
Ole1.Object.Select
' Format the text as bold.
Ole1.Object.Bold = True
```

Activating an Object from a File

Many OLE Automation applications allow the user to save objects in files. For example, a spreadsheet application that supports worksheet objects allows the user to save the worksheet in a file. The same application may support a chart object that the user can also save in a file. You use the **GetObject** function to activate an object that has been saved to a file.

To activate an object from a file, first dimension an object variable. You then call the **GetObject** function using the following syntax:

GetObject (*filename*[, *class*])

The *filename* argument is a string containing the full path and name of the file you want to activate. For example, let's say an application named SPDSHEET.EXE created an object that was saved in a file called REVENUE.SPD. The following code invokes SPDSHEET.EXE, loads the file REVENUE.SPD, and assigns REVENUE.SPD to an object variable:

```
Dim Spreadsheet As Object
Set Spreadsheet = GetObject("C:\ACCOUNTS\REVENUE.SPD")
```

If the *filename* argument is set to an empty string (""), the **GetObject** function returns the currently active object of the specified *class*. If there is no object of that type active, an error occurs.

The above example shows how to activate an entire file. However, some applications let you activate part of an object. To specify that you want to activate part of a file, add an exclamation point (!) or a backslash (\) to the end of the file name followed by a string that identifies the part of the file you want to activate. See the object's application documentation for information on how to create this string.

For example, let's say SPDSHEET.EXE is a spreadsheet application that uses R1C1 syntax. The following code could be used to activate a range of cells within REVENUE.SPD:

```
Set Spreadsheet = GetObject("C:\ACCOUNTS\REVENUE.SPD!R1C1:R10C20")
```

In the previous two examples, an application is invoked and an object is activated. Notice that in these examples the application name (SPDSHEET.EXE) is never specified. When you use **GetObject** to activate an object, the OLE DLL files determine the application to invoke and the object to activate based on the file name you provide. Some files, however, may support more than one class of object. Let's say the spreadsheet file, REVENUE.SPD, supports three different classes of objects: an application object, a worksheet object, and a toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, you use the optional *Class* parameter. Here's an example of activating the toolbar object in the file REVENUE.SPD:

```
Set Spreadsheet = GetObject("C:\REVENUE.SPD", "SPDSHEET.TOOLBAR")
```

Note When you use the **GetObject** function and do not include the *Class* of the object to be activated, then the default object of the specified file is activated.

Manipulating Objects

Once you have created a variable that references an OLE object, the object can be manipulated in Visual Basic in the same way as any Visual Basic object (such as a control). You use the *object.property* syntax to get and set the object's properties or to perform methods on the object.

Accessing an Object's Properties

To assign a value to a property of an object, put the object variable and property name on the left side of an equation and the desired property setting on the right side. For example:

```
Dim ObjVar As Object
Dim RowPos, ColPos
Set ObjVar = CreateObject("MyApplication.MyObjectType")
```

```
ObjVar.Text = "Hello, world"
ObjVar.Cell(RowPos, ColPos) = "Put this text in the specified cell."
```

You can also retrieve property values from an object:

```
Dim X

X = ObjVar.Text
X = ObjVar.Range(12, 32)

' Sets the font for ObjVar.Selection.
ObjVar.Selection.Font = 12
```

Note All arguments to OLE Automation objects use the **Variant** data type. When retrieving a value from a property or method, OLE Automation objects always return values with the **Variant** data type.

If you assign a variable with a data type other than **Variant** when setting a property value or performing a method, the variable is coerced to the **Variant** data type when the property is set or the method is performed.

Performing Methods

In addition to getting and setting properties, you can manipulate an object using the methods it supports. Some methods may return a value.

```
' IsBold method returns True or False.
X = ObjVar.Text.IsBold
If X Then
    ObjVar.Text = "The text is bold."
Else
    ObjVar.Text = "The text is not bold."
End If

' This method requires two arguments.
ObjVar.Move XPos, YPos
```

Methods that do not return a value behave like a subroutine. If you assign such a method to a variable, an error occurs.

Some objects contain subobjects. For example, a cell could be considered a subobject of a spreadsheet object. You can include multiple objects, properties, and methods on the same line of code using the dot syntax, just as you would with a Visual Basic object (for example, Form.Control.Property).

For example:

```
ObjVar.Cell(1,1).FontBold = True
```

Closing an Object

All OLE Automation objects support some method that closes the object and the application that created it. Since OLE objects can use a significant amount of memory in your Visual Basic application, it is a good idea to explicitly close an object when you no longer need it. To close an object, use the appropriate method (most objects support the **Close** method or the **Quit** method). For example:

```
' Closes the object.
ObjVar.Close
' Closes the application that created the object.
ObjVar.Quit
```

The user can close an object using the application that created the object.

Note When an object variable loses scope, the object and its application are not closed. However, you can no longer use that object variable to refer to the object. If the object is still active, you can use the **GetObject** function to assign another object variable to the object.

Limitations in Visual Basic

Some OLE objects support features that can't be accessed using Visual Basic. This section discusses those features.

Arrays and User-defined Types

Some objects have properties and methods that return an array of data or take an array as an argument. The following is a list of Visual Basic's limitations when using arrays with OLE objects. You cannot:

- Use an array or a user-defined type as an argument to a method.
- Set a property using an array or a user-defined type.
- Assign the return value of a property or method to an array variable or a variable of a user-defined type.

When a property or method returns an array, you can use the **LBound** and **UBound** functions to determine the size of an array. You can then access the individual elements of the array. For example:

```
Dim I, Value

For I = LBound(ObjVar.Selection) To UBound(ObjVar.Selection)
    Value = ObjVar.Selection(I)
Next
```

Named Arguments

You cannot use named arguments when calling an object's methods in Visual Basic.

Some objects have methods that require a plethora of arguments, many of which are optional. These methods are typically used to display one of the application's dialogs. The arguments to the method correspond to selections in the dialog. Such methods often allow you to set any desired arguments using predefined names. When calling a method using named arguments, the order in which the arguments appear does not matter. In Visual Basic, however, you cannot use named arguments. When calling a method that supports named arguments, you must specify each argument in the correct order. If you want to omit an optional argument, leave it blank.

For example, let's say an object supports a method called **FileOpen** that is used to display the object's File Open dialog. The following code shows how the method is called using named arguments and how to call the method from within Visual Basic:

Not Valid in Visual Basic

```
' When using named arguments, the order does not matter.
FileOpen Name = "MYDOC.DOC", ReadOnly = 0, Password = "Mahler"
FileOpen ReadOnly = 0, Name = "MYDOC.DOC", Password = "Mahler"
```

Valid in Visual Basic

```
' Each argument must be specified in the correct order.
ObjVar.FileOpen "MYDOC.DOC", 0, "Mahler"
' To omit the second argument, leave it blank.
ObjVar.FileOpen "MYDOC.DOC", , "Mahler"
```

Collections

A collection is an object that contains zero or more objects of a specified type. Most objects that support collections come in pairs—one describes the object in singular form, the other in plural form. For example, a collection of all the menus in an application may be accessed using the Menu object and the Menus object Collections are usually specified using the plural form of an object. For example the collection of a Row object would be called Rows; the collection of a Cell object would be called Cells.

Most collections support a property called Count. This property returns the number of elements in the collection. For example, in the code `x = ObjVar.Rows.Count`, `x` equals the number of rows in the object referenced by the object variable `ObjVar`.

You specify an individual element of a collection using an index. For example, Rows(1) specifies the first row in an object and Rows(Rows.Count) specifies the last row.

Iterating Through a Numeric Collection

You may encounter two problems when iterating through a collection of an OLE object:

- There is no guarantee the subscripts of the collection are numeric.
- If a collection does return a numeric subscript, there is no guarantee that the subscripts are contiguous.

Some objects provide methods that allow you to iterate through collections. If an object does not explicitly provide methods for doing this, you may not be able to cycle through an object's collections in your Visual Basic program.

If you know that a collection uses a numeric subscript, you can iterate through it; however, you'll need to provide error trapping since you don't know if the subscripts are contiguous.

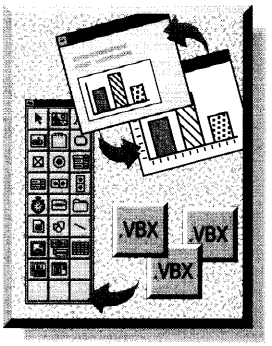
Collections That Are Indexed with Strings

Some collections do not use numeric subscripts. For example, if an object has a collection of styles, instead of using numeric subscripts, the Styles collection might use string subscripts. In code, the collection might look like this:

```
ObjVar.Styles("Normal")
ObjVar.Styles("Example")
ObjVar.Styles("Heading")
```

CHAPTER 24

Calling Procedures in DLLs



Dynamic-link libraries (DLLs) are a key feature of Microsoft Windows. As their name suggests, DLLs are libraries of procedures that applications can link to and use at run time rather than link to statically at compile time. This means that the libraries can be updated independently of the application, and many applications can share a single DLL. In fact, Microsoft Windows itself is composed of several DLLs that contain the procedures all applications use to perform their activities, such as displaying windows and graphics, managing memory, and so on. (These procedures are sometimes referred to as the Windows API, or application programming interface.)

Your Visual Basic applications can call the procedures in these DLLs to perform special actions that you can't perform directly in Visual Basic. You can also call procedures in other DLLs you have in your system. Unless otherwise noted, all discussion in this chapter about making a call to a DLL procedure applies equally to the operating environment DLLs and other DLLs.

Contents

- Overview
- Declaring a DLL Procedure
- Calling a DLL Procedure
- Special Considerations When Declaring DLL Procedures
- Calling DLL Procedures with Specific Data Types
- Converting Common Declarations

CALLDLLS.MAK

Many of the code examples in this chapter are taken from the CALLDLLS.MAK sample application. If you installed the sample applications, you will find this application in the \CALLDLLS subdirectory of the main Visual Basic directory (\VB\SAMPLES\CALLDLLS).

Overview

Because DLL procedures reside in files that are external to your Visual Basic application, you have to give Visual Basic some information so that it can find and execute the DLL procedures you want to use. You provide this information with the **Declare** statement. Once you have declared a DLL procedure, you can use it in your code like any other procedure (although you have to be especially careful about the arguments that you pass to DLL procedures).

There are two basic steps in using a DLL procedure:

1. Tell Visual Basic about the procedure by using a **Declare** statement.
2. Make the actual call.

You declare a DLL procedure only once. You then can call it any number of times.

Declaring a DLL Procedure

To declare a DLL procedure, place a **Declare** statement in the Declarations section of a form or code module. If you declare a DLL procedure in the Declarations section of a form, it is private to that form. If you declare a DLL procedure in a code module, it is public and can be called by code anywhere in your application.

If the procedure does not return a value, declare it as a **Sub** procedure. For example:

```
Declare Sub InvertRect Lib "User" (ByVal hDC As Integer,
➥ aRect As Rectangle)
```

If the procedure does return a value, declare it as a **Function** procedure. For example:

```
Declare Function GetSystemMetrics Lib "User" (ByVal n As Integer)
➥ As Integer
```

Notice the **Lib** and **ByVal** keywords in the **Declare** statement. The **Declare** statement can also contain an optional **Alias** keyword. The use of these keywords is explained in the section "Special Considerations When Declaring DLL Procedures," later in this chapter.

Note The declarations for DLL procedures are often too long to fit on a single line in this manual. When a line of code doesn't fit, a special character (➥) indicates that it continues on the next line. Even though they span several lines in this manual, these declarations must be entered as a single line in the Code window.

For More Information For the complete syntax of the **Declare** statement, see the *Language Reference*, or search Help for *Declare*.

Calling a DLL Procedure

Once a procedure is declared, you then can call it just as you would a Visual Basic statement or function. For example:

```
Sub Form_Load ()
Const SM_MOUSEPRESENT = 19
    Show
    If GetSystemMetrics(SM_MOUSEPRESENT) Then Print "Mouse installed"
End Sub
```

Important Visual Basic can't verify that you are passing correct values to a DLL procedure. If you pass incorrect values, the procedure may fail, which may cause your Visual Basic application to crash. This doesn't cause permanent harm, but you'll have to reload and restart your application. Take care when experimenting with DLL procedures, and save your work often.

Special Considerations When Declaring DLL Procedures

As you have probably realized, the declarations for DLL procedures can get fairly complex. The Visual Basic Professional Edition provides several files that make it easier to declare and call the DLL procedures in the Windows API. You will find these files in the \WINAPI subdirectory of the main Visual Basic directory.

File	Contents
WINAPI.TXT	Visual Basic DLL procedure, constant, and user-defined type declarations for the Windows API.
WIN31WH.HLP	Documentation for the entire Windows 3.1 API.
WIN31API.HLP	Visual Basic DLL procedure, constant, and user-defined type declarations for the Windows API, indexed and cross-referenced.

You can search for the declarations you want in the WIN31API.HLP Help file, copy them, and paste them into your code. Alternatively, you can load the WINAPI.TXT file in Microsoft Write or another word processor (it is too large for Notepad), copy the procedure declarations you want from this file, and paste them into your code.

Tip Don't load WINAPI.TXT into a code module. It is a very large file and will consume a lot of memory in your application. You will generally use only a handful of declarations in your code, so selectively copying the declarations you need is much more efficient.

Once you have pasted the appropriate DLL procedure declarations into your code, you simply call the procedures like any other procedure in your application. Because you have to take extra care when passing values to DLL procedures, however, you should first read the section "Calling DLL Procedures with Specific Data Types," later in this chapter.

If you are attempting to call a procedure in a DLL that is not part of the operating environment, you must determine the proper declaration for it. The rest of this section explains the syntax of the **Declare** statement in detail so that you can create the correct declaration for the procedure.

Specifying the Library

The **Lib** *libname* clause in the **Declare** statement tells Visual Basic where to find the dynamic-link library. For the operating environment DLLs, this is either "User," "GDI," "Kernel," or one of the other system DLLs such as "MMSystem." For other DLLs, the *libname* is a file specification that can include a path. For example:

```
Declare Function lzCopy& Lib "c:\win\lzexpand.dll" (ByVal S%, ByVal D%)
```

Notice that the *libname* is not case-sensitive.

Passing Arguments by Value or by Reference

By default, Visual Basic passes all arguments by reference. (When passing by reference, Visual Basic supplies a 32-bit far address.) However, many DLL procedures expect an argument to be passed by value. If you pass an argument by reference to a procedure that expects an argument passed by value, the procedure gets bad data and fails to operate properly.

To pass an argument by value, place the **ByVal** keyword in front of the argument declaration in the **Declare** statement. This ensures that each time you call the procedure, the argument is passed by value.

For example, the InvertRect procedure accepts its first argument by value and its second by reference:

```
Declare Sub InvertRect Lib "User" (ByVal hDC%, aRect As Rectangle)
```

Note When you're looking at DLL procedure documentation that uses C language syntax, remember that C passes by value all arguments except arrays.

Flexible Argument Types

Some DLL procedures can accept more than one type of data for the same argument. To pass more than one type of data, declare the argument with **As Any** to remove type restrictions. For example, you can declare a procedure like this:

```
Declare Function SendMessage Lib "User" (ByVal hWnd As Integer,
➥ ByVal msg As Integer, ByVal wp As Integer, lp As Any) As Long
```

You then can call this procedure with either a string or a long integer as its last argument:

```
Sub ScrollIt (anyText As TextBox, chars As Integer, lines As Integer)
' Scrolls contents of specified text box horizontally and vertically.
' anyText must be a text box with MultiLine = True.
' Use negative values for chars or lines to scroll left or up.
Const EM_LINESCROLL = WM_USER + 6
Dim scroll As Long, Dummy As Long
    scroll = chars * 65536 + lines
    Dummy = SendMessage(anyText.hWnd, EM_LINESCROLL, 0, ByVal scroll)
End Sub

Function FindItem (aList As ListBox, target As String) As Integer
' Returns listindex of first string matching specified target.
' For best results, the Sorted property of the list box should be True.
Const WM_USER = &H400, LB_FINDSTRING = WM_USER + 16
    FindItem = SendMessage(aList.hWnd, LB_FINDSTRING, -1, ByVal target)
End Function
```

Note that when you remove type restrictions, Visual Basic assumes the argument is passed by reference. Use **ByVal** in the actual call to the procedure (as shown in the FindItem example) to pass arguments by value. When passing strings, you must use **ByVal** to convert a Basic string into a null-terminated string. This is discussed in the section "Calling DLL Procedures with Specific Data Types," later in this chapter.

Nonstandard Names

Occasionally, a DLL procedure has a name that is not a legal identifier. It might have an invalid character (such as a hyphen), or the name might be the same as a Visual Basic reserved word (such as **Loop**). When this is the case, use the **Alias** keyword.

For example, some procedures in the operating environment DLLs begin with an underscore character. While you can use an underscore in a Visual Basic identifier, you cannot begin an identifier with an underscore. To use one of these procedures, you must declare the procedure with **Alias**:

```
Declare Function LOpen Lib "kernel" Alias "_lopen" (ByVal fn As String,
➥ ByVal f As Integer) As Integer
```

In this example, LOpen becomes the name of the procedure as it is referred to in your Visual Basic procedures. The name _lopen is the name recognized in the DLL.

You can use the **Alias** clause whenever it's convenient. For example, this **Declare** statement substitutes a shorter name (WinDir) for the full name (GetWindowsDirectory):

```
Declare Function WinDir Lib "Kernel" Alias "GetWindowsDirectory"
➥ (ByVal lpBuffer As String, ByVal nSize As Integer) As Integer
```

Now you can call the function with the shorter name:

```
Dim WinPath As String
    WinPath = String(145, Chr(0))
    WinPath = Left(WinPath, WinDir(WinPath, Len(WinPath)))
```

In addition to a name, all DLL procedures can be identified by an "ordinal number" that specifies the position of the procedure in the DLL. Some DLLs do not include the names of their procedures, and so require you to use ordinal numbers when declaring the procedures they contain. Using an ordinal number consumes less memory in your finished application and is slightly faster than identifying a procedure in a DLL by name, so you may want to use ordinal numbers when declaring other DLL procedures as well. To obtain the ordinal number of the DLL procedure you want to declare, you must consult the documentation for that DLL or use the EXEHDR utility supplied with the Microsoft C Compiler and some other compilers.

To declare a DLL procedure by ordinal number, you use the **Alias** clause with a string containing the number sign character (#) and the ordinal number of the procedure. For example, the ordinal number of the GetWinFlags function has the ordinal value 132 in the Windows kernel, so you would declare it like this:

```
Declare Function GetWinFlags Lib "kernel" Alias "#132" () As Long
```

Notice that you could specify any valid name for the procedure in this case, because Visual Basic is using the ordinal number to find the procedure in the DLL. You use the name you specified when you call the procedure:

```
Dim WFlags As Long
Const WF_80x87 = &H400&
    WFlags = GetWinFlags()
    If WinFlags And WF_80x87 Then Print "Math coprocessor available"
```

Calling DLL Procedures with Specific Data Types

Visual Basic incorporates a rich assortment of data types, including many—such as variable-length strings, **Currency**, and properties—that are not supported by the procedures in most dynamic-link libraries. Therefore, you must take care when using Visual Basic variables with DLL procedures.

Strings

The procedures in most DLLs (and all the procedures in the Windows API) expect standard C strings (sometimes called ASCIIZ strings), which end in a "null character" (binary zero). If a DLL procedure expects a null-terminated string as an argument, declare the argument as a string with the **ByVal** keyword. When used with a string argument, **ByVal** tells Visual Basic to pass the string as a null-terminated string. For example, the sndPlaySound function accepts a string that names a digitized sound (.WAV) file and plays that file (if you have Microsoft Windows 3.1 and the sound hardware and drivers that can handle these files):

```
Declare Function sndPlaySound Lib "MMSystem" (ByVal lpsound As String,
➥ ByVal flag As Integer) As Integer
```

Because the string argument for this procedure is declared with **ByVal**, Visual Basic automatically converts a variable-length string passed to this procedure into a null-terminated C string:

```
Dim SoundFile As String, worked As Integer
    SoundFile = Dir(GetWindowsDirectory() & " \" & "*.wav")
    worked = sndPlaySound(SoundFile, 1)
```

Some DLLs may be written specifically to work with Visual Basic. A DLL can be written to use standard Visual Basic strings, using procedures that Visual Basic supplies for this purpose. (These procedures are documented in the *Control Development Kit* in the *Professional Features* manual.) If a DLL procedure expects a Visual Basic string as an argument, you do not need to declare the argument with the **ByVal** keyword. In addition, some of the DLL procedures written for Visual Basic may act as functions that return strings. A DLL procedure cannot act as a function that returns strings unless it was written specifically for use with Visual Basic. A DLL written specifically for Visual Basic probably supplies a file containing the correct Visual Basic declarations for the procedures in the DLL. Consult the documentation for the DLL for details.

DLL Procedures That Modify Strings

Strings are always passed to DLL procedures by reference (remember, the **ByVal** keyword for string arguments specifies that Visual Basic should convert it to a null-terminated string, not that the string should be passed by value), so a DLL procedure can modify a Visual Basic string variable it receives as an argument. However, be careful when calling a DLL procedure that modifies a string. A DLL cannot increase the length of a Visual Basic string; it simply writes beyond the end of the string if it is not long enough. This corrupts other areas of memory. You can avoid this problem by making the string argument long enough that the DLL procedure can never write past the end of it.

For example, the GetWindowsDirectory procedure returns the path for the Windows directory in its first argument:

```
Declare Function GetWindowsDirectory Lib "Kernel"
➥ (ByVal lpBuffer As String, ByVal nSize As Integer) As Integer
```

A safe way to call this procedure is to first make the returned argument at least 255 characters long by filling it with characters—in this case, null (ANSI zero) characters:

```
Path = String(255, 0)

worked = GetWindowsDirectory(Path, Len(Path))
```

Another solution is to define the string as fixed length:

```
Dim Path As String * 255
worked = GetWindowsDirectory(Path, Len(Path))
```

When passing a fixed-length string to a DLL procedure for an argument declared **ByVal**, Visual Basic always converts fixed-length string arguments to null-terminated variable-length strings. So both of these processes amount to the same thing: creating a variable-length string long enough to contain the longest possible string that the procedure might return.

Note The operating environment DLL procedures generally do not return strings longer than 255 characters. While this is true of many other libraries, always consult the documentation for the procedure in question.

Visual Basic strings can be used when the DLL procedure calls for a memory buffer. Use one of the processes outlined in this section to ensure that the string is long enough to accept whatever data the procedure supplies.

Arrays

You can pass individual elements of an array the same way you pass any variable of the same type as the base type of the array. For example, you can use the sndPlaySound procedure declared earlier to play a series of .WAV files stored in an array:

```
Dim i As Integer, worked As Integer
    For i = 0 to UBound(WaveFiles)
        worked = sndPlaySound(WaveFile(i), 0)
    Next i
```

Sometimes you may want to pass an entire array to a DLL procedure. If the DLL procedure was written especially for Visual Basic, then you may be able to pass an array to it the same way you pass an array to a Visual Basic procedure: by passing the array with empty parentheses. Because Visual Basic arrays are special data structures, the DLL procedure must be written with Visual Basic in mind if it is to use a Visual Basic array. (Consult the documentation for the DLL to see if this is the case.)

If the DLL procedure doesn't accept Visual Basic arrays directly, you can still pass an entire array if it is a numeric array. You pass an entire numeric array by passing the first element of the array by reference. This works because numeric array data is always laid out sequentially in memory. A DLL procedure, if given the first element of an array, has access to all its elements. For example, there are internal tab stops in multiline (but not single-line) text box controls: If the text in the text box contains tab characters (character code 9), the text following the tab character is aligned at the next tab stop. You can set the position of these tab stops by calling the SendMessage function in the Windows API and passing an array that contains the new tab stop settings:

```
Declare Function SendMessage Lib "User" (ByVal hWnd As Integer,
➥ ByVal wMsg As Integer, ByVal wParam As Integer, lParam As Any) As Long
Const WM_USER = &H400, EM_SETTABSTOPS = WM_USER + 27

Sub ChangeTabs (anyText As TextBox, tabcount As Integer)
Dim dummy As Long, i As Integer
ReDim tabs(tabcount - 1)
    For i = 0 To UBound(tabs)
        tabs(i) = (i + 1) * 96  ' Tab stops measured in "dialog units."
    Next i
    ' Call with null pointer to empty existing tab stops.
    dummy = SendMessage(anyText.hWnd, EM_SETTABSTOPS, 0, ByVal 0&)
    ' Pass first element in array; other elements follow it in memory.
    dummy = SendMessage(anyText.hWnd, EM_SETTABSTOPS, tabcount, tabs(0))
    anyText.Refresh
End Sub
```

You cannot do this with string arrays, however. If a DLL procedure attempts to access memory beyond the end of the first element in a string array, it may corrupt memory or cause an error. Unless the DLL procedure was written with Visual Basic in mind (in which case you can pass the string array in the usual way, but passing it with empty parentheses), you cannot pass a string array to it.

Note If you pass the first element of a huge array (an array larger than 64K) to a DLL procedure, it will be able to access only the first 64K of data. Only DLL procedures written specifically for use with Visual Basic can access all of the data in a huge array.

User-Defined Types

Some DLL procedures take user-defined types as arguments. These are referred to as "structures" in C and as "records" in Pascal. DLL documentation often uses the C terminology.

As with arrays, you can pass the individual elements of a user-defined type the same way you would pass ordinary numeric or string variables.

You can pass an entire user-defined type as a single argument if you pass it by reference. User-defined types cannot be passed by value. Visual Basic passes the address of the first element, and the rest of the elements of a user-defined type are packed in memory following the first element. For example, several procedures in the operating environment DLLs accept a user-defined type with this structure (you can place this in the Declarations section of any code module):

```
Type Rectangle
    Left As Integer
    Top As Integer
    Right As Integer
    Bottom As Integer
End Type
```

Two of the procedures that accept a rectangle are DrawFocusRect, which draws a dotted outline around the specified rectangle, and InvertRect, which inverts the colors of the specified rectangle. To use the procedures, place these declarations in the Declarations section of a form or code module:

```
Declare Sub DrawFocusRect Lib "User" (ByVal hDC As Integer,
➥ aRect As Rectangle)
Declare Sub InvertRect Lib "User" (ByVal hDC As Integer,
➥ aRect As Rectangle)

Dim MouseRect As Rectangle
```

Now you can use the following **Sub** procedures to call the two DLL procedures—one to display a dotted line around a rectangle as you drag the mouse, and one to invert the colors of the rectangle when you release the mouse button:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    ScaleMode = 3
    If Button And 1 Then
        MouseRect.Left = X
        MouseRect.Top = Y
        MouseRect.Right = X
        MouseRect.Bottom = Y
    End If
End Sub

Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    ScaleMode = 3
    If Not (Button And 1) Then
        MouseRect.Right = X
        MouseRect.Bottom = Y
        InvertRect hDC, MouseRect
    End If
End Sub

Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single,
➥ Y As Single)
    ScaleMode = 3
    If Button And 1 Then
        DrawFocusRect hDC, MouseRect
        MouseRect.Right = X
        MouseRect.Bottom = Y
        DrawFocusRect hDC, MouseRect
    End If
End Sub
```

Notice that these procedures expect the values for the rectangle to be expressed in pixels, so you should set the ScaleMode property of the form to 3-Pixel at design time.

Most procedures that accept user-defined types do not expect them to contain string data. However, you can pass a user-defined type that includes string elements to a DLL procedure. If the string elements are fixed-length strings, they are packed in memory like any other value. However, variable-length strings are incorporated in a user-defined type as string descriptors, a Visual Basic data structure that takes four bytes for each variable-length string element. Unless a DLL procedure is written specifically for Visual Basic, it cannot use a user-defined type that contains variable-length strings.

Null Pointers

Some DLL procedures occasionally expect to receive a null pointer as an argument. If you need to pass a null pointer to a procedure, declare the argument with **As Any** (as described earlier in the section "Flexible Argument Types") and pass the expression `ByVal 0&`.

For example, the FindWindow procedure accepts two string arguments, so you would expect to declare it like this:

```
Declare Function FindWindow Lib "User" (ByVal Class As String,
➥ ByVal Cap As String) As Integer
```

However, FindWindow also accepts a null pointer for either or both of its arguments. If you declare it in this way, you cannot pass null pointers to it. Passing a zero-length string ("") does not work; this passes a pointer to a null string, not a null pointer.

The solution is to declare the procedure like this:

```
Declare Function FindWindow Lib "User" (Class As Any, Cap As Any)
➥ As Integer
```

Now you can still call the procedure and pass it two strings. But you can also pass a null pointer as one of the arguments if you want:

```
hWndExcel = FindWindow(ByVal 0&, ByVal "Microsoft Excel")
```

Notice the use of **ByVal** with both arguments. You must use **ByVal** when passing a string because Visual Basic passes arguments declared **As Any** by reference, and does not convert Basic strings to null-terminated strings. Including **ByVal** when passing a string declared **As Any** causes Visual Basic to convert the string to the null-terminated form expected by most DLL procedures.

Similarly, you use **ByVal** when passing the null pointer; this ensures that the routine gets zero (a null pointer) instead of a pointer to zero. Notice the ampersand character (&) after the zero. This informs Visual Basic that you are passing a **Long** integer (32-bit) zero. This ensures that a null pointer of the right size is passed; pointers in Visual Basic are always far (32-bit) pointers.

You can use this technique to pass a null pointer to any procedure that accepts an argument by reference.

Handles

The operating environment DLL procedures make extensive use of handles—handles to Windows (hWnd), handles to Device Contexts (hDC), and so on. A *handle* is a unique integer value defined by the operating environment and used to refer to objects such as forms or controls. When a procedure takes a handle as an argument, always declare it as a **ByVal Integer**. DLL functions that return a handle can be declared as **Integer** functions. Handles are ID numbers, not pointers or numeric values. You should never perform mathematical operations on them.

The hWnd property of forms and nongraphical controls and the hDC property of forms and picture box controls supply valid handles that you can pass to DLL procedures. Like any other property passed to a DLL procedure, they can be passed only by value.

Properties

Properties must be passed by value. If the corresponding argument is declared with **ByVal**, then you can pass the property directly. For example, you can determine the dimensions of the screen or printer in pixels with this procedure:

```
Declare Function GetDeviceCaps% Lib "GDI" (ByVal hDC%, ByVal nIndex%)
```

You can pass the hDC property of a form or the Printer object to this procedure to obtain the number of colors supported by the screen or the currently selected printer. For example:

```
Sub Form_Click ()
Const PLANES = 14, BITS = 12
    Print "Screen colors ";
    Print GetDeviceCaps(hDC, PLANES)* 2 ^ GetDeviceCaps(hDC, BITS)
    Print "Printer colors ";
    Print GetDeviceCaps(Printer.hDC, PLANES) *
    ➥ 2 ^ GetDeviceCaps(Printer.hDC, BITS)
End Sub
```

To pass a property by reference, you must use an intermediate variable. All strings are passed by reference; therefore, to pass a string property to a DLL procedure, you must first assign the property to a string variable and then pass the variable to the procedure. For example, suppose you want to use the GetWindowsDirectory procedure described earlier to set the Path property of a file list box control. You cannot do this:

```
worked = GetWindowsDirectory(File1.Path, Len(File1.Path))
```

Instead, use the following code to pass a property to this procedure:

```
Dim Temp As String, worked As Integer
Temp = String(255, 0)
worked = GetWindowsDirectory(Temp, Len(Temp))
File1.Path = Temp
```

Use this technique with numeric properties as well if you want to pass them to DLL procedures that accept an argument by reference.

Object Variables

You cannot pass a form or control variable to a DLL procedure unless the DLL has been written specifically for Visual Basic. Object variables are actually complex data structures, and DLL procedures that don't take this into account—such as the procedures in the operating environment DLLs—cannot make use of forms or controls passed to them. Likewise, you cannot pass any of the special objects (App, Screen, Clipboard, Printer, or Debug) to a DLL procedure.

Converting Common Declarations

The procedures in DLLs are most commonly documented using C language syntax. To call them from Visual Basic, you must translate them into valid **Declare** statements and call them correctly. When performing this translation, you may find Table 22.1 useful. It lists common C language declarations and their Visual Basic equivalents.

Table 22.1 Common C Language Declarations and Visual Basic Equivalents

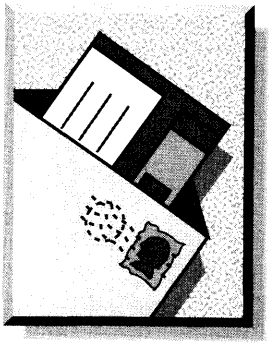
C language declaration	In Visual Basic declare as	Call with
Pointer to a string (LPSTR)	**ByVal** S **As String**	Any **String** or **Variant** variable
Pointer to an integer (LPINT)	I **As Integer**	Any **Integer** or **Variant** variable
Pointer to a long integer (LPDWORD)	L **As Long**	Any **Long** or **Variant** variable
Pointer to a structure (for example, LPRECT)	S **As** Rect	Any variable of that user-defined type
Integer (INT, UINT, WORD, BOOL)	**ByVal** I **As Integer**	Any **Integer** or **Variant** variable
Handle (hWnd, hDC, hMenu, and so on)	**ByVal** h **As Integer**	Any **Integer** or **Variant** variable
Long (DWORD, LONG)	**ByVal** L **As Long**	Any **Long** or **Variant** variable

Table 22.1 Common C Language Declarations and Visual Basic Equivalents *(cont'd)*

C language declaration	In Visual Basic declare as	Call with
Pointer to an array of integers	I **As Integer**	The first element of the array, such as I(0)
Pointer to a void (void *)	**As Any**	Any variable (use **ByVal** when passing a string)
Void (function return value)	**Sub** procedure	n/a
NULL	**As Any**	**ByVal** 0&

CHAPTER 25

Distributing Your Applications



Once you have created a Visual Basic application, you may want to distribute it to others. You can freely distribute any application you create with Visual Basic to anyone who owns Microsoft Windows. If you are going to distribute your application to more than a few individuals, or are planning to sell your application commercially, then you will want to write a setup program that installs your application on a customer's machine. This chapter shows you how to create a professional setup program for your application.

Contents

- The Visual Basic SetupWizard
- How to Use the Setup Toolkit

SETUP1.MAK

SETUP1.MAK is a sample setup program that installs the LOAN.EXE sample application on a user's disk. The source code for SETUP1.MAK is in the \SETUPKIT\SETUP1 subdirectory of the main Visual Basic directory. If you installed the sample applications, you will find the source code for LOAN.EXE in the \GRID subdirectory of the main Visual Basic directory (\VB\SAMPLES\GRID).

The Visual Basic SetupWizard

The Visual Basic SetupWizard greatly reduces the number of steps required to create distribution disks for your Visual Basic application. Based on information you provide, the SetupWizard:

- Creates a setup program (SETUP1.EXE).
- Builds your application's executable file.
- Compresses all files and assign them to a disk layout.
- Notifies you of the number of blank formatted disks needed.

- Copies distribution files to a blank formatted disk and prompts you to insert additional disks as needed.
- Notifies you when the master distribution disks are completed.

Note SetupWizard creates setup programs and distribution disks for Visual Basic applications. To create setup programs for other Microsoft Windows applications, use the Setup ToolKit provided in the Microsoft Windows SDK.

Creating Distribution Disks using the SetupWizard

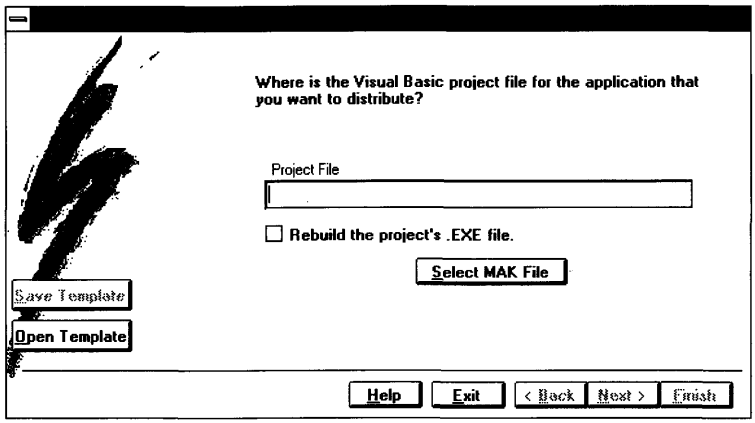
To utilize all the features of the SetupWizard, you must first save all the form and code modules in your application as ASCII text. To do this, select a form or code module in the project window, and choose Save File As on the File menu. Then check the Save As Text check box in the displayed dialog.

▶ To create distribution disks using the SetupWizard

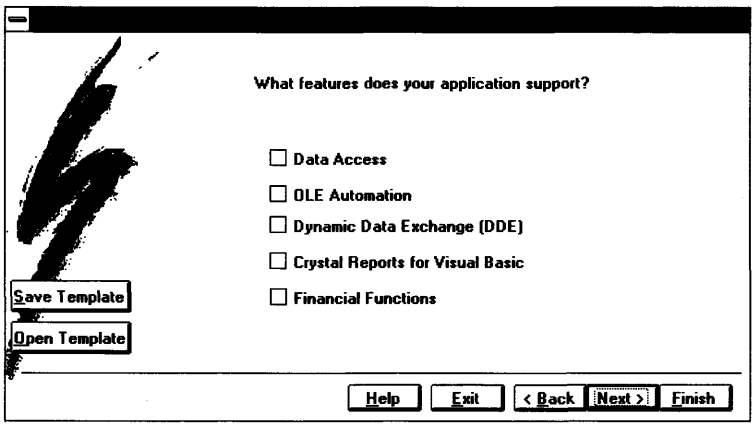
1. Double-click the SetupWizard icon in the Visual Basic program group.
2. Enter the path and file name for the .MAK file of the application you want to distribute in the Project File text box.

   To browse the files on your disk, click the Select MAK File command button in the opening screen. When you have entered the name of the application to distribute, click the Next button to continue.

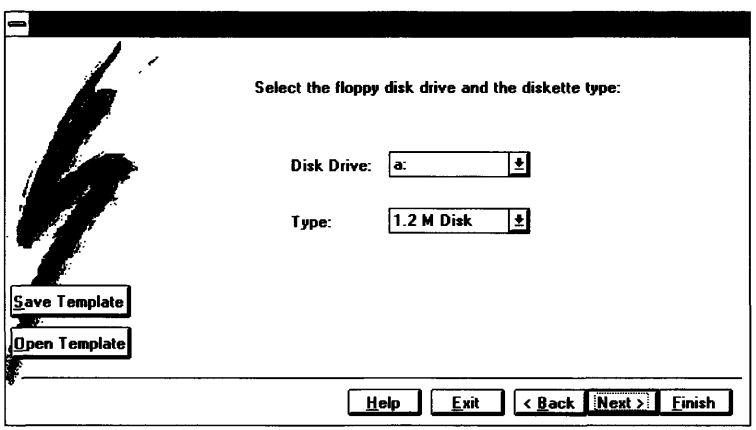
   If an executable (.EXE) does not exist for the specified application, the SetupWizard automatically builds the application's executable file. To rebuild the application's executable file, check the Rebuild check box.



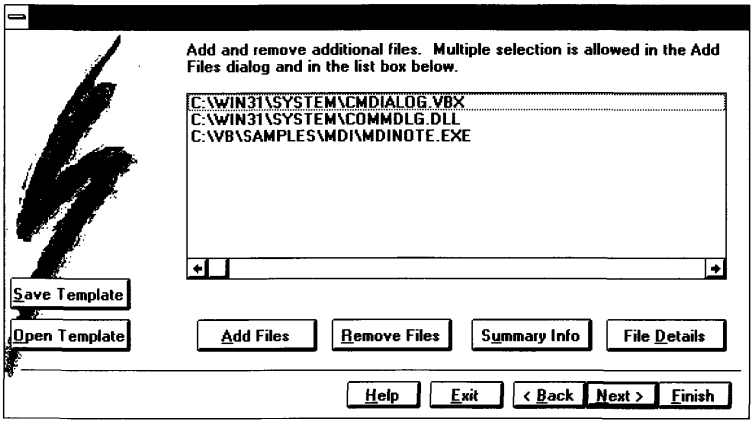
3. The SetupWizard searches your .MAK file to determine the files that need to be distributed. However, there are some files it cannot detect. When the following form is displayed, select the features that apply to your application, then click the Next button.



4. Select the disk drive and type of distribution disks you want to create. Click the Next button to continue.



5. Examine the list of files.



This form lists all the files SetupWizard has detected your application requires. If desired, you add or remove files from this list using the Add Files and Remove Files command buttons. Table 25.1 shows the files (.DLLs, .VBXs, etc.) required by certain types of Visual Basic applications.

Note Only list the files required by your application on this form. You do not need to list any files required by SETUP1.EXE (SETUP.LST, for example). The SetupWizard does this automatically.

Click the Next button to continue. The SetupWizard compresses all appropriate files, builds SETUP1.EXE, copies all files to the specified disk drive, and prompts for new disks as needed.

See the section "Determining the Files You Need to Distribute" later in this chapter for details on dynamic-link libraries (DLLs) and other files you may need to distribute with your application.

For More Information For information on the SetupWizard, see the online Help file SETUPWIZ.HLP.

How to Use the Visual Basic Setup Toolkit

Altougth the SetupWizard is designed to handle most setup scenarios a Visual Basic developer will encounter, some situations require that you to create a customized setup program.

If you want to write a setup program that has features not provided by the SetupWizard, such as using your own compression utility, or providing the user with the option of installing different portions of an application, you can do so by writing your own setup program using the Setup Toolkit. All the tools required to use the Setup Toolkit are located in the \SETUPKIT subdirectory of the main Visual Basic directory.

The Setup Toolkit allows you to:

- Compress and expand your application's files to fill the available disk space.
- Pre-install files required by your setup program on the customer's machine.
- Create a customized look and feel for your setup program.
- Install files into the appropriate Windows directories (\WINDOWS, \WINDOWS\SYSTEM) regardless of the actual directory names on your customer's system.
- Install your application's files on your customer's machine.
- Use the version stamping resources (VER.DLL) in Windows to determine whether a file should be copied to the customer's machine.
- Create Windows Program Manager groups and icons for your application.

The Setup Toolkit consists of utilities, DLLs, and a sample setup program written in Visual Basic. The following list briefly describes each file.

File name	Description
SETUP.EXE	The program that your customer runs to install your application. This is a pre-installation program sometimes referred to as a *bootstrap* program. SETUP.EXE installs VER.DLL and all the files listed in SETUP.LST on the customer's machine, and then invokes your custom setup program (SETUP1.EXE, or another name you choose). SETUP.EXE copies SETUP1.EXE into your customer's \WINDOWS directory and then invokes SETUP1.EXE to install your software.
SETUP1.EXE	A Visual Basic application that you customize. The source files for this program are located in the \SETUPKIT\SETUP1 subdirectory of the main Visual Basic directory. Use these source files as a template for creating your own setup program.

File name	Description
SETUP.LST	A text file that lists all the files SETUP.EXE pre-installs on your customer's machine. The first line of this file must be the name of your setup program (SETUP1.EXE, for example).
SETUP1.BAS	A collection of Visual Basic functions. These functions are used in SETUP1.MAK to perform the tasks required by a setup program.
SETUPKIT.DLL	A DLL that contains various functions used in SETUP1.BAS.
VER.DLL	A DLL that contains functions for copying and decompressing files. VER.DLL can also detect the version of a file that has a Microsoft Windows version stamp. You cannot compile version information in your Visual Basic .EXE file; however, some files do contain this version stamp. If a file does not have a Microsoft Windows version stamp, VER.DLL uses the file date and time to determine the file's version.

How to Use the Setup Toolkit

You create a professional setup program for your application using the following procedure. When you have finished this procedure, you will have a set of *distribution disks* that a customer can use to install your program.

▶ **To create distribution disks for your application**

1. List all the files you want to distribute.
2. Modify SETUP.LST to include all the files on your list.
3. Compress the appropriate files.
4. Determine the layout of your distribution disks (which files go on which disks).
5. Determine where to install the files on your customer's machine.
6. Write your setup program.
7. Create the distribution disks.
8. Test your distribution disks.

The following sections of this chapter describe these steps in detail.

Determining the Files You Need to Distribute

Before creating a setup program, you need to determine which files will be included on the distribution disks. Remember that the setup program requires the following files: SETUP.EXE, SETUP1.EXE, SETUP.LST, VER.DLL, and VBRUN300.DLL.

Note If you are using the SetupWizard, you do not need to include the above files in the Distribution Files list box.

The user will need other files to run your application. In addition to an executable file (.EXE), your application may require data files, custom controls (.VBX), or other files such as DLLs. The additional files your application may require include the following.

File name	Description
YOURAPP.EXE	Your application's .EXE file.
YOURAPP.DAT (optional) YOURAPP.TXT (optional) YOURAPP.INI (optional)	Any data files (.DAT), text files (.TXT), or initialization files (.INI) that your application requires.
YOURCC.VBX (optional)	Any custom controls required by your application.
YOURDLL.DLL (optional)	Any other DLLs required by your application. See the following table for a list of DLLs your application may require.

Note Visual Basic does not compile custom controls and DLLs as part of your application's .EXE file. If your application uses any custom controls or DLLs, you must include the appropriate .VBX and .DLL files on the distribution disks.

Depending on the operations your application performs and the custom controls it uses, you may also need to distribute the following DLLs.

Table 25.1 Required DLLs for Distributing Visual Basic Applications

DLL name	Required by	Description
COMPOBJ.DLL	OLE control OLE Automation	Required by OLE 2.0.
MSOLE2.VBX MSOLEVBX.DLL	OLE control	Required by the OLE custom control. Note that applications using the OLE control, or OLE Automation require Microsoft Windows version 3.1 or later.
OLE2.DLL	OLE control OLE Automation	Required by OLE 2.0.
OLE2.REG	OLE control OLE Automation	Used to register OLE 2.0 and OLE Automation in the user's REG.DAT file.
OLE2CONV.DLL	OLE control OLE Automation	Required by OLE 2.0.
OLE2DISP.DLL	OLE control OLE Automation	Required when performing OLE Automation tasks or accessing the OLE control's Object property.
OLE2NLS.DLL	OLE control OLE Automation	Facilitates string comparisons based on the user's national language.
OLE2PROX.DLL	OLE control OLE Automation	Required by OLE 2.0.
STORAGE.DLL	OLE control OLE Automation	Required by OLE 2.0.
VBOA300.DLL	OLE control OLE Automation	Required by Visual Basic and OLE 2.0.
GRID.VBX	Grid control	The grid custom control.

Table 25.1 Required DLLs for Distributing Visual Basic Applications *(continued)*

DLL name	Required by	Description
DDEML.DLL	DDE	Required if your application uses dynamic data exchange (DDE).
CMDIALOG.VBX	Common dialog control	The common dialog custom control.
COMMDLG.DLL	Common dialog control	Required by the common dialog custom control.
MSAFINX.DLL	Financial functions	Required if your application calls any of the Visual Basic financial functions. For a list of the financial functions, search Help for "Financial Functions."
VBDB300.DLL MSAES110.DLL MSAJT110.DLL	Data access	Required if your application uses the data control or the data access features of the Professional Edition.
BTRV110.DLL	Data access	Required if your application accesses data in Btrieve format.
PDX110.DLL	Data access	Required if your application accesses data in Paradox format.
XBS110.DLL	Data access	Required if your application accesses data in xBase format.
COMMDLG.DLL CTL3D.DLL DBNMP3.DLL OBDC.DLL OBDCADMIN.EXE ODBCINST.DLL ODBCINST.HLP	ODBC	(Professional Edition only) Required if your application uses Open Database Connectivity (ODBC).
SQLSRVR.DLL DBNMP3.DLL INSTCAT.SQL INSTCAT.48 DRVSSRVR.HLP	ODBC	(Professional Edition only) Required if your application uses ODBC to access SQL Server data.

Table 25.1 Required DLLs for Distributing Visual Basic Applications *(continued)*

DLL name	Required by	Description
SQORA.DLL ORASETUP.DLL ORA6WIN.DLL DRVORACL.HLP ORACLE.TXT PDSODBC.DLL	ODBC	(Professional Edition only) Required if your application uses ODBC to access Oracle data.
CRYSTAL.VBX CRPE.DLL CRXLATE.DLL COMMDLG.DLL PDIRJET.DLL PDBJET.DLL MSAJT110.DLL MSAES110.DLL PDSODBC.DLL	Crystal Reports	(Professional Edition only) Required by the Crystal Reports for Visual Basic custom control.

Caution If you are developing an application that requires Btrieve, you must have a licensing agreement with Novell to release the Btrieve for Windows driver (WBTRCALL.DLL) with your application. Applications that use Btrieve require both the Btrieve for Windows driver and the Btrieve ISAM dynamic-link library (BTRV110.DLL).

SETUP.LST

SETUP.LST is a text file that lists all the files SETUP.EXE pre-installs on your customer's machine. After SETUP.EXE installs the files specified in SETUP.LST, it invokes your setup program. You use SETUP.LST to specify the files that must be installed in order for SETUP1.EXE to run. You use SETUP1.EXE to install the remaining files (the files not required by SETUP1.EXE itself).

The name of your setup program must be the first line in SETUP.LST. You can give your setup program any name except "SETUP.EXE," which is the name of the pre-installation program. For example, you can name your setup program SETUP1.EXE, SETLOAN.EXE, or MYSETUP.EXE.

If you use SETUP1.MAK as a template for your setup program, but your setup program does not make calls to any additional DLLs or use any custom controls, then SETUP.LST contains only the name of your setup program, VBRUN300.DLL, and SETUPKIT.DLL. Each file name in SETUP.LST must occupy its own line.

In some cases, you may want to access DLLs or use custom controls in your setup program that are not already used by SETUP1.MAK. If so, you must include the names of these files in SETUP.LST.

Compressing the Setup Files

Once you have determined which files need to be included on the distribution disks, you may discover that all of them cannot fit on a single floppy disk. By compressing the files on your distribution disks, you may be able to reduce the number of disks required to distribute your application.

The Setup Toolkit provides a utility that allows you to compress files. It is called COMPRESS.EXE and is located in the \SETUPKIT\KITFILES subdirectory of the main Visual Basic directory. COMPRESS.EXE is an MS-DOS–based utility, not a Microsoft Windows–based application.

To compress a file

- At the MS-DOS prompt, type:

  compress -r *filename*

  The -r switch automatically replaces the last character of the compressed file with an underscore. The compressed file is encoded so that it is automatically renamed to the original file name when it is decompressed. For example, to compress the file LOAN.EXE, you type:

  compress -r LOAN.EXE

  The result is a compressed file named LOAN.EX_. When decompresssed, the file is renamed LOAN.EXE.

Note The -r switch is required when compressing files that are installed using SETUP.EXE or SETUP1.EXE.

You can compress any file on the distribution disks except SETUP.EXE, SETUP.LST, and VER.DLL. You can compress as many files as you want. VER.DLL recognizes when a file has been compressed and copies files to the user's disk correctly whether or not they have been compressed, as long as you use version 2.0 of Microsoft's COMPRESS.EXE, which comes with the Setup Toolkit. Compressed files automatically return to their original name when expanded on the user's disk.

For More Information For information on COMPRESS.EXE, see the file COMPRESS.TXT in the \SETUPKIT\KITFILES subdirectory of the main Visual Basic directory.

Determining the Layout of the Distribution Disks

Before you can write the setup program, you need to know the name of each file and the distribution disk on which it is located, so it's a good idea to map out the contents of each distribution disk before writing SETUP1.EXE.

The first distribution disk must contain the following files:

- SETUP.EXE
- SETUP1.EXE (or whatever you have named your setup program)
- SETUP.LST
- VER.DLL
- SETUPKIT.DLL
- VBRUN300.DLL
- Any other files listed in SETUP.LST (the files that will be pre-installed)

By calculating the size of the remaining files, you can determine which files each disk should contain.

Note The Setup Toolkit does not allow you to split a file between disks.

Determining Where to Install Files on the User's Machine

Each Visual Basic application requires at least two files—the application's .EXE file and the run-time file, VBRUN300.DLL. It is very likely that your application requires additional files. Before writing your setup program, you must determine where to install each of these files on the user's disk. The files required by your application can be divided into several *classes*:

- Program files
- Initialization files
- Operating system components
- Shared application resources

Suggested locations for each of these classes are described in the following sections.

Program Files

These files are essential for your application to run and are useful only in the context of your application. For example, the application's .EXE file or any data files that your application may require are considered program files. Program files should be installed in the *application directory*. The user is prompted for this directory during setup (sample code for this is in SETUP1.MAK).

Initialization Files

These files control options for the user of an application. Initialization (.INI) files should be installed in the main Microsoft Windows directory on the user's disk. This is the directory in which the WIN.INI file is located (usually in the \WINDOWS subdirectory). You determine the name of this directory on the user's disk by calling the GetWindowsDirectory() Windows API function. The code in SETUP1.MAK shows how to call this function.

Operating System Components

These files are normally included with Microsoft Windows, but you may want to distribute them with your application. For example, you may want to ensure that the user has a particular version of a DLL, such as OLE2.DLL or SHELL.DLL, by shipping it with your application. Files such as these should be installed in the \SYSTEM subdirectory of the main Microsoft Windows directory. You determine the name of this directory on the user's disk by calling the GetSystemDirectory() Windows API function. The code in SETUP1.MAK shows how to call this function.

Note When installing a file on the user's machine, it is imperative that you do not copy an older version of the file over a new version. The CopyFile function in SETUP1.BAS uses the VerInstall API to copy files to the user's machine. If a file already exists on the user's machine, the CopyFile function displays a dialog that prompts the user to rename the older file.

Shared Application Resources

Application resources may be shared by more than one application. For example, several different vendors may ship applications that use the same custom control. By installing .VBX files in the \WINDOWS\SYSTEM directory, you can ensure that all applications use the most current .VBX file.

Writing Your Setup Program

The quickest way to write your setup program is to modify the source code of the sample included with the Setup Toolkit, SETUP1.MAK. This sample setup program is located in the \SETUPKIT\SETUP1 subdirectory of the main Visual Basic directory. SETUP1.MAK installs the sample application LOAN.EXE, which is located in the \SAMPLES\GRID subdirectory of the main Visual Basic directory.

LOAN.EXE is a small sample application that uses the grid custom control. In order to run correctly, LOAN.EXE requires that GRID.VBX be properly installed on the user's system. This is a typical setup scenario for applications that use custom controls.

To customize the source code of SETUP1.MAK so that it installs your application instead of LOAN.EXE, you need to modify only the code in the Declarations section and the Form_Load event procedure of the SETUP1.FRM module. Use the following general procedure.

▶ **To customize SETUP1.MAK**

1. Modify the appropriate constants in the Declarations section of SETUP1.FRM.
2. In the Form_Load procedure of SETUP1.FRM, modify the arguments to the **PromptForNextDisk** function call.
3. Call the **CopyFile** function for each file on the current distribution disk.
4. Repeat steps 2–3 for each of the distribution disks, creating a new block of **PromptForNextDisk** and **CopyFile** calls for each disk.
5. Modify the arguments to the **CreateProgManGroup** and **CreateProgManItem** procedure calls.

Specific information on each of these steps can be found later in this chapter.

SETUP1.FRM Constants

Several useful constants are defined in the Declarations section of SETUP1.FRM. When customizing your setup program, redefine these constants to reflect information about your application.

Constant	Description
APPNAME	The name of your application. This constant is used to display the name of your application in the captions of forms and in messages throughout the setup program.
APPDIR	The default directory in which your application is to be installed on the user's disk.
WINSYSNEEDED	The total size, in bytes, of all the files (uncompressed) that the setup program will install in the user's \WINDOW and \WINDOWS\SYSTEM subdirectories. This number is used to determine whether the computer has enough disk space (on the drive that contains these directories) to install your application.
OTHERNEEDED	The total size, in bytes, of all the files (uncompressed) that the setup program will install in the application directory (the directory in which the user has chosen to install your application). This number is used to determine whether the computer has enough disk space to install your application (on the drive that contains the application directory).

This example of defining constants is excerpted from the Declarations section of SETUP1.FRM:

```
Const APPNAME = "Loan Application"
Const APPDIR = "C:\LOAN"
Const WINSYSNEEDED = 40896
Const OTHERNEEDED = 12555
```

PromptForNextDisk and CopyFile Functions

Use the **PromptForNextDisk** function to prompt the user to enter a disk. If the user enters the correct disk, the function returns **True**. Use the **CopyFile** function to copy a file from the distribution disk to the user's machine. You must call **CopyFile** once for each file on the disk. After copying all the files on a disk, you call **PromptForNextDisk** again, to prompt the user to insert the next disk.

Example

This example of the **PromptForNextDisk** and **CopyFile** functions is excerpted from the Form_Load event procedure of SETUP1.FRM:

```
' -----------
' Copy Files
' -----------
' Disk 1
' -----------
' Test to see if LOAN.EX_ is on the disk; if not, then you know the
' user did not insert the first disk.
If Not PromptForNextDisk(1, SourcePath$ + "LOAN.EX_") Then
    GoTo ErrorSetup
End If
' Install LOAN.EXE in the destPath$.
If Not CopyFile(SourcePath$, destPath$, "LOAN.EX_", "LOAN.EXE", 0) Then
    GoTo ErrorSetup
End If
' Install GRID.VBX in the user's \SYSTEM directory.
If Not CopyFile(SourcePath$, winSysDir$, "GRID.VB_", "GRID.VBX", 0) Then
    GoTo ErrorSetup
End If

' -----------
' Disk 2
' -----------
' Test to see if FOO.DA_ is on the disk; if not, then you know the
' user did not insert the second disk.
If Not PromptForNextDisk(2, SourcePath$ + "FOO.DA_") Then
    GoTo ErrorSetup
End If
' Install FOO.DAT in the destPath$.
If Not CopyFile(SourcePath$, destPath$, "FOO.DA_", "FOO.DAT", 0)  Then
    GoTo ErrorSetup
End If
```

CreateProgManGroup and CreateProgManItem Functions

These functions create a Windows program group and program item for your application.

This example of the **CreateProgManGroup** and **CreateProgManItem** functions is excerpted from the Form_Load event procedure of SETUP1.FRM:

```
' -------------------------------------
' Create Program Manager group and icon.
' -------------------------------------
CreateProgManGroup Setup1, "Loan Application", "LOAN.GRP"
CreateProgManItem Setup1, destPath$ + "LOAN.EXE", "Loan Application"
```

Creating the Distribution Disks

Before copying any files to the distribution disks, it is a good idea to label them. On the first disk, attach a label that reads:

```
Disk1: Setup Disk
Insert this disk in drive A
In the Program Manager, choose Run from the File menu
Type a:setup and press Enter
```

This label provides all the instructions your customer needs to install your application. You can label the remaining disks in a similar fashion. For example:

```
Disk2: MyApp
```

Now you are ready to copy your files to the appropriate distribution disks.

▶ **To create distribution disks**

1. Rename VER.DLL to VER.DL_ by typing:

   rename a:ver.dll a:ver.dl_

   You must change the VER.DLL file extension to .DL_ so that Microsoft Windows does not try to use it while the setup disk is in the floppy drive.

2. Copy these files onto the first disk:
   - SETUP.EXE
   - SETUP.LST
   - SETUP1.EXE (or whatever you have named your setup program)
   - SETUPKIT.DLL
   - VBRUN300.DLL
   - VER.DL_

   If you have compressed any of these files, the last character will be an underscore (for example, VBRUN300.DL_).

3. Copy the remaining files as specified by your disk layout.

Testing Your Distribution Disks

Once you have created your distribution disks, you should test them. Be sure to test your setup program on a machine that does not have Visual Basic or any of the custom controls required by your application.

▶ **To test your setup program using the distribution disks**

1. Insert the first disk in drive A.
2. From the Microsoft Windows Program Manager, choose Run from the File menu and type:

   a:setup

If your application does not install properly, check to make sure that the actual layout of the distribution disks matches the code in your setup program.

APPENDIX A

ASCII File Formats

Microsoft Visual Basic supports ASCII file formats for forms and project files. This appendix describes these file formats.

ASCII Forms Format

Visual Basic saves forms in one of two formats: binary (the default) and ASCII. Use ASCII format when you need to work with your forms in tools such as source-control tools, automated form generators, and source-browsing tools.

Saving Forms in ASCII Format

▶ **To save a form in ASCII format**

1. From the File menu, choose Save File As.
2. Select the Save As Text check box.

Note Using the Save Text command on the File menu does *not* produce an ASCII form. The Save Text command on the File menu saves *only the code in the currently selected form or module* in an ASCII file. The text describing the properties of the form and its controls is not included in an ASCII file created with the Save Text command.

Once you save or load a form in ASCII format, Visual Basic continues to save the form in ASCII format whenever you save the form, unless you clear the Save As Text check box.

Setting ASCII as the Default Format

You can also select ASCII forms as the default format for saving new files by setting the Default Save As Format option. You set this option by choosing Environment from the Options menu and then selecting the Default Save As Format option. There are two format settings: binary and text. To save all new forms and modules in ASCII format, set the Default Save As Format option to Text. This option affects only new forms. Visual Basic continues to save existing binary forms in binary format until you save them as text.

Printing ASCII Forms

If you want to print an ASCII form description, you can do so without having to save the form in ASCII format.

▶ **To print an ASCII form description**

- From the File menu, choose Print.
- Select the Form Text check box.

ASCII Form Structure

The structure of an ASCII form consists of:

- An optional version number.
- A block of text containing the form description.
- The Basic code for the form.

The form description contains the property settings of the form. Nested within the form description are blocks of text that define the properties of controls on the form. Controls contained within other controls have their properties nested within the text of the container. Figure A.1 illustrates the structure of the form description.

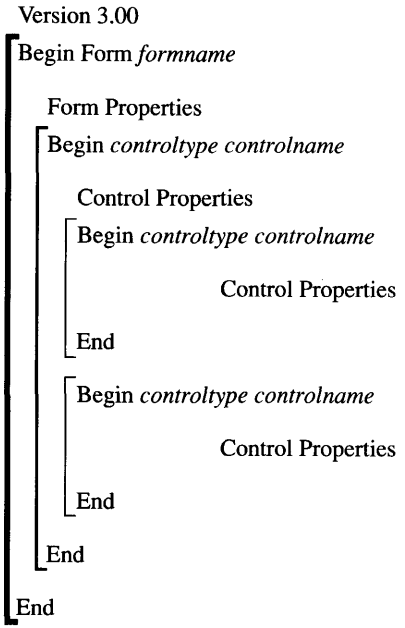


Figure A.1 Structure of the form description

Version Number

The version number for ASCII forms created with Visual Basic for Windows is 3.00. The version number for ASCII forms created with Visual Basic for MS-DOS is 1.00. If you include a version number in the ASCII forms you create for Visual Basic, make sure you use the correct one. If the version number is omitted from the ASCII form, Visual Basic assumes a version number of 3.00.

Form Description

The form description starts with a **Begin** statement and ends with an **End** statement. The syntax of the **Begin** statement is:

Begin {Form|MDIForm} *formname*

The **End** statement determines where the form description ends and the Basic code in the form begins. Without the **End** statement, Visual Basic tries to read the Basic code as if it were describing controls and properties of the form, thus producing errors.

Between the **Begin Form** and **End** statements are the properties of the form itself, followed by descriptions of each control on the form. Figure A.2 shows the nested structure of the form description in greater detail.

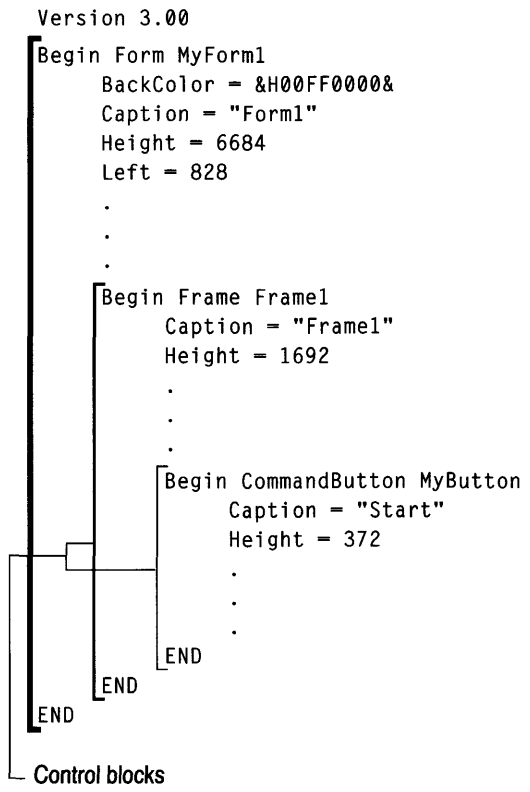


Figure A.2 Nested structure of the form description

Control Blocks

A control block consists of the text in the ASCII form description that defines the properties of an individual control. Like the form description itself, control blocks start with a **Begin** statement and end with an **End** statement. The syntax for a **Begin** statement of a control block is as follows:

Begin *controltype controlname*

The properties for the control appear between the **Begin** statement and the **End** statement.

Control Block Order

The order of the control blocks determines the z-order of the controls. *Z-order* is a relative ordering that determines how controls overlap each other on a form. The first control in the form description establishes the bottom of the z-order. Controls that appear later in the form description are higher in the z-order and therefore overlap controls that are lower in the z-order.

Embedded Control Blocks

Some controls can contain other controls. When a control is contained within another control, its control block is embedded in the control block of the container. You can embed control blocks inside:

- Frames
- Picture boxes
- Menus
- Custom controls, depending on their purpose

Embedded controls are commonly used to place option buttons inside a frame. Visual Basic must have all the information necessary for a container before adding any contained controls, so properties for a control must come before any embedded control blocks. Visual Basic ignores any properties within a control block that appear after embedded control blocks.

Menu Controls

Menu controls must appear together at the end of the form description, just before the Basic code begins. When Visual Basic encounters a menu control during the loading of an ASCII form, it expects to find all the menu controls together. Once it detects a non–menu control following one or more menu controls, Visual Basic assumes there are no more menu controls on the form and ignores any other menu controls it encounters during the loading of that form.

Shortcut Keys

Shortcut keys are keys you use to activate a menu control. The ASCII form uses the same syntax as the **SendKeys** statement to define key combinations: "+" = SHIFT, "^" = CTRL, and "{Fn}" = function key where *n* is the key number. Alphabetic characters represent themselves. Shortcut key syntax is:

```
Shortcut = ^{F4}        ' <CTRL><F4>
```

Note Top-level menus cannot have a shortcut key.

For More Information For complete syntax information on the SendKeys statement, see the *Language Reference*, or search Help for *SendKeys*.

Comments in the Form Description

You can add comments to the form description. The single quotation mark (') is the comment delimiter.

Warning Comments and formatting in the form description are not retained when you save the ASCII form in Visual Basic. However, comments and indents in the code section of the form file are preserved. If you add comments to the form description, Visual Basic does not preserve those comments when you save the form in Visual Basic.

Properties

When Visual Basic saves a form in ASCII format, it arranges the properties alphabetically. However, you can list properties in any order when creating an ASCII form.

When you create an ASCII form, any property you don't list is set to its default value when loaded. When Visual Basic saves a form as an ASCII file, it includes only those properties that do not use default values as their settings.

Syntax

Use this syntax to define properties in the ASCII form description:

property = value

Text property values must appear within double quotation marks. Boolean properties have a value of –1 for **True** and 0 for **False**. Visual Basic interprets any value other than –1 or 0 as **True**. Properties with enumerated values include their numeric value with the description of the value included as a comment. For example, the BorderStyle property appears like this in an ASCII form:

```
BorderStyle = 0          ' None
```

Binary Property Values

Some controls have properties that have binary data as their values, such as the Picture property of picture box and image controls or certain properties of custom controls. Visual Basic saves all binary data for a form in a binary data file separate from the ASCII form.

Visual Basic saves the binary data file in the same directory as the ASCII form. The binary data file has the same file name as the ASCII form and an .FRX file extension. Visual Basic reads the binary data file when loading the ASCII form. The binary data file (.FRX) must be available to the ASCII form when Visual Basic loads it. If you share ASCII forms with others that use a binary data file, make sure you provide the binary data file (.FRX) as well as the ASCII form (.FRM).

Properties having binary data as their values appear in the ASCII form as a reference to a byte offset in the binary data file. For example, the value of a Picture property appears like this in an ASCII form:

```
Begin Image imgDemo
    Picture = MYFORM.FRX:02EB
End
```

The property listing means that the binary data that defines the Picture property of this control begins at byte 2EB (hex) in the file MYFORM.FRX. Visual Basic assumes a byte offset of 0 if no byte offset is specified for a binary property.

Icon Property

The value of the Icon property in an ASCII form depends on what icon is used for the form. The following table lists Icon property values and how those properties appear in an ASCII form.

Icon property setting	ASCII form contents
The default icon	No reference to the Icon property.
(None)	`Icon = 0`
Any icon other than the default icon	Byte offset reference to the binary data file. For example: `Icon = MYFORM.FRX:0000`

TabIndex Property

If the TabIndex property is not specified, Visual Basic assigns the control the earliest possible location in the tab order once all other controls load.

Units of Measurement

Control sizes, x- and y-coordinates, and other property values using units of measurement are expressed in twips. When a control uses a ScaleMode other than twips, Visual Basic converts the twip values in the ASCII form to the units of measurement specified by the control's ScaleMode when loading the form.

Color Values

Color values appear as **RGB** values. For example, the ForeColor property appears like this:

```
ForeColor = &H00FF0000&
```

Visual Basic can also read **QBColor** values, converting them to **RGB** when loading the form. ASCII forms using **QBColor** values must use this syntax:

ForeColor = QBColor(*qbcolor*)

where *qbcolor* is a value from 0 to 15.

Note that the *qbcolor* argument corresponds to the color values used by graphics statements in other versions of Basic, such as Visual Basic for MS-DOS, Microsoft QuickBasic™, and the Microsoft Basic Professional Development System.

Custom Controls

Some custom controls have unnamed properties that the control sets. Unnamed properties in custom controls appear as *Prop<num>* where *num* is the index of the property in the control's property table. The value of properties that are not binary appears as:

```
Prop3 = 3500
```

The value of binary properties appears as:

```
Prop17 = FORM1.FRX:0010
```

Basic Code

The Basic code appears in the ASCII form immediately after the last **End** statement in the form description. Statements in the Declarations section of a form appear first, followed by event procedures, general procedures, and functions.

Sample Form

Figure A.3 shows BLANKER.FRM from the Blanker sample application.

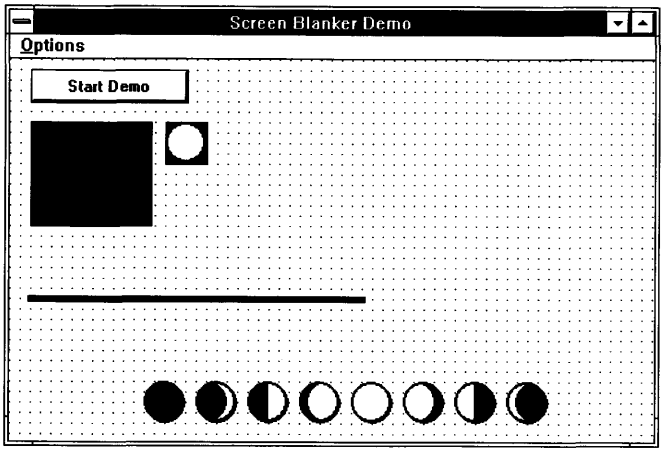


Figure A.3 Sample form from BLANKER.MAK

Sample ASCII Form

Here is part of the ASCII form saved in Visual Basic for the Blanker form. Portions of the ASCII form removed to save space are indicated by a vertical ellipsis.

```
VERSION 3.00
Begin Form DemoForm
    BackColor       =   &H00000000&
    Caption         =   "Screen Blanker Demo"
    ForeColor       =   &H00000000&
    Height          =   4416
    Icon            =   BLANKER.FRX:0000
    Left            =   1092
    LinkMode        =   1                       ' Source
    LinkTopic       =   "Form1"
    ScaleHeight     =   4416
    ScaleWidth      =   7476
    Top             =   2520
    Width           =   7476
    Begin CommandButton cmdStartStop
        BackColor       =   &H00000000&
        Caption         =   "Start Demo"
        Height          =   390
        Left            =   240
        TabIndex        =   0
        Top             =   120
        Width           =   1830
    End
    Begin PictureBox picBall
        AutoSize        =   -1                  ' True
        BackColor       =   &H00000000&
        BorderStyle     =   0                   ' None
        ForeColor       =   &H00FFFFFF&
        Height          =   384
        Left            =   1800
        Picture         =   BLANKER.FRX:0302
        ScaleHeight     =   384
        ScaleWidth      =   384
        TabIndex        =   1
        Top             =   720
        Visible         =   0                   ' False
        Width           =   384
    End
    Begin Line linLineCtl
        DrawWidth       =   5
        ForeColor       =   &H00FF0000&
        Visible         =   0                   ' False
        X1              =   240
        X2              =   4080
        Y1              =   2760
```

```
        Y2              =   2760
    End
    .
    .
    .
    Begin Menu mnuOption
        Caption         =   "&Options"
        Begin Menu mnuLineCtlDemo
            Caption     =   "&Jumpy Line"
            Checked     =   -1              ' True
        End
        Begin Menu mnuCtlMoveDemo
            Caption     =   "Re&bound"
        End
        .
        .
        .
        Begin Menu mnuExit
            Caption     =   "E&xit"
        End
    End
End
.
.
.
Dim Shared FrameNum
Dim Shared XPos
Dim Shared YPos
Dim Shared DoTrap
Dim Shared Motion
.
.
.
Sub CircleDemo ()
    DoTrap = False
    DrawWidth = 1
    DrawStyle = 1
    DrawMode = 7
    Do While DoEvents() And DoTrap = False
    XPos = ScaleWidth / 2
    YPos = ScaleHeight / 2
    Radius = ((YPos * .9) + 1) * Rnd
    RimColor = QBColor(Int(15 * Rnd))
    Circle (XPos, YPos), Radius, RimColor
    Loop
    Cls
End Sub
.
.
.
Sub cmdStartStop_KeyDown (KeyCode As Integer, Shift As Integer)
    DoTrap = True
End Sub
```

Loading and Saving ASCII Forms

When Visual Basic loads an ASCII form into memory, it first converts the form to binary format, so loading ASCII forms takes somewhat longer than loading forms saved in binary format.

When you load an ASCII form in Visual Basic, make changes to that form, and then save those changes, Visual Basic rewrites the ASCII file. Any comments or formatting changes that you had previously made to the ASCII form description are lost when you save the form again in Visual Basic; however, comments that you added to the Basic code following the form description are retained.

File Loading Errors

When Visual Basic encounters an error while loading an ASCII form, it creates a log file and reports that error in the log file. Visual Basic adds error messages to the log file each time it encounters an error in the ASCII form. When the form load is finished, Visual Basic displays a message that tells you an error log file was created.

The log file has the same file name as the ASCII form with a .LOG extension. For example, if errors occurred when loading MYFORM.FRM, Visual Basic would create a log file called MYFORM.LOG. If you reload MYFORM.FRM later and errors continue to occur when loading the form, Visual Basic overwrites the previous MYFORM.LOG log file.

Error Log Messages

The following error messages can appear in an error log file. Note that these error messages deal only with problems that may occur when Visual Basic loads ASCII forms. They do not indicate any problems that may exist in event procedures, general procedures, or any other part of the Basic code.

Property *propertyname* in control *controlname* had an invalid value.

This message appears if Visual Basic encounters an invalid value for a property. Visual Basic changes the property value to the default value for that property.

Control *controlname* could not be loaded.

This message appears if Visual Basic encounters an unknown control in the form description. Visual Basic creates a picture box to represent the unknown control, giving that picture box any valid properties from the unknown control description. When this message appears, a number of invalid property errors are likely to follow.

Form *filename* could not be loaded.

This message appears if Visual Basic encounters the end of file unexpectedly or if the first **Begin** statement is missing.

The Form or MDIForm name *formname* is not valid; cannot load this form.

This message appears if the name of a form is not a valid string in Visual Basic. Visual Basic will not load the form.

Valid strings must start with a letter; can include only letters, numbers and underscores; and must have 40 or fewer characters.

The control name *controlname* is invalid.

This message appears if the name of a control is not a valid string in Visual Basic. Visual Basic will not load the control.

The property name *propertyname* in control *controlname* is invalid.

This message appears if the name of a property is not a valid string in Visual Basic or is longer than 30 characters. Visual Basic will not set the property.

Control *controlname* has a quoted string where the property name should be.

This message appears if Visual Basic finds text inside quotation marks instead of a property name, which you do not place inside quotation marks. For example:

```
"Caption" = "Err, what's that?"
```

In this case, the property name Caption should not have been enclosed in quotation marks. Visual Basic ignores the line in the form description that produced this error.

Property *propertyname* in control *controlname* has an invalid property index.

This message appears if Visual Basic finds a property name with a property index greater than 255. For example:

```
Prop300 = 5436
```

Visual Basic ignores the line in the form description that produced this error.

Property *propertyname* in control *controlname* could not be loaded.

This message appears if Visual Basic encounters an unknown property. Visual Basic skips this property when loading the form.

Property *propertyname* in control *controlname* must be a quoted string.

This message appears if Visual Basic finds a property value without quotation marks that should appear inside quotation marks. For example:

```
Caption = FooBar
```

Visual Basic ignores the line in the form description that produced this error.

Property *propertyname* in control *controlname* has an invalid value.

This message appears if Visual Basic finds a property with a value that is not correct for that control. For example:

```
Top = Cahr(22)      ' Really wanted Char(22).
```

Visual Basic sets the property with its default value.

Property *propertyname* in control *controlname* has an invalid file reference.

This message appears if Visual Basic couldn't use a file name reference. This will happen if the referenced file (probably a binary data file for the form) is not found at the specified directory.

Property *propertyname* in control *controlname* could not be set.

This message appears if Visual Basic cannot set the property of the specified control as indicated by the form description.

Class *classname* in control *controlname* is not a loaded control class.

This message appears if Visual Basic finds a class name it doesn't recognize.

Did not find an index property and control *controlname* already exists. Cannot create this control.

This message appears if Visual Basic finds a control without an index that has the same name as a previously loaded control. Visual Basic doesn't load the control.

Control name too long; truncated to *controlname*.

This message appears if Visual Basic finds a control name longer than 30 characters. Visual Basic loads the control, truncating the name.

Syntax error: property *propertyname* in control *controlname* is missing an '='.

This message appears if Visual Basic finds a property name and value without an equal sign between them. For example:

```
Text        "Bertrand Russell"
```

Visual Basic doesn't load the property.

Cannot load separator menu item in Menu *menuname*.

This message appears if Visual Basic finds a top-level menu control defined as a menu separator. Top-level menu controls cannot be menu separators. Visual Basic does not set the separator property when this error occurs.

Cannot load Menu *menuname*.

This message appears if Visual Basic finds a menu control whose parent menu is defined as a menu separator. Menu controls that act as parents for menu controls in a submenu cannot be menu separators. Visual Basic does not load the menu control.

This message also appears if Visual Basic finds a menu control whose parent menu has its Checked property set to **True**. Menu controls that act as parents for menu controls in a submenu cannot be checked. Visual Basic does not load the menu control.

Cannot set Checked property in Menu *menuname*.

This message appears if Visual Basic finds a top-level menu control with its Checked property set to **True**. Top-level menus cannot be checked. Visual Basic loads the menu control, but doesn't set its Checked property.

Cannot set Shortcut property in *menuname*.

This message appears if Visual Basic finds a top-level menu control with a shortcut key defined. Top-level menus cannot have a shortcut key. Visual Basic loads the menu control, but doesn't set the Shortcut property.

Project File (.MAK) Format

Visual Basic always saves project files (.MAK) in ASCII format. Visual Basic will load version 1.0 project files, which were saved in binary format, but subsequently saves those project files in ASCII format.

The following table describes text that can appear in project files.

ASCII project file text	Description
filename	File names of forms (.FRM), custom controls (.VBX), or modules (.BAS) included in the project. Visual Basic assumes the default file extensions unless specified. If the first file listed in the project file is a form, that form is the startup form.
ProjWinSize = *top*[, *left*[, *width*[, *height*]]]	Initial position and size of the Project window.
ProjWinShow = *show*	Value indicating the display condition of the Project window. 0 = Hidden 1 = Minimized 2 = Normal
Command = "*stringexpression*"	Initial setting for **Command$**.
IconForm = "*stringexpression*"	Form with the icon used for the .EXE. Visual Basic uses the default icon if this is not specified.
HelpFile = "*filename*"	File name of the WinHelp file containing the help for the application.
Title = "*stringexpression*"	Text used as the application title in the Make .EXE dialog or if the /MAKE command-line option is used to create an .EXE file.
EXEName = "*stringexpression*"	File name used as the application's .EXE file name in the Make .EXE dialog.
Path = "*stringexpression*"	Path setting from the Make .EXE dialog.

Sample .MAK File

Here is the .MAK file for the TimeCard sample application:

```
TIMECARD.FRM
FONTDIAL.FRM
TCCANCEL.FRM
ProjWinSize=110,381,307,306
ProjWinShow = 2
Title = "TimeCard"
EXEName = "TIMECARD.EXE"
Path = "C:\VB\SAMPLES\PRINT"
```

APPENDIX B

Icon Library

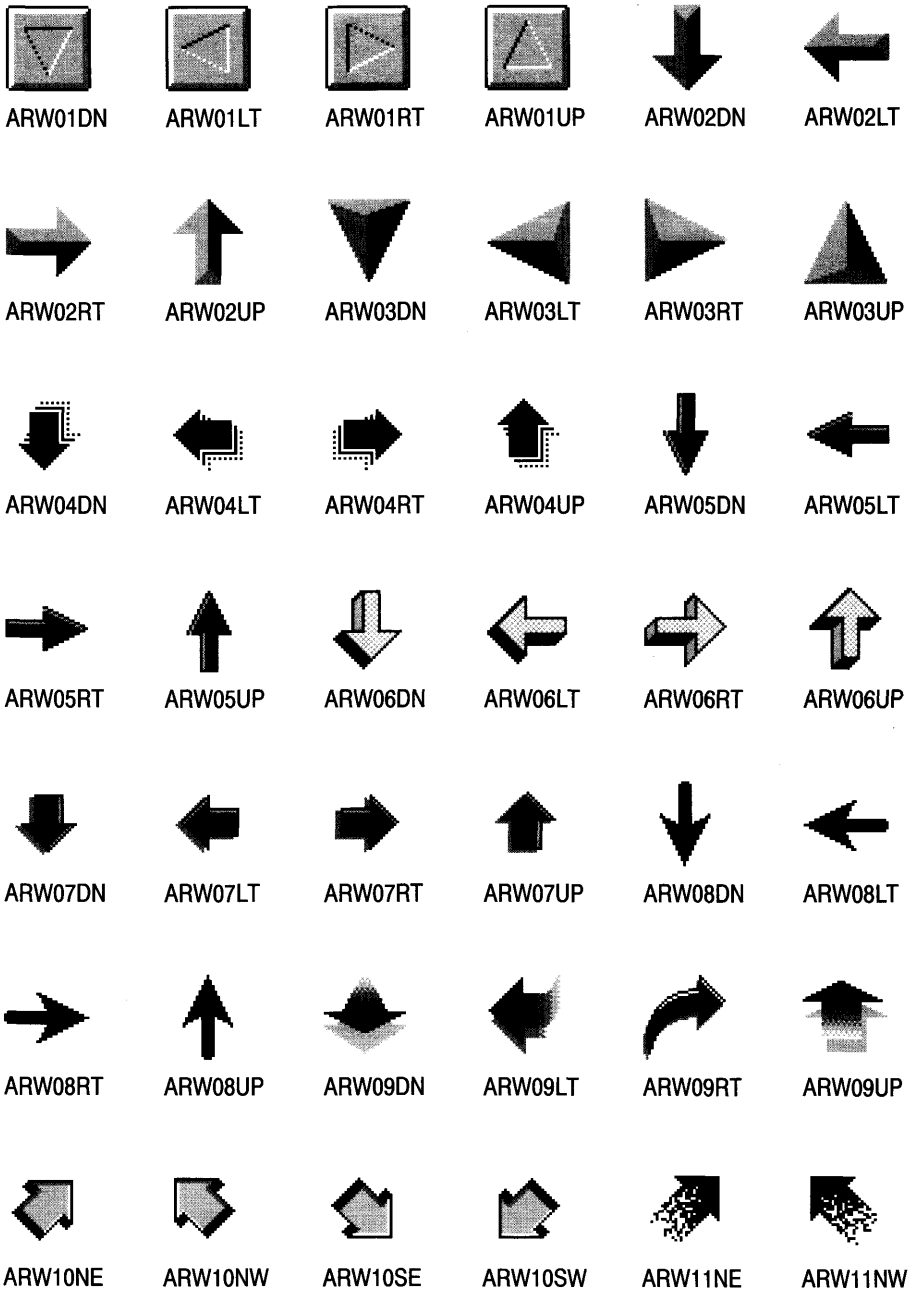
Visual Basic ships with a collection of more than 450 icons that you can use to enhance your applications. This appendix shows printed examples of all the icons. Remember that the icons are designed to be used online, where they appear in full color (16 colors).

For information on how to add icons to your Visual Basic applications, or to see examples of the icons online, search Help for *Icon Library*.

The icons in the Icon Library are divided into the following categories.

Graphics category	Directory	Description
Arrows and pointers	ARROWS	Arrows and pointers
Communication	COMM	Networks, servers, phones, printers
Computers	COMPUTER	Computers, disk drives, diskettes, MDI forms, keyboards, mouse devices
Drag-and-drop	DRAGDROP	Hands and arrows dragging and dropping file folders
Elements	ELEMENTS	Earth, water, fire, clouds, weather, the sun, phases of the moon
Flags	FLAGS	International flags
Industry and transportation	INDUSTRY	Bicycles, cars, airplanes, factories, tools
Mail	MAIL	Mailboxes, envelopes, stamps, forms
Miscellaneous	MISC	People, clocks, first aid signs, locks and keys, buttons, punctuation marks
Office	OFFICE	Card files, charts, file cabinets, folders, graphs, paper clips, labels
Traffic signs	TRAFFIC	Traffic signs
Writing	WRITING	Books, notepads, pens, and pencils

Arrows and Pointers



ARW11SE

ARW11SW

POINT01

POINT02

POINT03

POINT04

POINT05

POINT06

POINT07

POINT08

POINT09

POINT10

POINT11

POINT12

POINT13

POINT14

POINT15

Communication

HANDSHAK

NET01

NET02

NET03

NET04

NET05

NET06

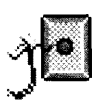
NET07

NET08

NET09A

NET09B

NET10A


NET10B

NET11

NET12

NET13

NET14

PHONE01

PHONE02

PHONE03

PHONE04

PHONE05

PHONE06

PHONE07

PHONE08

PHONE09

PHONE10

PHONE11

PHONE12

PHONE13

PHONE14

PHONE15

Computers

DISK01

DISK02

DISK03

DISK04

DISK05

DISK06

DISK07

DISK08

DISK09

DISK10

DISK11

DISK12

DISK13

DISK14

DISK15

DISKS01

DISKS02

DISKS03

DISKS04

DRIVE01

DRIVE02

KEY01

KEY02

KEY03

KEY04

KEY05

KEY06

KEY07

KEYBRD01

KEYBRD02

KEYBRD03

KEYS01

KEYS02

KEYS03

MAC01

MAC02

MAC03

MAC04

MONITR01

MOUSE01

MOUSE02

MOUSE03

MOUSE04

PC01

PC02

PC03

PC04

TRASH01

TRASH02A

TRASH02B

TRASH03

TRASH04A

TRASH04B

FORM

MDICHILD

MDIPARNT

Drag-and-Drop

DRAG1PG

DRAG2PG

DROP1PG

DRAGFLDR

DROPFLDR

DRAG3PG

Elements

CLOUD

EARTH

LITENING

FIRE

MOON01

MOON02

MOON03

MOON04

MOON05

MOON06

MOON07

MOON08

RAIN

SNOW

SUN

WATER

Flags

FLGASTRL

FLGAUSTA

FLGBRAZL

FLGCAN

FLGDEN

FLGFIN

FLGFRAN

FLGGERM

FLGIREL

FLGITALY

FLGJAPAN

FLGMEX

FLGNETH

FLGNORW

FLGNZ

FLGPORT

FLGRUS

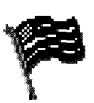
FLGSPAIN

FLGSWED

FLGSWITZ

FLGTURK

FLGUK


FLGUSA01

FLGUSA02

CTRCAN

CTRFRAN

CTRGERM

CTRITALY

CTRJAPAN

CTRMEX

CTRSPAIN

CTRUK

CTRUSA

Industry and Transportation

BICYCLE

CARS

FACTORY

GASPUMP

HAMMER

PLANE

ROCKET

SINEWAVE

WRENCH

Mail

MAIL01A

MAIL01B

MAIL02A

MAIL02B

MAIL03

MAIL04

MAIL05A

MAIL05B

MAIL06

MAIL07

MAIL08

MAIL09

MAIL10

MAIL11

MAIL12

MAIL13

MAIL14

MAIL15

MAIL16A

MAIL16B

MAIL17

MAIL18

Miscellaneous

BINOCULR

BRIDGE

BULLSEYE

CHECKMRK

CLOCK01

CLOCK02

CLOCK03

CLOCK04

EAR

EYE

FACE01

FACE02

FACE03

HOUSE

LIGHTOFF

LIGHTON

MISC01

MISC02

MISC03

MISC04

MISC05

MISC06

MISC07

MISC08

MISC09

MISC10

MISC11

MISC12

MISC13

MISC14

MISC15

MISC16A

MISC16B

MISC17A

MISC17B

MISC18

MISC19

MISC20

MISC21

MISC22

MISC23

MISC24

MISC25

MISC26

MISC27

MISC28

MISC29

MISC30

MISC31

MISC32

MISC33

MISC34

MISC35

MISC36

MISC37

MISC38

MISC39A

MISC39B

MISC40

MISC41

MISC42

MISC43

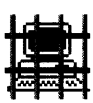
MISC44

SECUR01A

SECUR01B

SECUR02A

SECUR02B


SECUR03

SECUR04

SECUR05

SECUR06

SECUR07

SECUR08

TIMER01

WATCH01

WATCH02

Office

CLIP01

CLIP02

CLIP03

CLIP04

CLIP05

CLIP06

CLIP07

CLIP08

CRDFLE01

CRDFLE02

CRDFLE03

CRDFLE04

CRDFLE05

CRDFLE06

CRDFLE07

CRDFLE08

CRDFLE09

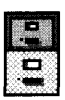
CRDFLE10

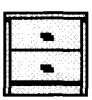
CRDFLE11

CRDFLE12

CRDFLE13

FILES01A


FILES01B


FILES02A

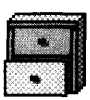
FILES02B

FILES03A

FILES03B

FILES04

FILES05A


FILES05B

FILES06

FILES07

FILES08

FILES09

FILES10

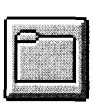
FOLDER01

FOLDER02

FOLDER03

FOLDER04

FOLDER05


FOLDER06

FOLDRS01

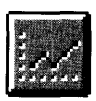
FOLDRS02

GRAPH01

GRAPH02

GRAPH03

GRAPH04


GRAPH05

GRAPH06

GRAPH07

GRAPH08

GRAPH09

GRAPH10

GRAPH11

GRAPH12

GRAPH13

GRAPH14

LABELS

RULERS

Traffic Signs

TRFFC01

TRFFC02

TRFFC03

TRFFC04

TRFFC05

TRFFC06

TRFFC07

TRFFC08

TRFFC09

TRFFC10A

TRFFC10B

TRFFC10C

TRFFC11

TRFFC12

TRFFC13

TRFFC14

TRFFC15

TRFFC16

TRFFC17

TRFFC18A

TRFFC18B

TRFFC19A

TRFFC19B

TRFFC20

Writing

BOOK01A

BOOK01B

BOOK02

BOOK03

BOOK04

BOOK05

BOOKS01

BOOKS02

BOOKS03

BOOKS04

BOOKS05

ERASE01

ERASE02

NOTE01

NOTE02

NOTE03

NOTE04

NOTE05

NOTE06

NOTE07

NOTE08

NOTE09

NOTE10

NOTE11

NOTE12

NOTE13

NOTE14

NOTE15

NOTE16

NOTE17

NOTE18

PEN01

PEN02

PEN03

PEN04

PEN05

PEN06

PEN07

PENCIL01

PENCIL02

PENCIL03

PENCIL04

PENCIL05

PENCIL06

PENCIL07

PENCIL08

PENCIL09

PENCIL10

PENCIL11

PENCIL12

PENCIL13

PENCIL14

PENS01

PENS02

PENS03

PENS04

APPENDIX C

Compatibility with Other Versions

Like every vital programming language, Basic is constantly evolving. In general, every new version is compatible with its predecessor; however, new features may conflict with existing code. Also, binary source-file formats may change between versions. Finally, platform changes—such as moving from MS-DOS to Windows—may introduce differences in the way features behave.

This appendix is written for programmers who have applications written in a previous version of Visual Basic for Windows or Visual Basic for MS-DOS and want to modify or update these applications using Visual Basic version 3.0 for Windows. If you have applications written in other versions of Basic, such as Microsoft QuickBasic™, you may want to read the section on Visual Basic for MS-DOS, since much of the same information applies for all MS-DOS Basics.

After you read this appendix, you will know how to:

- Upgrade a Visual Basic for Windows application to Visual Basic version 3.0.
- Convert Visual Basic applications from MS-DOS to Windows.
- Handle some of the differences between Windows and MS-DOS platforms.

Note The Microsoft Visual Basic for MS-DOS *Programmer's Guide* includes procedures for converting applications from MS-DOS to Windows. The information in this appendix supercedes those procedures. The Microsoft Visual Basic Translator shipped with Visual Basic for MS-DOS will run only with Visual Basic version 1.0 for Windows.

Visual Basic 1.0 for Windows

Version 3.0 has significant performance improvements over version 1.0, so there is a compelling reason to rebuild existing Visual Basic applications using version 3.0—even if you don't take advantage of new language features. This section shows how to convert a version 1.0 application to Visual Basic 3.0.

Important Once you save a project in Visual Basic 3.0 for Windows, you cannot open it again in Visual Basic 1.0 for Windows. The two versions do not use identical file formats.

Although the (binary) file formats of version 2.0 and 3.0 are also different, both versions allow you to save projects as ASCII text.

▶ **To upgrade a Visual Basic version 1.0 application to version 3.0**

1. Make a backup copy of the project and archive it.
2. Start Visual Basic version 3.0 and open the version 1.0 project.

   Visual Basic displays a dialog box informing you that the project is in an old format. Choose OK.
3. Set a default data type for any form or code module that does not already contain a **Def***type* statement.
4. Resolve any conflicts between new, version 3.0, keywords, and existing symbol names. (*Symbols* are the names of variables, constants, objects, procedures, parameters, or labels used in code.)
5. Run the application to test the changes in step 4.
6. Save the project.

Setting the Default Data Type

In Visual Basic 1.0 for Windows, the default data type is **Single**. In versions 2.0 and 3.0, the default data type is **Variant**. If you use **Get** and **Put** with variables of the default data type, you will seriously disrupt existing random-access files, since the **Single** and **Variant** data types are different sizes.

This applies to variables for which you have not declared a data type. For example:

```
Put #1, x                ' x relies on the default data type.
```

To solve this problem, declare the data type of these variables by doing one of the following:

- Explicitly declare the variable. For example:

  ```
  Dim x As Single
  ```

- Use a type-declaration character. For example:

```
Put #1, x!          ' x! is a variable of data type Single.
```

- Add a **Def***type* statement to set the default data type to **Single**. For example:

```
DefSng A-Z          ' You can specify any letter range needed.
```

Visual Basic 2.0 for Windows

This section lists the compatibility issues between Visual Basic versions 2.0 and 3.0.

The OLE Control

The OLE control in Visual Basic version 3.0 is not compatible with the OLE client control that shipped with Visual Basic 2.0. Not only has the name of the OLE control been changed, the control has also been updated to exploit some of the features of OLE 2.0 and to support OLE Automation. If you want, however, you can continue to use the custom control OLECLIEN.VBX that shipped with version 2.0.

The OLE control requires Microsoft Windows version 3.1 or later.

Converting Visual Basic Version 2.0 ODBC Applications

When converting an application that uses the ODBC features of Visual Basic 2.0 (Professional Edition) to Visual Basic 3.0 , be aware of the following incompatibilities:

- The FT_ constants have been removed from CONSTANT.TXT and have been renamed with the prefix DB_. These constants, along with many other new constants, are located in the file DATACONS.TXT.
- The trappable error numbers and messages related to data access have changed. The Visual Basic 3.0 trappable errors are listed in Help and in the *Language Reference*.
- In Visual Basic version 3.0, the Tabledef.Name property returns *Owner*.Tablename. In version 2.0, it just returned Tablename.
- In version 2.0 transactions were performed per database. In version 3.0, transactions are always global (even if you use the database methods, which are no longer recommended). This only makes a difference if you have code that opens multiple databases and maintains separate transactions for each database.

The implementation of data access objects in Visual Basic version 3.0 is far more extensive than in Visual Basic version 2.0. When converting applications using ODBC, consider the following strategies:

- Use the data control to replace hard-coded procedures that perform equivalent functions.
- Data access is no longer limited to Microsoft SQL Server through an ODBC link. You can now access a variety of databases including Microsoft Access, FoxPro, Btrieve, Paradox, and dBASE. You can also access external servers including Microsoft SQL Server and Oracle through ODBC.
- Use the **Move**, **Seek**, and **Find** methods to move freely within three types of recordset objects.
- Limitations placed on the updatability of **Dynasets** have been eased. In many cases, a unique index is no longer required to update a recordset object.

For More Information For information on the data control, see Chapter 20, "Accessing Databases with the Data Control." If you have the Professional Edition, see the *Data Access Guide* for details on using the other data access features.

Resolving Keyword Conflicts

Visual Basic version 2.0 has 50 keywords that are not available in version 1.0. Visual Basic version 3.0 has added over 40 keywords to those available in version 2.0. If you use one of these words as the name of a symbol in code, it may cause a syntax error when you run the project. To fix this error, search for and replace the old symbol name with a new, unique name in all code modules.

For object names that conflict with keywords, you can resolve the conflict without renaming the object. Simply enclose the name in square brackets ([ ]). For example:

```
[New].cmdOne.Caption = "One"          ' New is now reserved.
```

Brackets identify a symbol as an object name. They cannot be used with other symbols, such as variable or procedure names.

New Keywords in Visual Basic 2.0

The following is a list of words reserved by Visual Basic version 2.0, but not by version 1.0. Items with asterisks (*) are reserved in both the Standard Edition and the Professional Edition, although the features are available only in the Professional Edition.

AddNew*	**Arrange**	**CommitTrans***
AppendChunk*	**BeginTrans***	**CreateDynaset***

CStr
CVar
CVDate
DefVar
Delete*
Edit*
ExecuteSQL*
Explicit
False
FieldSize*
FileCopy
FileDateTime
FileLen
GetAttr
GetChunk
IsDate
IsEmpty
IsNull
IsNumeric
Like
Me
MoveFirst*
MoveLast*
MoveNext*
MovePrevious*
MoveRelative*
New
NextBlock*
Nothing
Null
OpenDatabase*
Preserve
Private
RegisterDatabase*
Rollback*
Set
SetAttr
StrComp
Trim$
True
Update*
Variant
VarType
ZOrder

New Keywords in Visual Basic 3.0

The following is a list of words reserved by Visual Basic version 3.0, but not by version 2.0. Items with asterisks (*) are reserved in both the Standard Edition and the Professional Edition, although the features are available only in the Professional Edition.

Choose
Clone*
CommitTrans
CompactDatabase
CreateDatabase*
CreateObject
CreateQueryDef*
CreateSnapshot*
DateAdd
DateDiff
DatePart
DDB
DeleteQueryDef*
Execute
FindFirst
FindLast
FindNext
FindPrevious
FreeLocks
FV
GetObject
IIf
IPmt
ListFields*
ListIndexes*
ListParameters*

ListTables*
NPer
OpenQueryDef*
OpenTable*
Partition
Pmt
PopupMenu
PPmt
PV
Rate
RepairDatabase
Rollback
Seek*
SetDataAccessOption
SetDefaultWorkspace
SLN
Switch
SYD
UpdateControls
UpdateRecord

Visual Basic for MS-DOS

If you have Visual Basic for MS-DOS, you can create applications that run under both MS-DOS and Windows. Such applications exist as separate .EXE files for each target environment. There is no way to create a single file that will run in both environments.

When you convert a Visual Basic application from MS-DOS to Windows, you are faced with the following problems:

- The binary file formats of the modules are different for the two products.
- The appearance of some forms and controls is different.
- There are language differences between Visual Basic for MS-DOS and Visual Basic for Windows.

  Some of these language differences result because Windows and MS-DOS control their respective environments in very different ways. Other differences occur because some language components of Visual Basic for MS-DOS are simply not supported by Visual Basic for Windows.

You resolve these problems using the following general procedure. The rest of this appendix includes specific information on these general steps.

▶ To convert an application from MS-DOS to Windows

1. Start Visual Basic for MS-DOS and open the project to convert.
2. Move all procedure declarations into an include file (.BI) and add a **$INCLUDE** metacommand at the module level of each file.
3. Save each form and code module in text format.
4. Copy the project's .MAK, .FRM, and .BAS files to a new directory.

5. Edit the project file (.MAK) to change explicit path names (if used).
6. Open the project in Visual Basic for Windows and view each form. If an error occurs reading the MS-DOS form definition, Visual Basic for Windows creates a .LOG file and displays a warning.
7. Review the .LOG files (if any) to see a listing of incompatibilities. Fix these errors using Visual Basic for Windows or a text editor.
8. Adjust the visual interface of the application to best match the capabilities of Windows.
9. Resolve any conflicts between Visual Basic for Windows keywords and existing symbol names. (*Symbols* are the names of variables, constants, objects, procedures, parameters, or labels used in code.)
10. Convert platform-specific code to work with the target environment.
11. Run your application in the target environment to verify its behavior and to test your changes.

Note Visual Basic automatically capitalizes keywords based on an internal convention. In MS-DOS, the convention is all uppercase. In Windows, the convention is to capitalize the first letter of each keyword. Although the appearance of your code may change, the content remains the same.

Moving Procedure Declarations to an Include File

Visual Basic for MS-DOS uses **DECLARE** statements to declare Basic and Quick library procedures (.QLB). Visual Basic for Windows uses **Declare** only for DLLs.

To avoid syntax errors when you move your MS-DOS application to Windows, move **DECLARE** statements to a Visual Basic for MS-DOS include file (.BI) and add a **$INCLUDE** metacommand at the module level of each file. For example:

```
'$INCLUDE: 'DOSINC.BI'
```

Since Visual Basic for Windows does not recognize the **$INCLUDE** metacommand, this statement is treated as a comment. The code will continue to run correctly in Visual Basic for MS-DOS, however.

Saving Form and Code Modules in Text Format

By default, all forms and code modules (.FRM and .BAS) are saved in a binary format. As mentioned previously, however, the binary format is not compatible between versions of Visual Basic. To load an MS-DOS project in Windows, you must first save all of the files in text format. Once a file is saved in text format, it will continue to be saved in that format by default.

▶ **To save a form or code module in text format**

1. Start Visual Basic for MS-DOS and open the project to convert.
2. From the File menu, choose Save File As.

   Visual Basic displays the Save File As dialog box.
3. Select the Save as Text check box.
4. Choose OK.

Copying Project Files

After you save the forms and code modules in text format, you should copy the entire project to a new location so you do not overwrite your source application. Copy only the .MAK file and the files listed in the .MAK file. Since Windows and MS-DOS custom controls are not compatible, there is no need to copy files with .QLB extensions. Portions of applications that depend on custom controls or Quick library procedures (.QLB files) must be rewritten.

Editing the Project File

If you saved your modules to a directory other than the current one, Visual Basic for MS-DOS includes the paths in your project file (.MAK). Load this file in a text editor, and change the paths to reflect the directory of the new project.

Opening the Project and Viewing Forms

After copying and editing the project files, start Visual Basic for Windows and open the project. View each form in the project by selecting the file in the Project window and choosing the View Form button.

If errors occur, Visual Basic for Windows displays a warning, `Errors during load. Refer to 'filename.LOG' for details`. Make a note of the file name and choose the OK button to close the dialog box.

Reviewing the .LOG Files

Two types of errors occur when loading an MS-DOS form in Visual Basic for Windows: invalid form names and invalid properties.

Visual Basic for Windows forms cannot be named "MDIForm." If you have a form with this name in your MS-DOS project, it will generate the error message `MDIForm is a class name` and the form will not load. You must change the form name in the .FRM file using a text editor. For example:

```
Version 1.00
BEGIN Form tMDIForm          ' Added "t."
```

After making this change, reload the project.

When you view an MS-DOS form that has a property or property setting not supported by Visual Basic for Windows, Visual Basic for Windows creates a .LOG file similar to the one shown below:

```
Line 120 :Property BorderStyle in control picJAW could not be set.
Line 141 :Property Attached in control HScroll1 could not be loaded.
Line 254 :Property Attached in control VScroll1 could not be loaded.
```

It is important to review the .LOG files to check for properties that may be used in code or that critically affect the appearance or behavior of your application.

When you save a form file in ASCII format, Visual Basic for Windows omits default property settings, invalid property settings, and invalid properties. Since Visual Basic for Windows treats omitted property settings as default values, invalid property settings revert to their default. For more information, see Appendix A, "ASCII File Formats."

Note Visual Basic for Windows includes a Save as Text option. If you select this option, all your files are saved as text by default. It's a good idea to use this option when converting applications from MS-DOS, since it makes it possible to edit form descriptions using a text editor.

Adjusting the Visual Interface

When Visual Basic for Windows loads a project created in Visual Basic for MS-DOS, it changes the appearance of forms as they are loaded. These changes are due to the differences between Visual Basic for MS-DOS and Visual Basic for Windows:

- The position and size of controls may change because the Visual Basic for MS-DOS character coordinates are converted to twips in Visual Basic for Windows.
- Colors may change because the Visual Basic for MS-DOS colors are converted to **RGB** values.
- Custom controls are mapped to picture controls.

Use the Visual Basic for Windows design tools to adjust the appearance and placement of controls as appropriate.

You may be able to define global changes to the appearance of your application by editing the ASCII form description for each form module before you load the form in Visual Basic. For information about the format of the ASCII form description, see Appendix A, "ASCII File Formats."

Resolving Keyword Conflicts

Visual Basic for Windows includes about 40 keywords that are not supported by Visual Basic for MS-DOS. If you used one of these words as a symbol in code, it causes a syntax error when you attempt to run the project. To fix this error, search for and replace the old symbol name with a new, unique name in all modules.

If you used one of these keywords as a procedure name or parameter in a procedure definition, the procedure is placed in the Declarations section of the code module. You must cut and paste the code from the procedure into a new procedure definition.

Keywords Incompatible with Visual Basic for MS-DOS

The following words are not reserved by Visual Basic for MS-DOS but are reserved by Visual Basic for Windows.

AddNew
AppendChunk
Arrange
BeginTrans
Choose
Clone
CommitTrans
CompactDatabase
Compare
CreateDatabase
CreateDynaset
CreateObject
CreateQueryDef
CreateSnapshot
CStr
CVar
CVDate
DateAdd
DateDiff
DatePart
DDB
DefVar
DeleteQueryDef
Edit
Execute
ExecuteSQL
False
FieldSize
FileCopy
FileDateTime
FileLen
FindFirst
FindLast
FindNext
FindPrevious
FreeLocks
FV
GetAttr
GetChunk
GetData
GetFormat
GetObject
Global
IIf
IPmt
IsDate
IsEmpty
IsHidden
IsNull
IsNumeric
Like
LinkExecute
LinkPoke
LinkRequest
LinkSend
ListFields
ListIndexes
ListParameters
ListTables
Me
MoveRelative
New
NextBlock
Nothing
NPer
Null

OpenDatabase	**RepairDatabase**	**StrComp**
OpenQueryDef	**Rollback**	**Switch**
OpenTable	**Scale**	**SYD**
Partition	**Seek**	**Text**
Pmt	**SendKeys**	**Trim**
PopupMenu	**Set**	**True**
PPmt	**SetAttr**	**UpdateControls**
Private	**SetData**	**UpdateRecord**
PV	**SetDataAccessOption**	**Variant**
Rate	**SetDefaultWorkspace**	**VarType**
RegisterDatabase	**SLN**	**ZOrder**

Converting Platform-Specific Code

Some applications, such as the sample application CALC.FRM, will run fine on either MS-DOS or Windows without changing much code. However, more complex applications can require significant recoding. There are several categories of platform-specific code that you must change to move an application from MS-DOS to Windows:

- **System calls.** Code that relies on MS-DOS interrupts must be rewritten.
- **Data type and scoping differences.** Visual Basic for Windows and Visual Basic for MS-DOS have different default data types. The scoping rules for arrays are also different.
- **Unsupported keywords**. Most of the Visual Basic for MS-DOS keywords that are not supported in Windows have functionality that is replaced by forms and event-driven programming style.
- **Different coding mechanisms.** Some features, such as declaring global variables and displaying run-time graphics, require different coding mechanisms because of language differences between Visual Basic for Windows and MS-DOS.

The following four sections summarize these differences.

System Calls

Many MS-DOS system calls have some equivalent under Windows. However, the mechanism for calling Windows functions is very different from using MS-DOS interrupts. For information on calling Windows functions, see Chapter 24, "Calling Procedures in DLLs."

Data Type and Scoping Differences

In Visual Basic for MS-DOS, the default data type is **SINGLE**. In Visual Basic for Windows, the default data type is **Variant**. This difference can cause serious errors using **Get** and **Put** on existing data files, since the two data types are different sizes. To avoid this, add a **DefSng** statement to each module that doesn't already include a **Def***type* statement.

To remain consistent with earlier versions of Basic, Visual Basic for MS-DOS does not allow shared array variables to be shadowed at the procedure level. For example, the following code behaves differently in MS-DOS and Windows:

```
' Module level
DIM SHARED Array() AS INTEGER
SUB ChangeArray ()
' In MS-DOS, dimensions shared array. In Windows, creates
' a new copy of the array (shadows shared array).
    DIM Array(10)
END SUB
```

To avoid this unexpected behavior, rename the procedure-level array.

Unsupported Keywords

Visual Basic for Windows omits about 100 keywords that are supported by Visual Basic for MS-DOS. Using any of these keywords in the MS-DOS application that you convert to Windows results in omitted functionality. You must rewrite any code that relies on these keywords.

ALL	**CSRLIN**	**ERDEV$**
BLOAD	**CVI**	**EVENT**
BOF	**CVC**	**FIELD**
BSAVE	**CVD**	**FILES**
CALLS	**CVDMBF**	**FN**
CDECL	**CVL**	**FRE**
CHAIN	**CVS**	**GETINDEX$**
CHECKPOINT	**CVSMBF**	**INKEY$**
COLOR	**DELETEINDEX**	**INP**
COM	**DELETETABLE**	**INSERT**
COMMON	**DRAW**	**IOCTL**
CREATEINDEX	**ERDEV**	**IOCTL$**

ISAM	**PMAP**	**SSEGADD**
KEY	**POKE**	**STACK**
LIST	**POS**	**STICK**
LOCATE	**PRESET**	**STRIG**
LPOS	**RUN**	**SWAP**
LPRINT	**SADD**	**SYSTEM**
MKC$	**SAVEPOINT**	**TEXTCOMP**
MKDMBF$	**SCREEN**	**TRON**
MKI$	**SEEKEQ**	**TROFF**
MKL$	**SEEKGE**	**UEVENT**
MKS$	**SEEKGT**	**UPDATE**
MKSMBF$	**SEG**	**VARPTR**
OUT	**SETINDEX**	**VARPTR$**
PAINT	**SETMEM**	**VARSEG**
PALETTE	**SETUEVENT**	**VIEW**
PCOPY	**SIGNAL**	**WAIT**
PEEK	**SLEEP**	**WINDOW**
PEN	**SOUND**	
PLAY	**SSEG**	

Note Visual Basic for Windows does not support **DEF FN** functions.

Different Coding Mechanisms

The scoping mechanisms are different between Visual Basic for MS-DOS and Windows, as shown by the following table.

Visual Basic for MS-DOS construction	Visual Basic for Windows equivalent
COMMON SHARED	**Global**.
DIM SHARED (at module level)	**Dim** (at module level).
SHARED attribute (at procedure level)	None. Use module-level variable, which is visible to all procedures.
COMMON	None. Module-level–only variables are not useful in Visual Basic for Windows.

Visual Basic for MS-DOS applications cannot display forms and graphics at the same time. This limitation does not exist in Visual Basic for Windows. Code that displays graphics must be completely rewritten for use in Windows.

Similarly, custom controls are very different in Visual Basic for MS-DOS and Windows. Custom controls and Basic code that rely on custom controls must be rewritten for Windows.

Running Your Converted Application

To complete the conversion, run your application in Visual Basic for Windows and test for errors. The following is a list of problems you may encounter. Only the most likely problems are listed here, along with the error message you may see.

Problem	Possible error message	Corrective action
Visual Basic for Windows reserves **True** and **False** as intrinsic constants.	`Expected: identifier`	Remove or comment out constant declarations for TRUE and FALSE.
Visual Basic for Windows allows executable code only inside a procedure; there can be no executable code or labels at the module level of a Visual Basic for Windows application.	`Invalid outside Sub or Function`	Move module-level executable code into a **Sub Main** ( ) procedure and set it to be the startup procedure. Convert module-level error-handling code to local error handlers.
Visual Basic for Windows does not recognize the **DATA** statement.	`Expected: statement`	Convert the values in the **DATA** statement(s) to a sequential file; replace the **DATA** statement(s) with a block of code that opens, reads, and closes the file.
Visual Basic for Windows does not recognize the **COMMON** keyword.	`Expected: end-of-statement`	Replace the **COMMON SHARED** keywords with the keyword **GLOBAL.** Move module-level code into procedures.
Visual Basic for Windows puts invalid procedure definitions at the Declarations level of a module.	`Expected: parameter or )`	Cut the procedure code from the Declarations level and paste it into a valid procedure definition. Delete the invalid procedure definition.

Techniques for Platform-Independent Applications

Writing an application that will convert easily from MS-DOS to Windows requires some planning. Here are some suggestions that make the process easier.

- **Start with MS-DOS.** The appearance of MS-DOS forms requires less adjustment when converted to Windows. Also, since the forms-related features of Visual Basic for MS-DOS are a subset of those available in Windows, you are less likely to inadvertently use a feature for which there is no solution.
- **Use the portable subset of the language.** The lists of unsupported keywords provided earlier in this chapter will give you an idea of features to avoid when planning a portable application.
- **Debug your MS-DOS application before you convert it to Windows.** If you start with an application that runs without errors, you can assume that errors you encounter when converting are related to differences between the two platforms. This will save you lots of time trouble-shooting.
- **Create parallel include files.** In Visual Basic for MS-DOS, place **COMMON SHARED** and constant declarations in an include file (.BI). For the Visual Basic for Windows application, add a Basic module (.BAS) with parallel **Global** declarations and appropriate constants.
- **Package portable procedures in code modules.** Procedures that will run in MS-DOS or Windows without modification should be placed in code modules (.BAS) that you can share between the two platforms.
- **Modularize run-time graphics.** Place all run-time graphics procedures in a separate module. Visual Basic for MS-DOS does not support graphics while forms are showing, and code for MS-DOS graphics is more hardware-dependent than similar code for Windows.
- **Use parentheses with all FUNCTION calls.** Visual Basic for MS-DOS lets you omit the parentheses when you call a **FUNCTION** procedure that takes no arguments. In Windows, the parentheses are required to identify the call as a **Function** procedure.
- **Test your portability.** When developing a large application, periodically test its portability during development. This will help you develop a "portable" coding style.

Detecting the Platform at Startup

Applications created with Visual Basic for Windows include code that will automatically start Windows if it is not running on your system. This feature can be used to create an application that will run in either Windows or MS-DOS.

The following batch file starts CALCWIN.EXE. If Windows is not available or cannot start, the batch file runs CALCDOS.EXE instead.

```
@ECHO OFF
CALCWIN.EXE
IF ERRORLEVEL 1 CALCDOS.EXE
```

The application sets an MS-DOS error code of 1 if it can't start Windows. Otherwise, the error code is 0 when the application ends.

Note that this feature also applies to MS-DOS sessions. If Windows can't start because it is already running, the preceding batch file will start CALCDOS.EXE.

APPENDIX D

Specifications and Limitations

This appendix describes the limitations that apply to Visual Basic and to the applications you create with Visual Basic. While many of these limitations are described in terms of a specific number, keep in mind that other conditions (such as available memory and system resources) may impose a restriction before the specified limit is reached.

- Hardware and Software Requirements for Visual Basic Applications
- Application Limitations
- Form Limitations
- Control Limitations
- Code Limitations
- Windows Limitations

Hardware and Software Requirements for Visual Basic Applications

These are the hardware and software requirements for Visual Basic applications:

- Microsoft Windows 3.0 or later, running in standard or enhanced mode.
- 80286 or higher microprocessor.
- CGA, EGA, VGA, 8514, Hercules, or other display compatible with Windows.
- 1 MB of memory.

It is possible to write Visual Basic applications that don't require a mouse. Of course, you may need to do additional work to provide keyboard access that meets the specific needs of your application.

Application Limitations

The following limitations apply to a Visual Basic application.

Objects per Application

There is a limit of 256 distinct objects in a project. This limit includes all form types, control types, and global objects. It does not include multiple instances of forms or instances of controls on forms.

Forms per Application

There is a limit of approximately 230 forms in a project. An application can have up to 80 of these forms loaded at one time (each loaded instance of a form counts towards this total). The total number of windows in the system (including controls and windows in other applications) may restrict the number of forms that may be visible; for more information, see the section "Total Number of Controls" later in this appendix.

Procedures per Application

The sum of all procedures, modules and forms, and DLL declarations in the application must be less than 5,200. All procedures (public and private) count towards this limit except event procedures that don't contain any code.

Form Limitations

The following limitations apply to each form in Visual Basic.

Total Number of Controls

The maximum number of controls allowed on a single form is 470 (or less, depending on the type of controls). However, there is a limit of 254 control names per form. A control array counts only once toward this limit because all the controls in the array share a single control name. Thus, up to 470 controls may exist per form, depending on how many controls are in control arrays.

Each nongraphical control (all the controls except shape, line, image, and label) uses a window. Each window uses system resources, limiting the total number of windows that can be in existence at one time. Under Windows 3.1, this limit is about 600 windows; under Windows 3.0, this limit is about 475 windows. Notice that *all* windows in existence count towards this limit, including forms and windows in other running applications. Performance may degrade as the number of controls increases.

To reduce consumption of system resources, use the shape, line, label, and image controls instead of picture box controls to create or display graphics.

Controls in Open Forms

The maximum number of controls in all open forms is about 600, not including graphical controls and labels.

Properties

The data in form properties and properties for all controls on a given form are stored together in a single data segment limited to 64K, except the List property of combo box and list box controls, and the Text property of multiline text box controls.

Control Limitations

The following limitations apply to controls in Visual Basic.

List Property

The List property of each list box or combo box gets its own data segment, limited to 64K. The maximum number of items in each list is 5,440. Each item in the List property is limited to 1K. Any item over this limit is truncated.

Text Property

The Text property of each text box is limited to 32K. The characters in the Text property for each text box are stored with the other form and control properties in the form's single 64K data segment, unless the MultiLine property of the text box is set to **True**. The Text property for each multiline text box is stored in its own data segment.

Caption Property

The Caption property of a label is limited to 1K. The Caption property of command buttons, option buttons, check boxes, and frames are limited to 255 characters. Any caption over these limits is truncated.

Tag Property

The Tag property of a control is limited to 32K. Any tag over this limit causes an `Out of memory` error.

Code Limitations

Each procedure can contain up to 64K of p-code (the internal representation of your code). The module-level Declarations section of each form or code module can also contain up to 64K of p-code. If a procedure or module exceeds this limit, Visual Basic generates an `Out of memory` error. If you encounter this error, you can avoid it by breaking extremely large procedures into several smaller procedures, or by moving module-level declarations into another module.

The amount of p-code in a procedure or Declarations section is roughly equivalent to the number of characters in the textual representation of the procedure or Declarations section. To estimate the amount of p-code in a procedure or module-level Declarations section, create an ASCII file containing the procedure or Declarations section. The size of the ASCII file is a rough approximation of size of the p-code in the procedure or Declarations section.

This estimate is true only when you are working in the development environment. Visual Basic does not include identifiers (procedure, variable, and object names) or comments in the .EXE files you create, so the resulting .EXE is much smaller. If you don't exceed the 64K limit on p-code in the development environment, you won't exceed it in the finished .EXE.

Symbol Tables

Visual Basic uses "symbol tables" to store the names of identifiers (variables, procedures, constants, and so on) in your code. Each symbol table is limited to 64K.

Module Symbol Table

Each form and code module uses a symbol table that contains:

- The actual text of the names of **Sub** and **Function** procedures
- Module-level and local variable names
- Module-level DLL procedure declarations
- Line numbers
- Line labels
- An additional 4 bytes of overhead for each of these names and symbols

These tables are each limited to 64K, but this limit is almost never reached. These tables are not included in .EXE files you create, reducing the size of the finished .EXE.

Global Symbol Table

The entire application uses a single symbol table that contains all global names. This includes:

- Global constant names
- Global variable names
- User-defined type definitions
- Module names
- Global DLL procedure declarations
- An additional 4 bytes of overhead for each of these names and symbols

The global symbol table is limited to 64K, or about 4,600 global names per application, assuming 10 bytes (plus 4 bytes overhead) per name. If an application exceeds the 64K limit, an `Out of memory` error occurs. To avoid this problem, use shorter global names and eliminate unused global variable, constant, and DLL procedure declarations. When you create an .EXE file, most of this table is omitted from the .EXE (only the DLL procedure declarations are included).

Data Limitations

The following limitations apply to variables in the Visual Basic language.

Global Data

Each application gets one data segment of up to 64K for global variables and constants.

Form and Code Module Data

Each form and module has its own data segment that can be up to 64K. This data segment contains the following data:

- Local variables declared with **Static**
- Module-level fixed-length string variables
- Module-level variables other than arrays and variable-length strings
- Module-level constants
- Tracking data for arrays and variable-length strings
- Tracking data for controls referenced in code for a form
- Tracking data for nonstatic local variables

Size of Variable Types

Visual Basic variables come in seven standard types, as shown in the following table.

Type name	Size	Range
Integer	2 bytes	−32,768 to 32,767
Long	4 bytes	−2,147,483,648 to 2,147,483,647
Single	4 bytes	−3.40E38 to −1.40E-45 for negative values; 1.40E-45 to 3.40E38 for positive values
Double	8 bytes	−1.79E308 to -4.94E-324 for negative values; 4.94E-324 to 1.79E308 for positive values

Type name	Size	Range
Currency	8 bytes	−9.22E+14 to 9.22E+14
String	4 bytes, plus 0 to 64K bytes (1 byte per character in the string)	0 to 64K characters
Variant (the default)	16 bytes, plus 1 byte per character (up to 64K) if storing a string	Depends on data

String Data

Individual strings always have a maximum size of 64K characters. This limit applies whether the string is an ordinary variable, an element in a user-defined type, or an element in an array. However, the *sum* of the lengths of multiple strings can exceed 64K in certain circumstances. Each of the following groups of strings can exceed 64K:

- The sum of all global variable-length string variables.
- The sum of all module-level variable-length string variables.
- The sum of all local variable-length string variables.
- The sum of all variable-length string elements in user-defined types.

The following groups of strings *cannot* exceed 64K:

- The sum of all global, module-level, or local fixed-length strings.
- The sum of all elements of each variable-length string array within a user-defined type.

Because of the way Visual Basic allocates memory for strings, an `Out of string space` error may occur if you initially assign a small number of characters to a string variable and later attempt to assign a very large number of characters to the variable. If you encounter this problem, you can avoid it by initially assigning a longer string to the variable.

Arrays

Array indexes must be in the range −32,768 to 32,767.

Visual Basic supports huge arrays (arrays containing data that exceeds 64K in size). The total amount of data in an array can exceed 64K for any type of array except arrays of variable-length strings and arrays of objects (such as Form and Control). The maximum size of a huge array is:

- 64 MB in Windows enhanced mode
- 1 MB in Windows standard mode

Huge arrays of the following types use more memory than indicated by multiplying the number of elements by the memory used by each element:

- Arrays of fixed-length strings or user-defined types, when the size of the fixed-length string or user-defined type is not a power of 2
- **Variant** arrays
- Arrays of user-defined types of any size containing **Variant** or variable-length strings as elements

Huge arrays of these types use at least 2 extra bytes at the beginning of each 64K segment used by the array. The number of extra bytes exceeds 2 by an amount that depends on the size of a single element in the array but which never exceeds the size of one element in the array. For example, a huge array of variants uses 16 extra bytes per segment.

User-Defined Types

No variable of a user-defined type can exceed 64K, though the sum of variable-length strings in a user-defined type may exceed 64K (variable-length strings occupy only 6 bytes each in the user-defined type; the actual contents of a string are stored separately). User-defined types can be defined in terms of other user-defined types, but the total aggregate size of the types cannot exceed 64K.

User-defined types can contain arrays of variable-length strings, but the total size of the elements in the array cannot exceed 64K. Moreover, each element of an array of variable-length strings in a user-defined type uses 2 bytes in the data segment used to store the string data. Thus, an array with a large number of elements cannot store as much string data. For example, a 16,384-element array of variable-length strings within a user-defined type can store only 32K of string data (the other 32K is consumed by the elements themselves: 16,384 elements at 2 bytes each).

Stack Space

Each application uses a single stack limited to 20K. The 20K size cannot be changed. Notice that an `Out of stack space` error can easily occur if your program performs uncontrolled recursion, such as a cascading event.

Arguments and local variables in procedures take up stack space at run time. Global, module-level, and static variables do not take up stack space because they are allocated in the data segment for forms or modules.

Visual Basic itself uses some of the stack for its own purposes, such as storing intermediate values when evaluating expressions, so the stack space available to your application is always less than 20K. This "overhead" is more significant when you are running your application in the Visual Basic development environment, so your application has more of the 20K stack space available to it when it is running as a finished .EXE. Also, any DLL procedures you call use this stack while they are executing.

For tips on conserving stack space, see Chapter 11, "Optimizing Your Application for Size and Speed."

Windows Limitations

Some limitations on Visual Basic, and the applications you create with it, are imposed by Microsoft Windows. These limitations may change when you install a different version of Windows.

Windows Resources

Every open window uses some "system resources" (data areas used by Microsoft Windows). If you run out of system resources, an `Out of memory` error occurs. You can check the percentage of system resources remaining by choosing About from the Help menu in the Program Manager or File Manager. To reclaim system resources, close windows (such as open form and code windows, as well as windows in other applications) and quit running applications.

RUN= and SHELL= Settings

Both the RUN= setting in WIN.INI and SHELL= in SYSTEM.INI support applications created with Visual Basic.

- SHELL= works only under Windows 3.1 and later
- RUN= works under Windows 3.0 and later

Multiple-Document Interface (MDI) Applications

For forms with the MDIChild property set to **True**, the BorderStyle, CommandBox, MinButton, and MaxButton properties are fully functional only in Windows 3.1. These properties work differently in Windows 3.0. For more information, see Chapter 14, "Multiple-Document Interface (MDI) Applications."

The Debug Version of Microsoft Windows

If you run Visual Basic on the debug version of Microsoft Windows provided with the Microsoft Windows SDK, all properties, including List and Text properties, go into a single segment, up to 64K per form or module. Other memory management limits may also differ under the debug version of Microsoft Windows.

APPENDIX E

Microsoft Product Support Services

Microsoft offers a variety of support options to help you get the most from Visual Basic. This appendix summarizes these options.

If you have a question about Visual Basic, first look in the printed product documentation, or consult Help. If you cannot find the answer, contact Microsoft Product Support Services.

Outside the United States, contact Microsoft Product Support Services at the Microsoft subsidiary office that serves your area. For information about Microsoft subsidiary offices, see the section "Support Services Worldwide" later in this appendix.

Product Support Within the United States

You can obtain product support in several ways, and locate additional training and consultation services in your area. You have the following means of obtaining product support:

Call Microsoft FastTips You can hear recorded responses to common questions about Microsoft Visual Basic. You can also order technical notes that will be sent to your fax machine. FastTips is available seven days a week, 24 hours a day.

- For assistance with Microsoft Visual Basic, dial (206) 646-5107.

You can use the following keys on your touch-tone telephone after you reach FastTips:

- To advance to the next message, press *.
- To repeat the current message, press 7.
- To return to the beginning of FastTips, press #.

Use the Microsoft Forums on CompuServe Microsoft Product Support Services is available on several CompuServe® forums. For an introductory CompuServe membership kit specifically for Microsoft users, dial (800) 848-8199 and ask for operator 230. If you are already a CompuServe member, type **GO MSBASIC** at any ! prompt.

Use the Microsoft Download Service Access technical notes and supplementary files covering common Microsoft Visual Basic product support issues via modem on the Microsoft Download Service at (206) 936-6735. This service is available 24 hours a day, seven days a week.

Call Microsoft Product Support Services You can reach Microsoft Product Support Services between 6:00 A.M. and 6:00 P.M. Pacific time, Monday through Friday.

- For setup and installation assistance with Microsoft Visual Basic, dial (206) 646-5105. This service is designed to get you up and running quickly with Visual Basic.
- For "how-to" assistance and in-depth support for advanced programming issues, call Microsoft OnCall for Visual Basic at (900) 896-9876. The rate is $2 per minute. You can also obtain support by calling our credit card line for Visual Basic at the rate of $20 per call. To use this service, call (206) 646-5106. Please have your Visa, MasterCard, or American Express card ready.

When you call, you should be at your computer and have the appropriate product documentation at hand. Be prepared to give the following information:

- The version of Microsoft Visual Basic you are using.
- The type of hardware you are using, including network hardware, if applicable.
- The operating system you are using.
- The exact wording of any messages that appeared on your screen.
- A description of what happened and what you were trying to do when the problem occurred.
- A description of how you tried to solve the problem.

Microsoft TDD/TT (Text Telephone) Support Microsoft Product Support Services is available for the deaf and hard of hearing. Using a special TDD/TT modem, dial (206) 635-4948. Call between 6:00 A.M. and 6:00 P.M. Pacific time, Monday through Friday.

Note Microsoft's support services are subject to Microsoft's prices, terms, and conditions in place in each country at the time the services are used.

Getting Product Training and Consultation Services

The Microsoft Solution Providers program comprises strategic independent vendors who specialize in software development and customization, systems integration, consulting, and training. The program offers a new way for value-added resellers, systems integrators, trainers, and consultants to use Microsoft technology to deliver support, training, information, and educational opportunities. For more information about the program or the Microsoft Solution Provider nearest to you, please call (800) 227-4679 from 6:30 A.M. to 5:30 P.M. (Pacific standard time).

Support Services Worldwide

If you are outside the United States and have a question about Microsoft Visual Basic, Microsoft offers a variety of no-charge and fee-based support options. To solve your problem, you can:

- Consult this manual's index and other printed product documentation.
- Check context-sensitive online Help, available from the Help menu.
- Check the README files that come with your product disks. These files provide general information that became available after the books in the product package were published.
- Consult electronic options such as CompuServe forums or bulletin board systems, if available.

If you cannot find a solution, you can receive information on how to obtain product support by contacting the Microsoft subsidiary office that serves your country.

Note Microsoft's support services are subject to Microsoft's prices, terms, and conditions in place in each country at the time the services are used.

Calling a Microsoft Subsidiary Office

When you call, you should be at your computer with Microsoft Visual Basic running and the product documentation at hand. Have your file open and be prepared to give the following information:

- The version of Microsoft Visual Basic you are using.
- The type of hardware you are using, including network hardware, if applicable.
- The operating system you are using.
- The exact wording of any messages that appeared on your screen.
- A description of what happened and what you were trying to do when the problem occurred.
- A description of how you tried to solve the problem.

Microsoft subsidiary offices and the countries they serve are listed in the following table.

Area	Telephone numbers
Argentina	Microsoft de Argentina S.A. Phone: (54) (1) 814-0356 Fax: (54) (1) 814-0372
Australia	Microsoft Pty. Ltd. Phone: (61) (02) 870-2200 Fax: (02) 805-1108 Bulletin Board Service: (61) (02) 870-2348 Technical Support: (61) (02) 870-2131 Sales Information Centre: (02) 870-2100
Austria	Microsoft Ges.m.b.H. Phone: 0222 - 68 76 07 Fax: 0222 - 68 16 2710 Information: 0660 - 6520 Prices, updates, etc.: 0660 - 6520 CompuServe: GO MSEURO (Microsoft Central Europe) Technical Support, Visual Basic: 0660 - 6516
Baltic States	See Germany

Area	Telephone numbers
Belgium	Microsoft NV Phone: 02-7322590 Fax: 02-7351609 Technical Support Bulletin Board Service: 02-7350045 (1200/2400/ 9600 baud, 8 bits, no parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: 02-5133274 (English speaking) Technical Support: 02-5023432 (French speaking) Technical Support: 02-5132268 Technical Support Fax: (31) 2503-24304
Bermuda	See Venezuela
Bolivia	See Argentina
Brazil	Microsoft Informatica Ltda. Phone: (55) (11) 530-4455 Fax: (55) (11) 240-2205 Technical Support Phone: (55) (11) 533-2922 Technical Support Fax: (55) (11) 241-1157 Technical Support Bulletin Board Service: (55) (11) 543-9257
Canada	Microsoft Canada Inc. Phone: 1 (416) 568-0434 Fax: 1 (416) 568-4689 Technical Support Phone: 1 (416) 568-3503 Technical Support Facsimile: 1 (416) 568-4689 Technical Support Bulletin Board Service: 1 (416) 507-3022 Telecommunication Device for the Deaf (TDD) 1 (416) 568-9641
Caribbean Countries	See Venezuela
Central America	See Venezuela
Chile	Microsoft Chile S.A. Tel: 56 2 218 5771 Fax: 56 2 218 5747
Colombia	Microsoft Colombia Tel: (571) 618 2245 Soporte Tecnico: (571) 618 2255 Fax: (571) 618 2269
Denmark	Microsoft Denmark AS Phone: (45) (44) 89 01 00 Fax: (45) (44) 68 55 10
Ecuador	See Venezuela

Area	Telephone numbers
England	Microsoft Limited Phone: (44) (734) 270000 Fax: (44) (734) 270002 Upgrades: (44) (81) 893-8000 Technical Support: Main Line (All Products): (44) (734) 271000 Database Direct Support Line: (44) (734) 271126 OnLine Service Assistance: (44) (734) 270374 Bulletin Board Service: (44) (734) 270065 (2400 baud) Fax Information Service: (44) (734) 270080
Finland	Microsoft OY Phone: (358) (0) 525 501 Fax: (358) (0) 522 955
France	Microsoft France Phone: (33) (1) 69-86-46-46 Telex: MSPARIS 604322F Fax: (33) (1) 64-46-06-60 Technical Support Phone: (33) (1) 69-86-10-20 Technical Support Fax: (33) (1) 69-28-00-28
French Polynesia	See France
Greece	Microsoft Hellas, S.A. Phone: (30) (1) 6896 663 Fax: (30) (1) 6896 662
Germany	Microsoft GmbH Phone: 089 - 3176-0 Telex: (17) 89 83 28 MS GMBH D Fax: 089 - 3176-1000 Information: 089 - 3176 1199 Prices, updates, etc.: 089 - 3176 1199 Bulletin Board, Device Drivers, Tech Notes: Btx: microsoft# or *610808000# CompuServe: GO MSEURO (Microsoft Central Europe) Technical Support, Visual Basic: 089 - 3176 - 1151
Hong Kong	Microsoft Hong Kong Limited Technical Support: (852) 804-4222 Fax: (852) 560-2217
Ireland	See England
Israel	Microsoft Israel Ltd. Phone: 972-3-575-7034 Fax: 972-3-575-7065

Area	Telephone numbers
Italy	Microsoft SpA Phone: (39) (2) 269121 Telex: 340321 I Fax: (39) (2) 21072020 Customer Service (Prices, New Products Info, Products Literature): (39) (2) 26901359 Bulletin Board: (39) (2) 21072051 Technical Support, Visual Basic: (39) (2) 26901364
Japan	Microsoft Company Ltd. Phone: (81) (3) 5454-8025 Fax: (81) (3) 5454-7972 Visual Basic: (81) (3) 5454 2352 PSS Technical Support Fax: (81) (3) 5454-7955 Customer Service (Version Upgrade/Registration): Phone: (81) (3) 5454 2305 Fax: (81) (3) 5454-7952 Channel Marketing (Pre-Sales Product Support): Information Center Phone: (81) (3) 5454-2300 Fax: (81) (3) 5454 7951
Korea	Microsoft CH Phone: (82) (2) 552-9505 Fax: (82) (2) 555-1724 Technical Support: (82) (2) 563-9230
Liechtenstein	See Switzerland (German speaking)
Luxemburg	Microsoft NV Phone: (32) 2-7322590 Fax: (32) 2-7351609 Technical Support Bulletin Board Service: (31) 2503-34221 (1200/2400/ 9600 baud, 8 bits, no parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: (31) 2503-77877 (English speaking) Technical Support: (31) 2503-77853 (French speaking) Technical Support: (32) 2-5132268 Technical Support Fax: (31) 2503-24304
México	Microsoft México, S.A. de C.V. Phone: (52) (5) 325-0910 Fax: (52) (5) 280-7940 Technical Support: (52) (5) 325-0912 Sales: (52) (5) 325-0911

Area	Telephone numbers
Netherlands	Microsoft BV Phone: 02503-13181 Fax: 02503-37761 Technical Support Bulletin Board Service: 02503-34221 (1200/2400/ 9600 baud, 8 bits, no parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: 02503-77877 (English speaking) Technical Support: 02503-77853 Technical Support Fax: 02503-24304
New Zealand	Technology Link Centre Phone: 64 (9) 358-3724 Fax: 64 (9) 358-3726 Technical Support Applications: 64 (9) 357-5575
Northern Ireland	See England
Norway	Microsoft Norway AS Phone: (47) (2) 95 06 65 Fax: (47) (2) 95 06 64 Technical Support: (47) (2) 18 35 00
Papua New Guinea	See Australia
Paraguay	See Argentina
Peru	See Venezuela
Portugal	MSFT, Lda. Phone: (351) 1 4412205 Fax: (351) 1 4412101
Puerto Rico	See Venezuela
Republic of China	Microsoft Taiwan Corp. Phone: (886) (2) 504-3122 Fax: (886) (2) 504-3121 Technical Support: (886) (2) 504-3188
Republic of Ireland	See England
Scotland	See England
Spain	Microsoft Iberica SRL Phone: (34) (1) 804-0000 Fax: (34) (1) 803-8310 Technical Support: (34) (1) 803-9960

Area	Telephone numbers
Sweden	Microsoft AB Phone: (46) (8) 752 56 00 Fax: (46) (8) 750 51 58 Technical Support: Applications: (46) (8) 752 68 50 Development and Network Products: (46) (8) 752 60 50 MS-DOS: (46) (071) 21 05 15 (SEK 4.55/min) Sales Support: (46) (8) 752 56 30 Bulletin Board Service: (46) (8) 750 47 42 Fax Information Service: (46) (8) 752 29 00
Switzerland	Microsoft AG Phone: 01 - 839 61 11 Fax: 01 - 831 08 69 Written Information: Phone: 155 59 00 Fax: 064 - 224294 Microsoft Info-Service, Postfach, 8099 Zürich Prices, updates, etc.: 01/839 61 11 CompuServe: GO MSEURO(Microsoft Central Europe) Technical Support (German speaking) Visual Basic: 01 - 342 - 4086 Technical Support (French speaking): 022 - 738 96 88
Uruguay	See Argentina
Venezuela	Corporation MS 90 de Venezuela S.A. Phone: 0058.2.914739 Fax: 0058.2.923835
Wales	See England

APPENDIX F

Accessibility for People with Disabilities

Microsoft is committed to making its products and services easier for everyone to use. This appendix provides information about the following products and services that make Microsoft Windows more accessible for people with disabilities:

- Microsoft support services for people who are deaf or hard-of-hearing
- Access Pack for Microsoft Windows, a software utility that makes using Microsoft Windows easier for people with motion or hearing disabilities
- Keyboard layouts designed for people who type with only one hand or a wand
- Microsoft software documentation on audio cassettes and floppy disks
- Products available for people who are blind or have low vision
- Hints for customizing Microsoft Windows
- Information about other products and services for people with disabilities

Note The information in this section applies only to Windows users in the United States. If you are outside the United States, your Windows package contains a subsidiary information card listing product support telephone numbers and addresses. You can contact your subsidiary to find out whether the type of products and services described in this appendix are available in your area.

Microsoft Support Services for People Who Are Deaf or Hard-of-Hearing

Through a text telephone (TT or TDD) service, Microsoft provides people who are deaf or hard-of-hearing complete access to Microsoft's product and customer support services.

You can contact Microsoft Product Support Services using a text telephone by dialing (206) 635–4948 between 6:00 A.M. and 6:00 P.M. Pacific time. Microsoft's product support services are subject to Microsoft's prices, terms, and conditions in place at the time the service is used.

Access Pack for Microsoft Windows

Microsoft distributes Access Pack for Microsoft Windows, which provides people with movement or hearing disabilities better access to computers running Microsoft Windows. Access Pack for Microsoft Windows contains several features that do the following:

- Allow single-fingered typing of SHIFT, CTRL and ALT key combinations.
- Ignore accidental keystrokes.
- Adjust the rate at which a character is repeated when you hold down a key, or turn off character repeating entirely.
- Prevent extra characters if you unintentionally press a key more than once.
- Allow you to control the mouse cursor using the keyboard.
- Allow you to control the computer's keyboard and mouse using an alternate input device.
- Provide a visual cue when the computer makes sounds.

Access Pack for Microsoft Windows is included on the Microsoft Windows Driver Library in the file ACCESS.EXE. If you have a modem, you can download Microsoft Windows Driver Library components from network services, including the following:

- CompuServe
- GEnie™
- Microsoft OnLine
- Various user-group bulletin boards (such as the bulletin-board services on the Association of PC User Groups network)
- Microsoft Download Service (MSDL), which you can reach by calling (206) 936–MSDL (936–6735) any time except between 1:00 A.M. and 2:30 A.M. Pacific time. Use the following communications settings.

For this setting	Specify
Baud rate	1200, 2400, or 9600
Parity	None
Data bits	8
Stop bits	1

People within the United States who do not have a modem can order disks by calling Microsoft Product Support Services at (206) 637–7098 or (206) 635–4948 (text telephone).

Keyboard Layouts for Single-Handed Users

Microsoft distributes Dvorak keyboard layouts that make the most frequently typed characters on a keyboard more accessible to people who have difficulty using the standard "QWERTY" layout. There are three Dvorak layouts: one for two-handed users, one for people who type only with their left hand, and one for people who type only with their right hand. The left- or right-hand keyboard layouts can also be used by people who type with a single finger or a wand. You do not need to purchase any special equipment in order to use these features.

Microsoft Windows already supports the Dvorak keyboard layout, which can be useful for coping with or avoiding repetitive motion injuries associated with typing. The two layouts for people who type with only one hand are distributed as Microsoft Application Note GA0650. They are also contained in file GA0650.ZIP on most network services or GA0650.EXE on the Microsoft Download Service. For instructions on obtaining this application note, see the section "Access Pack for Microsoft Windows" earlier in this appendix.

Documentation on Audio Cassettes and Floppy Disks

People who cannot use printed documentation can obtain most of Microsoft's publications from Recording for the Blind, Inc. Recording for the Blind distributes these documents to registered members of their distribution service, either on audio cassettes or on floppy disks. Recording for the Blind's collection contains more than 80,000 titles, including Microsoft product documentation and books from Microsoft Press. You can contact Recording for the Blind at the following address:

Recording for the Blind, Inc.
20 Roszel Road
Princeton, NJ 08540
Phone: (800) 221–4792 Fax: (609) 987–8116

From outside the United States, you can contact Recording for the Blind at (609) 452–0606.

Products for People Who Are Blind or Have Low Vision

There are numerous products available to help people who are blind or have low vision use Microsoft Windows. For people with low vision, there are several screen enlargement utilities, and for people who cannot use visual information, there are screen readers that provide alternative output by synthesized voice or refreshable Braille displays. In addition, people with low vision can customize the Microsoft Windows display to suit their disabilities.

For more information on the various products available, see the section "Getting More Information" later in this appendix. For more information about customizing Microsoft Windows for people with low vision, see the section "Customizing Windows" later in this appendix.

Customizing Windows

There are many ways you can adjust the appearance and behavior of Microsoft Windows to suit varying eyesight and motor skills without requiring any additional software or hardware. These include ways to adjust the appearance as well as the behavior of the mouse and keyboard. The specific methods available depend on which operating system you are using. Application notes are available describing the specific methods available for each operating system.

For information relating to Windows 3.0, see Application Note GA0000. For information relating to Windows 3.1 or Windows for Workgroups 3.1, see Application Note GA0000. For information relating to Windows NT 3.1, see Application Note GA0000. For information on obtaining application notes, see the section "Access Pack for Microsoft Windows" earlier in this appendix.

Getting More Information

For more information on Microsoft products and services for people with disabilities, contact Microsoft Customer Sales and Service at (800) 426–9400 (voice) or (206) 635–4948 (text telephone).

The Trace R&D Center at the University of Wisconsin at Madison produces a book and a compact disc that describe products that help people with disabilities use computers. The book, titled *Trace ResourceBook,* provides descriptions and photographs of about 2,000 products. The compact disc, titled *CO-NET CD,* provides a database of more than 17,000 products and other information for people with disabilities. It is issued twice a year and should be available in many public libraries by early 1993.

You can contact the Trace R&D Center by using the following address or telephone numbers:

Trace R&D Center
S-151 Waisman Center
1500 Highland Avenue
Madison, WI 53705-2280
Voice telephone:(608) 263–2309
Text telephone: (608) 263–5408
Fax: (608) 262–8838

For general information and recommendations on how computers can help specific people, you should consult a trained evaluator who can best match the individual's needs with the available solutions.

If you are in the United States, you can obtain information about resources in your area by calling the National Information System, an information and referral center for people with disabilities, at the following address:

National Information System (NIS)
Center for Developmental Disabilities
University of South Carolina, Benson Bldg.
Columbia, SC 29208
Phone: (800) 922–9234 (voice/text telephone outside South Carolina)
(800) 922–1107 (voice/text telephone in South Carolina)
(803) 777–6222 (voice/text telephone outside the United States)
Fax: (800) 777–6058

This service is available only in the English language.

Index

Symbols

! (exclamation point)
  activating parts of files 552
  separating form and control references 137
  Single type-declaration character 165
" " (double quotation marks), formatting strings 160–161
(number sign)
  Double type-declaration character 165
  formatting numbers 160–161, 389
$ (dollar sign)
  formatting numbers 389
  String type-declaration character 165
% (percent sign), Integer type-declaration character 165
& (ampersand)
  access keys 40
  concatenating 158
  Long type-declaration character 165
  Menu access keys 80
  octal/hexadecimal formats 134
' (apostrophe), designating comments 134
+ (plus sign), concatenating 158
, (comma)
  declaring constants 178–179
  using with Print method 384, 387
. (period), separating form and control references 137
; (semicolon), using with Print method 384
= (equal sign), declaring constants 178–179
? (question mark)
  Print method shortcut key 228
  wildcards in file list boxes 429
@ (at sign), Currency type-declaration character 164–165
[ ] (brackets), using with reserved words 135–136
3D check box control 463
3D panel control 463

A

About dialog box, Text Editor application 93
Absolute movement, defined 329
Access keys
  assigning to menu control, Menu Design 85
  introduced 85
  menu, (&) ampersands 80
Access options
  Deny-Read options 460
  Deny-Write options 460
Access, shared, Exclusive property 460
Accessing
  databases with data control 453
  linked and embedded objects 551
Action argument 479–480
Action property
  creating objects at run time 535
  OLE controls 536
  OLE_CREATE_EMBED 538
  persistent data 541
  SAVE_TO_FILE 541
Action, canceling 480
Activating objects
  discussed 539
  in place 539
ActiveControl property
  and DDE 509
  determining current focus 65
  vs. ActiveForm property 316
ActiveForm property
  determining current focus 65
  specifying current form 303, 316
  vs. ActiveControl property 316
Add Data Access files, distributing applications 574
Add File menu command 116
Add OLE files 548, 545
Add Watch dialog box 99
Adding files to project 116
AddItem method
  adding grid rows 295–296
  code example, list boxes 56–57
  index argument 53
  ordering 53
  syntax 52
AddNew method
  aborting 466
  discussed 465, 472–473, 480
Administration, data 460
Advantages of Object Linking and Embedding (OLE) 543–544
ALARM.MAK, timer controls application 405
Alias keyword
  declaring nonstandard names 561–562
  and DLLs 558
Align property
  described 334
  and status bar 315
  and toolbar 313

ALT key
keyboard handling 419–420
testing status 276
Ampersand (&)
access keys 40, 76
concatenating 158
Long type-declaration character 165
Menu access keys 80
octal/hexadecimal formats 134
Angles, degrees to radians 349
Animation
BUTTERF.MAK application 23–26
code examples
longer animation 332
moving effects 368
moving within border of forms 330
simple animation 331
control arrays 332
effects 368
simple effects 330–333
toggling two pictures 330
Xor Pen, DrawMode setting 368
ANSI code
KeyDown, KeyUp events 421–422
Search Help for ANSI
API functions, online Help 5
Apostrophe ('), designating comments 134
App object, *See* Application object
AppActivate statement
sending keystrokes with DDE 522
APPDIR, SETUP1.FRM constants 587
AppendChunk method 476
Application
container 526
data control, using 454
Application development
See also OLE; Procedures; Projects
controls *See* Controls
End statement 126
event-driven applications 124–125
multiple modules
error-handling guidelines 246
overview 126–129
naming conventions 135–136
properties *See* Properties
startup 125
Stop statement 126
terminating 126
traditional applications 124–125
Unload statement 126
variables *See* Variables
Application directory, installing program files on user's machine 584
Application files
distributing 578
Application files *(continued)*
installing 577
Application names 495
Application object (App)
and DDE conversations 503
defined 206
EXEName property 503, 512
Search Help for App; Applications
Applications
creating
data, displaying and editing 453
from projects 112
online lessons 8
update information 453
data-aware, making 453
debugging, online lessons 8
distributing
compressing files 583
discussed 548, 573, 577
SETUP.LST 582
files 577
front-end, creating with OLE 523
installing 573
interoperability between 523
OLE, distributing 544
projects, creating from 112
sample *See* Sample applications
sharing data
discussed 547
Windows, earlier versions 547
user installation 573
APPNAME, SETUP1.FRM constants 587
Arcs
Circle method 350
code examples
pie chart with missing slice 351
simple arcs 351
drawing 348–351
Arguments
Action
canceling 480
discussed 479–480
ByVal keyword 169
data types available 168–169
DataError 475
default passing, DLLs 560
DLL uses *See* Declare statement
multiple data types 561
named, limitations in Visual Basic 555
passing by value 169
programming techniques 262
Save 479
user-defined data types 176–177
Arrange property, and Window menus 311

Arrays
See also .DLL
Boolean 58
control *See* Control arrays
control variable arrays
described 190–191
vs. control arrays 191–192
data types 170
declaring 171–172
defined 170
Dim statement 171
dynamic *See* Dynamic arrays
form variable arrays 190
global 171
Global statement 171
huge 175
index numbers 170
LBound function 170
limitations in Visual Basic 554
List property uses 54
local 171, 174
lower bound 170–172
manipulation using loops 172
Menu Control, creating 79
module-level 171
multidimensional 172–173
New keyword 190
object arrays, restrictions on 190
Option Base statement 171
programming techniques 259
Selected property uses 58
size
LBound function 555
UBound function 555
Static statement 171
To keyword 172
UBound function 170
upper bound 170–171
user-defined 176–178
Arrow keys, keyboard handling 419–420
ASCII
characters
detecting with KeyPress event 420
Search Help for ASCII
codes, embedded quotation marks 42
format databases 483
text format
default save format 116
Project files 113
Assigning Access keys 85
At sign (@), Currency type-declaration character 164–165
Attribute property 488
Auto-commit transactions 477
AutoActivate property 537–539
AUTOLOAD.MAK file
discussed 115
editing 121
environment options, setting 119
loading .VBX controls 524
modifying for MDI applications 299
Automatic links
See also DDE; DDE links
bitmap graphics 502
creating 501
defined 496
destination links 497–498
user-defined 509–513
adding Paste Link command 509–510
Copy command 511–513
viewing through Clipboard 509
AutoRedraw property
changing at run time 361
default value 359
defined 358
graphics and resizing windows 360
impact of 362
limited to picture box controls 360
normal layering uses 363–364
PrintForm method 404
programming techniques 257–258, 261–262
using 358–361
AutoSize property
described 333
picture box controls 325
resizing controls 333
sizing label controls 38–39
AutoVerbMenu property 539
Axis, determining for ellipses 347

B

B option, drawing boxes with Line method 344
BackColor property
defined 358–369
setting with PSet method 341
Search Help for BackColor
Background colors, Environment options 120
Background palette, defined 376
Background processing
DoEvents function 417
suspending vs. terminating 415–416
timing 411
BACKSPACE key, keyboard handling 420
.BAS (code module) files
See also Code modules
discussed 114
Basics, transferring from other 4
Beginning of File (BOF) 468, 470

BeginTrans command, transactions processing 477–478
BIBLIO.MAK sample database project 454
BIBLIO.MDB sample database 454
Bibliography of current books on databases 454
Binary data
  (.FRX) files 112
  Picture boxes 466
Binary-access files
  accessing records 451
  defined 436
  functions, listed 446
Bit-field argument
  defined 266
  mouse button status 271
BitEdit application, user-defined bitmaps 375
Bitmap fonts, defined 378
Bitmap graphics
  See also Grid control graphics
  .BMP (bitmap graphics) files 321
  clearing controls 25
  .DIB (bitmap graphics) files 321
  image controls 321
  loading 28
  printing
    considerations 394
    to printer 401
Bitmaps 373–375
Bits
  Button argument 274
  Shift argument 276
  testing mouse button status 274, 276
.BMP (bitmap graphics) files 321, 325
BOF (Beginning of File), moving to 468, 470
BOF/EOF property 468
Bookmark property 471
Bookmarks, moving to specific records 471
BorderColor property *Search Help for* BorderColor
BorderStyle property
  code example 367
  described 366
  MDI child forms 300
  with shape controls vs. other controls 367
  Search Help for BorderStyle
BorderStyle standard dialog box property 98
BorderWidth property, outline thickness 366
Bound controls 457, 462–464
Boxes
  code example 344
  drawing 344–345
  with FillColor settings 344
Brackets ([]), using with reserved words 135–136
Break mode, debugging
  adding watch expressions 223–224
  assigning values 230
  defined 212
Break mode, debugging *(continued)*
  displaying variables 214
  idle loops 221
  overview 213
  setting breakpoints 215–216
  setting execution order 219–220
  Stop statement 214, 217
  switching to 214
  title bar display 211
  toolbar buttons 213
Breakpoint statement *See* Debugging
Breakpoints, debugging 210, 215–217
Btrieve databases
  accessing 458
  discussed 453, 459
  external 481, 483
  opening 458
Build Disks command
  distributing applications 574
  distribution disks 574
BUTTERF.MAK, simple animation application 23–26
Button argument
  defined 266
  mouse button status 271
  using with mouse events 271–274
  valid binary values
    MouseDown event 272
    MouseMove event 273
    MouseUp event 272
Buttons, data control 457
ByVal keyword
  See also Declare statement
  and DLLs 558
  passing arguments by value 169
  specifying data types 168

C

CALC.MAK, Calculator application 31, 69–70
Call statement, Sub procedure 131
CALLDLLS.MAK, DLL application 557
Calling procedures, programming techniques 257, 260, 262
Calls list
  defined 240
  error handling 228–243
  order of procedure execution 242
  order of procedure resumption 242
  unexpected procedure execution 243
  Search Help for Calls
Calls, debugging 210, 220–221
Cancel button dialog boxes, setting 100
Cancel button, default 36
Cancel dialog boxes, setting 100

Case → see Select Case
Cancel property
changing font display 379
and command button controls 36
creating access keys 40
and label controls 38–39
Canceling Action 480
Carriage-return character, terminating cell data by rows 289
Cascading event, defined 125
Cells, grid control *See* Grid control cells
Centimeters, scale measurement 353
CF_BOTH (&H3&) Flags property 107
CF_PRINTERFONTS (&H2&) Flags property 107
CF_SCREENFONTS (&H1&) Flags property 107
Change event
and file-system controls 431–433
directory list boxes 433
drive list boxes 432
scroll bars 61
selecting option buttons 47
Changed property 479
Changes to data, saving 457
Characters, defined 353
ChDir statement
and directory list boxes 428
executable files and 428
ChDrive statement, and drive list boxes 427
Check box control
capturing user input 43–45
code example 44–45
discussed 463
naming conventions 35–36
property settings (list) 44
selecting multiple boxes 43
Toolbox icon 33
Check marks, menus 86
Check syntax, Environment options 120
Checked property, Menu Controls 87–88
Child forms 298–300, 304–305
Choose Color dialog boxes, displaying 103
Choose Font dialog box 103, 107
Chr function, embedded quotation marks 42–43
Circle method
code examples, drawing 269, 346–347, 350
defined 338
drawing 343, 347–348
FillColor property 369
negative arguments and arcs 350
positioning circles 345
printer output 393
radius argument 347
relative positioning of objects 343
setting line width 366
Step keyword 345
syntax 345, 347, 349
Circles
See also Circle method
arcs 348
clearing area 340
code example 346
defining drawing area 345
drawing 345–346
drawing with shape controls 322
ellipses 347
FillColor settings 369
fundamentals 339
ovals *See* Ellipses
relative positioning of objects 363
using custom scales 346
Circular references 180
Class parameter, OLE Automation 552
Class property
creating objects 535
creating objects at run time 535
Clear method
clearing lists 53
code example, list boxes 55–57
programming techniques 257
Clearing
See also Clear method; Cls method
drawing area 340
Click events
procedures for Menu Controls 81
selecting option buttons 47
with combo boxes 51
with command button controls 36–37
with image controls 37
Click-A-Line application 267–268
Clip art, source of pictures 325
Clip property
cell ranges, filling 289
formatting requirements 289
Clipboard
and DDE 510–513
copying/pasting Color palettes 375
images, pasting 466
object 81
selecting text 405–406
transferring data with 547
viewing DDE conversations 509
Clipboard objects
Clear method 407
copying pictures 324
Cut, Copy, and Paste 407–408
defined 206, 406
GetData method 408–409
GetText method *See* GetText method
multiple formats, coexisting 408

Clipboard objects *(continued)*
overview 406–411
SetData method 408–409
SetText method *See* SetText method
Search Help for Clipboard
ClipControls property
default value 361
defined 361
impact of 362
Line method 361
normal layering uses 363–364
Paint event 361
using 361–362
Clipping region, defined 361
Close method
discussed 473, 476
OLE Automation 554
Close statement
file access uses 438, 443, 446
sequential-access files 443
Closing databases, Data Controls 457
Closing Recordsets 476
Cls method
clearing drawing area 340
defined 339
printing 385
programming techniques 260
syntax 341
using with AutoRedraw property 363
CMDIALOG.VBX (common dialog) custom control
discussed 114
distributing application files 579
Code
examples *See* Code examples
loading text files as 118
manipulating records with 467
menu control arrays, writing 81
menu controls, writing 81
modules *See* Code modules
sample 7
saving as text 117
saving as text file 118
setting properties in 535
source, managing 116
window *See* Code window
Code examples
Alarm Clock 414–416
available fonts, listed 382
check boxes 44–45
colors
setting pixel by pixel 340–341
setting with RGB function 319
concatenation vs. addition 160
control arrays
sharing properties 337
Code examples *(continued)*
unloading 338
controls, adding/hiding 336
creating custom scales 356
Cut, Copy, and Paste 409–410
defining drawing area 354
detecting specific keys 420, 423–424
determining data format 410–411
drawing
arcs 349–351
boxes 345
circles 346
ellipses 348
pie chart with missing slice 350
polygons 342
randomly scattered lines 344
solid, dashed, dotted lines 367
varied line width 366
entering uppercase text 420
error handling
delayed handling 251
disabling error traps 253
disk I/O procedures 247–248
disk-related errors 249–250
division by zero 242
file name selection by user 242–243
invalid drive selection 237
formatting time 392
hiding/showing controls 335
identifying list selections 54
idle loops 417
list boxes 55–57
longer animation 332
mouse events
activating/deactivating drawing 270
centering controls 267, 282
dragging and inverting colors 281
dragging controls 279–280
drawing lines 268–269, 275
invisible controls 280
moving controls 267, 280, 282–283
testing Shift argument 276
moving controls 327
multi-column list boxes 57–59
OLE2DEMO.MAK 523
placing text on Clipboard 407
positioning objects 354
prime numbers 418
printing
end of documents 403
formatted output 403
multiple forms with Printer object 399
multiple items per line 384
multiple-page documents 403
one item per line 384

Code examples *(continued)*
specific locations 387
tables, custom and default tabs 387–388
provided with Microsoft Visual Basic
See Sample applications
resizing command buttons 334
running from Help 11
selecting option buttons 47
setting colors with scroll bars 61–64
simple animation 331
toggling Boolean properties 335
word wrapping, text boxes 40–41
Code modules
ASCII text format 116
assignment statement, defined 136
comments 134
contents 123
control structures, listed 140–141
creating 115
data types 140
decision structures 141–144
declaring global variables 152
described 126
elements 127
event procedures *See* Event procedures
exiting control structures 148
exiting procedures 148
form-specific procedures 127
general procedures *See* Procedures
multiple commands per line 134
nested controls 147
numbers *See* Numbers
private procedures
defined 132
Private keyword 133
uses 132–133
Search Help for Private
programming techniques 260
public procedures
defined 132
restrictions 133
uses 132–133
saving as text 116
text
replacing as 117
saving as 116–117
using properties 136–139
variables 139–140
Code window
defined 21
Object box, defined 22
online Help 10
opening 21
opening in break mode 214
Print method 222
Code window *(continued)*
Procedure list box, defined 22
switching between procedures 28–29
Search Help for Code window
ColAlignment property, aligning text, non-fixed columns 290
CollatingOrder database properties 488
Collections
containing strings, limitations in Visual Basic 556
Count property 556
defined 555
discussed 485
Fields 474, 486–487, 489
index 556
Indexes 487
limitations in Visual Basic 555
numeric, iterating 556
TableDefs 485
Color
argument, drawing lines 365
dialog box
displaying 106
using 106
online lessons for using 8
palettes
256 colors 372
copying and pasting 375
defined 372
dialog boxes 106
editing bitmaps with 375
increasing with logical palette 373
static colors 372
system palette limitations 375
VGA default 372
vs. Color Palette (Window menu) 372
z-order effects, described 376
property
boxes 107
dialog boxes 106
Colors
adapting for monitors 373
changing pixel values 369
code examples
inverting while dragging 281
setting by pixels 340–341
setting with RGB function 319
dithering 372
FillColor settings 344
foreground, PSet method 340
properties, listed 369
setting
pixel by pixel 402
using hexadecimal numbers 370
via PSet 340
with constants 370
with QBColor function 319

Colors *(continued)*
with RGB function 319
supported 371
supported for metafiles 371
system palette, defined 372
using BackColor, ForeColor properties 358
values 318
working with 370–376
Search Help for Color palette
Cols property, defined 286
Columns property, list box values 57
Columns, relational database objects 481
ColWidth property, defined 287
Combo box controls
accessing list items 54
adding items 52–53
AddItem method *See* AddItem method
capturing user input 49–57
Clear method *See* Clear method
counting items 55–56
creating access keys 40
defined 49
display options 49
drop-down 50
drop-down list boxes 51
identifying selections 54
List property 54
ListCount property 55–57
ListIndex property 54–55
mouse event recognition 266
naming conventions 35–36
position of list items 54–55
RemoveItem method *See* RemoveItem method
removing items 53
selecting items 50
selecting text 405–406
simple 50
Sorted property *See* Sorted property
sorting list 53
styles, listed 50–51
Text property 54
Toolbox icon 33–34
unsupported events, listed 51
when to use 51
Comma (,)
declaring constants 179
using with Print method 384, 387
Command button control
actions described 37
Caption property 37
Click event 37–38
code examples
printing to printer 400
resizing 334
default controls 37
Command button control *(continued)*
naming conventions 35–36
printing image to printer 397
selecting 37
toolbar uses 313
Toolbox icon 33
vs. image controls 38
Command buttons
adding to dialog boxes 98–99
placing in dialog boxes 98–99
Command function
debugging and command-line arguments 232–233
Search Help for Command
debugging 232–233
Command-line arguments
Project options 120
commandline variable 113
Commands
BeginTrans 477–478
CommitTrans 477
Rollback 477–478
COMMDLG.DLL, distributing application files 579
Comments
apostrophe ('), designating 134
documenting code 134
Committing transactions 477
CommitTrans command, transactions processing 477
Common Dialog Custom Control (CMDIALOG.VBX)
discussed 114
using 103
COMPOBJ.DLL
distributed with applications 548
distributed with OLE applications 544
distributing application files 579
COMPRESS.EXE, compressing files *See* Compressing files
Compressing application files 577
Compressing files
distributing applications 573
not compressible
(list) 583
SETUP.EXE not compressible 583
VER.DLL not compressible 583
WSETUP.LST not compressible 583
Setup 573, 583
Configuring printers 108
Connect property
databases 458
discussed 459, 467, 488
RecordSource property 462
CONSTANT.TXT file
color constants 370
constants 479
DDE constants 501, 510
ENTER, LEAVE, OVER, defined 281
keyboard constants 421–422

CONSTANT.TXT file *(continued)*
OLE constants 535
Print dialog boxes 109
sample constants 92
selecting data formats 409, 411
setting default system colors 370
specifying color values 318
symbolic constants 179
system colors 371
Window menu constants 311
Constants
(list) 479
.BAS (code module) files 114
circular references 180
CONSTANT.TXT 479
cycles, defined 18
DATA_ACTION constants 479
declaring 179
.FRM (form) files 114
programming techniques 260
SETUP1.FRM 587
symbolic 178–180
validation 479
Container Application 526
See also OLE
Container forms *See* MDI
Containers, accessing OLE objects 548
Contents, Help 9
Context-sensitive Help 10
Control arrays
adding controls at run time 71–73
adding/removing 335–338
Calculator application 69–70
changing at run time 71
code examples
adding/hiding controls 336
hiding/showing controls 335
longer animation 332
sharing properties 337
unloading arrays 338
combining events 63
creating 70–71
defined 68, 331
function restrictions 170
Index property *See* Index property
Load statement *See* Load statement
menu control arrays *See* Menu control arrays
Name property *See* Name property
shared properties 336
Unload statement *See* Unload statement
vs. control variable arrays 191–192
Control bar *See* Toolbar
Control Development Guide, online Help 5
ControlBox property, MDI child forms 300
ControlBox standard dialog box property 98
Controls
See also specific control
accessing values through properties 138–139
adding/removing 335–338
assigning drag icons 278
automatically dragging 278
binding 453
BorderStyle property uses 367
bound 453
types of 463
working with 462
capturing user input with 43–64
changing font display
See also Fonts
Caption property 379
Print method 379
Text property 379
changing name 128
changing run-time state 67–68
changing size and shape 333
child, defined 46–47
code examples
adding/hiding 336
hiding/showing 335
moving 327
resizing command buttons 334
command buttons as default 37
container, defined 46–47
creating access keys 40
creating control arrays 70–71
current focus 64–65
Data
See also Data Control
discussed 457
data-aware 453–454, 462–463
detecting keyboard input 419
disabling dialog box 101
dragging and dropping 276–283
drawing on forms 13
Enabled property *See* Enabled property
file-system *See* File-system controls
form-level handling 423–424
hiding/showing 334–335
Image 466
moving
and resizing 318
dynamically 326–330
Left, Top properties 326
speed and smoothness 330
using mouse events 266, 282–283
within containers 46–47
naming conventions 34–36
object variables *See* Object variables
online lessons 8
Parent property, defined 45–47

Controls *(continued)*
printing image to printer 397
programming techniques 257–258, 261–262
properties, listed 138
resizing 333–334
setting focus 408
submenu 83
tab order *See* Tab order
Toolbox icons, listed 32–34
transferring between forms 137
using property values directly 138–139
Visible property *See* Visible property
Controls collection
See also Forms collection
overview 205–206
syntax 205
CONTROLS.MAK
command button controls 37–38
controls application 32
image controls 38
label controls 39–40
Conversations *See* DDE
Conversion
converting degrees to radians 349
Variant variables 159
Search Help for conversion
Coordinate systems
absolute movement 328
changing default 386
code examples
creating custom scales 356
moving line controls 327
relative movement 330
custom scales 353–357
defaults 318
defined 317
defining 354
mouse coordinates
defined 266
initial 282–283
moving line controls 327
moving/resizing controls 318
negative numbers 355
object dimensions 355
passing coordinates 342–343
positioning circles 345
printing on forms and picture boxes 385
relative movement 328
scale defaults 352
setting standard scales 353
specific print locations 386–387
twip, inch, and centimeter equivalents 386
Search Help for coordinate systems; ScaleMode
Copies, specifying, Print dialog box 108
CopyFile function, Setup Program 586–587
Copying
Clipboard object 407–408
Color palettes 375
copy operations 466
creating control arrays 70–71
DDE uses 511–513
pictures between objects 324
pictures for import 323
pictures from Clipboard object 324
CopyPen, DrawMode setting described 368
Count property
collections 556
discussed 485
COUNTOLE.DLL
distributed with applications 548
distributed with OLE applications 544
distributing application files 579
Create Embedded Object command 533
Create Link command 533
CreateObject function 526, 548
CreateProgManGroup function, Setup Program 586, 588
CreateProgManItem function, Setup Program 586, 588
Creating Projects 115
CRW.HLP, online Help 5
Crystal Reports, online Help 5
CTL3D.DLL, distributing application files 579
CTRL key
keyboard handling 419–420
testing status 276
CTRLREF.HLP, online Help 5
Currency data type 164
Current record
defined 468
editing 473
field values, editing 474
managing 468
Current statement *See* Debugging
CurrentX, CurrentY properties
code example, TextHeight, TextWidth methods 387
passing coordinates 342–343
Printer object 395
printing 385
syntax 386
TextHeight, TextWidth methods 386–387
Custom colors, dialog boxes 106
Custom Control (.VBX) files
adding to project 114, 524
default 524
defined 524
distributing application files 579
grid controls *See* Grid controls
GRID.VBX, LOAN.EXE sample program 586
Microsoft Visual Basic Professional Edition 114
MSOLE2.VBX 524
OLE controls See OLE

Custom Control (.VBX) files *(continued)*
Projects, adding to 114
third-party developers' 114
Custom Control Reference, online Help 5
Custom dialog boxes
creating 96
display options 101
displaying 101
Custom scales
code example 356
creating 353–357
creating with Scale method 358
defining drawing area 354
defining upper-left corner 354
drawing circles 346, 347
Custom solutions, OLE Automation 547
Customized look, Setup Program 577
Customizing existing dialog boxes 97
Cut command, moving controls 46
Cut operations 466
Cutting
Clipboard object 407–408
selecting text 405–406
Cycle, defined 180

D

Dashed lines 366–367
Data
accessing
faster 460
OLE objects 547
administration 460
binary Picture boxes 466
changes, saving changes 457
constants 479
displaying, from other applications 523
edit 539
errors handling 475
exclusive access 460
fields, handling large 476
linking and embedding 524
linking to, in files 531
linking, Paste Special 532
manipulating
discussed 457
from other applications 523
OLE objects 547
Recordset Objects 460
passing to fields in bound controls 457
persistent
discussed 541
embedded objects 532
play or edit 539
Data *(continued)*
reading into OLE control 542
saving
from OLE objects 541
Update method 465
sharing
between applications 547
OLE automation 547
transferring with Clipboard 547
updating 481
Data constants 479
Data Control
accessing databases with 453
applications, using in 454
binding text boxes to 454
buttons 457
capturing error events 457
closing databases 457
data events, managed by 463
data fields, passing to bound controls 457
database tables, opening 457
databases, closing 457
databases, updating 457
error events, capturing 457
opening database tables 457
opening recordsets 457
Recordset objects 460
recordsets, opening 457
RecordSource property 467
Refresh method 467
updating databases 457
using 457
Data errors, handling 475
Data events, managed by Data Control 463
Data fields
handling large 476
large, handling 476
passing data to, in bound controls 457
Data files, distributing application files 579
Data Manager sample database application
databases
creating 485
importing with 483
mapping 488
indexes, creating 485
mapping databases 488
tables, creating 485
Data property 544
Data types
See also .DLL
arguments *See* Arguments
arrays *See* Arrays
Currency 164
declaring 148–149
default 149

Data types *(continued)*
Dim statement 165
Double 164
function commands with 170, 178
Function procedures 132
Global statement 165
Integer 164
Long 164
Single 164
Static statement 165
String 164
Sub procedures 131
Type statement 176
user-defined 176–178
variables 140
Variant *See* Variant variables
Data Updates, controlling 476
Data-aware applications, creating 453
Data-aware controls
built-in 463
discussed 453–454, 462
third-party 463
Toolbox 463
Data-aware properties 463
Database application, sample (DATAMGR.EXE) 454
Database design 481
Database engine, Microsoft Access 453
Database Objects 481, 484–485, 488
Database project, sample (BIBLIO.MAK) 454
Database tables, opening with the Data Control 457
DatabaseName property
databases 458–459
discussed 454, 460, 467
Databases
access, exclusive 460
accessing 458
adding records to 464
administration 460
applications which access, creating 453
ASCII format 483
BIBLIO.MDB sample database 454
bibliography of current books 454
Btrieve 453, 458–459
changes, validating 479
closing with Data Control 457
code, manipulating data with 467
collections 485
Connect property 459
connecting to local or remote 457
creating sample 454
Data Manager, creating with 485
data, using 465
dBASE 453, 458
dBASE III 459
dBASE IV 459
Databases *(continued)*
defining 484
deleting records 464, 466
describing 484
design 481
exclusive access 460
exclusive mode 458
external 481–483
fields defined 482
FoxPro 458–459
imported with Data Manager 483
indexes, defined 482
local, connecting 457
locating with DatabaseName property 459
mapping 483, 488
mapping tables 488
Microsoft Access 454, 458–462
Microsoft Access-format 483
Microsoft FoxPro 453, 458–459
names, specifying at design time 458
native format, Visual Basic 483
networked servers 459
new, opening 467
ODBC 459
opening 458, 460, 467–468
Paradox 453, 458–459
password 459
properties 462, 488
queries defined 482
ReadOnly property 462
records 464–467, 482
RecordSource list 460
Refresh method 460
relational model 483
remote 457–459
servers, remote networked 459
specifying names at design time 458
structure 457, 483
TableDefs collection 485
tables 462, 482, 488
transactions spanning 477
updating with Data Controls 457
validating changes to 479
viewing properties 488
Visual Basic native format 483
DataChanged data-aware properties 463, 466
DataError argument 475
DataField data-aware properties 454, 463–464
DATAMGR.EXE sample database application 454
DataSource data-aware properties 463–464
DataText property 544
Date/time
Control Panel settings 161
date/time format 162
IsDate function 162

Date/time *(continued)*
Variant variables 160–162
Search Help for Date
DateCreated property 488
Dates
formatting
international syntax 391–392
syntax 391
using named formats 389–390
Search Help for Now
DB_UPDATABLEFIELD bit, adding records 472
dBASE databases 453, 458–459, 481, 483
dBASE III databases 459
dBASE IV databases 459
DblClick event, combo boxes 51
DBNMP3.DLL, distributing application files 579
DDE (dynamic data exchange)
See also DDE links
AppActivate statement 522
application names, defined 495
application size considerations 504
conversations
defined 494
initiating 494–496
multiple 494
terminating 494, 504
defined 494
destinations
defined 494
initiating conversation role 494
label controls 494
link 497–498
picture box controls 494
text box controls 494
DoEvents statement 519–520
error handling 517–520
items
defined 495
effects of changing 495
LinkItem property 500
overview 496
launching other applications 513–514
other applications and 516–517
picture box controls 516
poking data 514
requesting data 515
sending commands 515
SendKeys statement 521–522
Shell function 513–514
source, defined 494
topics
case-sensitive 495
defined 494
overview 495–496
trappable errors *See* Error messages
DDE (dynamic data exchange) *(continued)*
version compatibility, terminology 494, 497
DDE and OLE, comparing 543
DDE links
See also DDE
Automatic *See* Automatic links
creating at design time 497–499
defined 496
destination
creating 497–498
defined 497
events 505–507
LinkClose event 506, 508
LinkError event 506, 508, 518–519
LinkExecute event 508–509
LinkItem property 500
LinkMode property 500, 504–505
LinkNotify event 507
LinkOpen event 506, 507
LinkPoke method 514
LinkRequest method *See* LinkRequest method
LinkSend method 516
LinkTimeout property 503
LinkTopic property 499–500, 503–504
Manual *See* Manual links
Notify *See* Notify links
obtaining data from 497–498
Paste Link command *See* Paste Link command
sending data through 498–499
source
creating 498–499
defined 498
events 505, 507–509
terminating errors 501
types of 496
DDE.MAK, DDE application 493
DDEML.DLL, distributing application files 579
Debug object
defined 206
Immediate pane 227
Search Help for Debug object
Debugging
accessing restrictions 230
application design techniques 233
applications, online lessons 8
break mode *See* Break mode, debugging
breakpoints
defined 210
examining application status 217
setting 215–216
statements, identifying 216
toolbar button 210
Calls
defined 210
idle loops 221

Debugging *(continued)*
nested procedures 221
toolbar button 210
using 220–221
Command function 233
command-line arguments 232–233
compile errors
defined 208
syntax errors 208
Current statement, defined 216
data declaration limitations 230
Debug window
Code window and 222
Immediate pane *See* Immediate pane
opening in break mode 214
viewing with 222
Watch pane *See* Watch pane
design-time mode
adding watch expressions 223–224
defined 212
setting breakpoints 215–216
title bar display 211
toolbar buttons 213
event conflicts
GotFocus event 232
KeyDown event 222
LostFocus event 232
MouseDown event 232
Instant Watch
defined 210
toolbar button 210
using 226–227
vs. watch expression 226
logic errors, defined 209
modes
changing from toolbar 212–213
listed 211–212
nested procedures, tracing 221
overview 209–210
Procedure Step
defined 210
toolbar button 210
using 218–219
run-time errors, defined 209
run-time mode
defined 212
fixing errors 215
title bar display 211
toolbar buttons 213
setting execution order 219–220
Single Step
defined 210
simulating procedure stepping 219
switching modes with 218
Debugging *(continued)*
toolbar button 210
using 218–219
tips 29, 234
toolbar buttons, listed 210
Decimals 166
Declare statement
array elements
huge arrays and 566
passing 565–566
converting for Microsoft Visual Basic 570–571
DLL uses 558
handles 569
multiple data types 561
multiple-line declarations 558
nonstandard names 561–562
null pointers 568
object variables 570
passing arguments by value or by reference 560
properties 569–570
specifying library 560
string arguments
length of 564
modifying during pass 564
null-terminated strings 563
unsupported data types 563
user-defined types 566–567
Search Help for Declare
Default button, setting dialog boxes 100
Default printers 108
Default Project 113
Default Save As format, Environment options 120
Default Save format 116
Degrees and radians 348–349
Delete method 466
Deleting
OLE objects, existing 533
records 466, 475
Deny-Read options in Tables 460
Deny-Write options in Tables 460
Design time
creating OLE objects 530
defined 39
displaying dialogs 536
menus, creating at design time 76
vs. Run time objects, creating 529
Design, database 481
Destinations *See* DDE
Development tools, accessing OLE objects 548
Device unavailable error message 236
Dialog boxes
Add Watch 99
adding 75
adding titles 97
Cancel button, setting 100

Dialog boxes *(continued)*
Choose Color, displaying 103
Choose Font
displaying 103, 107
Flags property 107
font properties 108
color 106
Command buttons 98–99
Common dialog custom control, using 103
creating
custom 96
discussed 75
new 97
custom
adding titles 97
colors 106
creating 96
defined 96
display options (list) 101
displaying 101
hiding forms 101
Load statement 101
loaded forms, displaying 101
loading forms into memory 101
modal forms, displaying 101
modeless forms, displaying 101
customizing existing 97
Default button, setting 100
directories, specifying in File Open 104
disabling controls 101
display options 102
displaying
commonly used 103
design time, at 536
information to users 93
drives, specifying in File Open 104
exiting, display options 102
file name extensions, specifying in File Open 104
File Open, displaying 103–105
File Print, displaying 103
File Save, displaying 103–104
focus 100
font properties, setting 108
Fonts 107
forms, unloading to free memory 103
freeing memory by unloading forms 103
Hide method display options 102
Insert Object 525–526, 530
modal 93–94
modeless 94
Paste Special 526, 534
predefined 94, 96
Print 108–109
prompting users for data 93
Dialog boxes *(continued)*
properties, setting standard 97
read-only operations, specifying in File Open 104
setting standard properties 97
standard properties
(list) 98
setting 97
Text Editor application, About dialog box 93
titles, adding and removing 97
Unload statement, display options 102
using 93
WINHELP.EXE 103
working with 93
.DIB (bitmap graphics) files 321
Dim statement
arrays
control variable arrays 192
local 171
module-level 171
variables
declaring 139–140, 149, 165
local 151–152
module-level 152
multiple, per statement 165
object 183
Variant data type 149
Search Help for Dim
Dimensions, objects 355
Dir function, file access uses 438, 443, 446
Dir$ function, file access uses 438, 443, 446
Directories
application, installing on user's machine 584
specifying File Open 104
Directory list box control
Change event 431, 433
ChDir statement 428
creating access keys 40
Drive property 427
ListCount property 428
ListIndex property 428
mouse event recognition 266
multiple controls application 26–29
naming conventions 35–36
overview 427–428
Path property 427, 428
Toolbox icon 33–34, 425
Disabling dialog box controls 101
Disk not ready error message 236
Disks, creating distribution 574
Display options, custom dialog boxes 101
Displaying
dialogs at design time 536
information in predefined dialog boxes, MsgBox statement 96
OLE dialogs to user 540

Displaying *(continued)*
other applications' data, Object Linking and Embedding (OLE) 523
Distributing applications
Add Data Access files 574
application files 578–579
Build Disks command 574
creating distribution disks 574
customized look 577
discussed 573, 577
distribution disks, creating and testing 589–590
files (list) 585, 589
GetSystemDirectory() Windows API function 585
installing application files 577
installing files 577
LOAN.EXE sample application 573
new applications 582–584
OLE Automation files 574
pre-installing files on users' machine 577
Setup program 586
SETUP1.MAK sample Setup program 573, 586
SetupWizard 548, 573–574
using OLE Automation, files (list) 548
version stamping resources 577
Search Help for Setup
Distributing OLE applications 544–545
Distribution disks 574–590
Distribution Files list box 574
Dithering 372–373
Division by zero, code example 241–242
.DLL (Dynamic-link-library) files 579
Alias keyword 558
ByVal keyword 558
C language conversions 570–571
calling DLL procedures 559, 563–570
constants 114
Declare statement *See* Declare statement
declaring DLL procedures 558–592
defined 557
discussed 111, 579
external procedures 114
Function procedures 558
general procedures 114
Lib keyword 558
multiple-line declarations 558
OLE objects, and 529
overview 558
sharing between applications 557
Sub procedures 558
subroutines 114
types 114
variables 114
Do While loop, idle loops 416–418
Do...Loop, loop structure
See also Do While loop
Do...Loop, loop structure *(continued)*
defined 144–145
DoColor procedure 63–64
Document windows, defined 297
Documentation
online 8
updated 3
DoEvents function
discussed 417–418
Search Help for DoEvents
DoEvents statement
See also DoEvents function
DDE-generated 519–520
vs. DoEvents function 417
Search Help for DoEvents
Dollar sign ($)
formatting numbers 389
String type-declaration character 164–165
Dotted lines
code example 367
drawing 366
Double data type 164
Double quotation marks (" "), formatting strings 161
Drag method
controlling start/stop 282
defined 277
Drag-and-drop operations
assigning drag icons 278
automatically dragging controls 278
centering controls 282
controlling start/stop 282
defined 276
Drag method 277, 282
DragDrop event *See* DragDrop event
DragIcon property *See* DragIcon property
DragMode property *See* DragMode property
DragOver event *See* DragOver event
highlighting controls 280
mouse coordinates, initial 282–283
moving controls 277
multiple-form operations 277
Tag property 279–280
DragDrop event
arguments available 279
code examples 279–283
defined 277
TypeOf keyword 280
using 279–280
working with Drag method 282
Dragging, defined 276
DragIcon property 277–278, 280, 282
DragMode property 277–278
DragOver event 280–281
DrawCmd procedure 397, 400–401

Drawing
arcs 348–351
boxes 344–345
circles 322, 345–346
clearing area 340
controls 19
defining drawing area 354
ellipses 347
end points 342
FillColor settings 344
fundamentals 339–341
lines 321, 341–343
ovals 322
rectangles 322
relative positioning of objects 343
squares 322
Drawing OLE controls on forms 525
DrawLbl procedure, Printer object and forms 397
DrawMode property
described 368–369
Search Help for DrawMode
DrawPic procedure
code example 401
Printer object and forms 397
DrawStyle property
described 366–367
Search Help for DrawStyle
DrawWidth property 366
Drive list box control
Change event 431–432
ChDrive statement 427
creating access keys 40
Drive property 426–427
error handling 432–433
mouse event recognition 266
multiple controls application 26–29
naming conventions 35–36
overview 426–427
Toolbox icon 33–34, 425
Drive property 426–427
Drives, specifying File Open 104
Drop-down combo boxes 50–51
Drop-down list boxes
defined 51
drive list box uses 426
unsupported event 51
Dropping, defined 276
Dynamic arrays 173–175
Dynamic Data Exchange (DDE)
See also DDE
comparing with OLE 543
Dynasets 485

E

Edit menu controls, Text Editor Application 80
Editing
AUTOLOAD.MAK file 121
current record 473
fields, values of specific 474
values of specific fields 474
Ellipses 347
Embedded objects
See also OLE objects
creating 526, 528, 532, 551
creating at run time 537–538
creating empty at run time 538
data considerations 527
displaying 551
empty, creating at run time 538
OLE controls 527
persistent data 532
vs. linked objects 527–528
Embedded quotation marks 42–43
Embedding data 524
Enabled property
changing run-time focus 67–68
default value of controls 68
setting 412
setting focus requirements 65
End of File (EOF) 468–470
End statement 476
EndDoc method
clearing Printer object 399
code example 403
printer output 393
ENTER key
keyboard handling 420
sending keystrokes with DDE 522
Environment options
background colors 120
check syntax 120
default Save As format 120
defined 107–108
grid 120
Option Base statement 171
Option Explicit statement 150–151
Save project on Run 120
setting 119–120
tab stop width 120
text colors 120
variable declarations required 120
VB.INI file 119
Environment Options dialog box 116
EOF (End of File) 468–470
EOF function
file access uses 438, 443, 446
sequential-access files 443–444

Equal sign (=), declaring constants 179
Err function
code example, delayed handling 251
returning error codes 237, 239, 251
Error codes
returning with Err function 237, 239, 251
user-defined errors 246–247
Error Err statement
defined 240
multiple-module applications 246
user-defined errors 246–247
Error events 457–458
Error function, returning error messages 237
Error handling
See also Debugging
active traps 238, 243
branching for trapped errors 238
calling DLL procedures 559
calls list 240, 243
cascading events 125
centralized error handling 247–251
code examples
delayed handling 251
disabling error traps 253
disk I/O procedures 247–248
disk-related errors 249–250
division by zero 242
file name selection by user 242–243
invalid drive selection 237
DDE and 517–520
delayed handling 251–252
disabling error traps 239, 253
drive list box errors 432–433
enabling traps 238, 243
exiting error-handling procedures 240–243
guidelines 238–240
identifying error-handling procedures 239
line label considerations 243
LinkError event 518–519
Load statement 71
multiple-module applications 246
naming convention errors 135
preventing perpetual looping 245
reasons for using 236–238
Resume statement 239–243
resuming execution 238–239
returning codes with Err function 251
returning error codes and messages 237
testing for using Null 163
transparent error messages 241
unanticipated errors 243–245
user-defined errors 246–247
Search Help for errors
Error messages
Device unavailable 236
Error messages *(continued)*
discussed 107
dialog box, MSgBox statement 96
Disk not ready 236
log 117
Microsoft Visual Basic versions 240
online Help 11
returning with Error function 237
Search Help for trappable errors
Error statement, preventing perpetual looping 245
Error trapping
DDE conversations 503, 513
defined 237
delayed handling 251–252
disabling traps 253
enabling 243
Error$(DataErr) function 475
Errors
data, handling 475
trappable
databases, opening 468
discussed 466–467, 470–471
run time 475–476
ESC key
aborting DDE conversations 503
keyboard handling 420
Event procedures
cascading event errors 125
control names, changing 128
creating 22
defined 22, 124, 127
exiting 148
multiple controls application 28
overview 127–128
syntax 128
writing 457
Search Help for events
Event-driven applications
defined 124
vs. procedural 124–125
Event-driven programming *See* Event procedures
Event-driven programs, writing, online lessons 8
Events, validate 476, 479–480
Examples, running code from Help 11
Exclamation point (!)
separating form, control references 137
Single type-declaration character 164–165
Exclusive access 460
Exclusive mode, error events 458
Exclusive property 458, 467, 470
Executable (.EXE) files
application name uses 495
creating 29
discussed 112
distributing application files 579

Executable (.EXE) files *(continued)*
file control concerns 428
making 119
OLE objects, and 529
picture considerations 324
running 119
EXEName property, defined 503, 512
Exit statement
terminating control structures 147–148
terminating procedures 148
Explicit declarations 150–151
External procedures
.BAS (code module) files 114
.FRM (form) files 114
External tools, generating forms with 116

F

F1 Function Key, online Help 9
Features, Microsoft Visual Basic 13
Fetch verbs 539
Fields
collections 474, 486–489
databases, defined 482
discussed 485
mapping 489
objects 486
properties 489
properties, listing 490
relational database objects 481
specific, editing values of 474
valid 464
validation 473, 476
values, editing 474
Fields Index property 490
FieldSize method 476
File access
accessing records 451–452
binary *See* Binary-access files
file location table, defined 451–452
file types 435
functions list 438, 443, 446
Kill statement 442
Name statement 442
overview 435
random *See* Random-access files
record number, defined 436
sequential *See* Sequential-access files
statements, listed 438, 443, 446
variables, declaring 438
File attributes 429
File list box control
Attribute properties 429
File attributes 429
File list box control *(continued)*
mouse event recognition 266
multiple controls application 26–29
MultiSelect property 429
naming conventions 36
overview 429
Path property 429
Pattern property 429
Toolbox icon 33–34, 425
wildcards 429
File location tables, defined 451–452
File menu
Add File 116
New Form 115
New MDI form 115
New module 115
Remove File 116
Save File 116
Save File As 116
File name extensions, specifying in File Open 104
File names, selection by user, code example 242–243
File Open dialog boxes
displaying 103–105
directories, specifying 104
drives, specifying 104
file name extension, specifying 104
read-only operations 104
File Print dialog boxes, displaying 103
File Save dialog boxes, displaying 103–104
File-system controls
combining controls 430
directory *See* Directory list box controls
drive *See* Drive list box controls
file *See* File list box controls
File Seeker application 430–433
overview 425–426
Pattern property 426
Search Help for file-system controls
FileCopy function, file access uses 438, 443, 446
FileDateTime function, file access uses 438, 443, 446
FileLen function
file access uses 438, 443, 446
vs. LOF function 444
Files
See also File access; Projects
activating objects from 551
activating parts of 552
adding to Project 116
adding, creating, or removing from Project 113
applications
compressing 577
installing 577
ASCII (text) format 113
AUTOLOAD.MAK 115
.BAS (code module) files 111

Files *(continued)*
binary data (.FRX) 112
classes, installing on users' machine 584
code module (.BAS) 111
compressing 583
compression 573
CONSTANT.TXT 535
custom control (.VBX) files 111, 579
data files 579
distributed with applications (list) 548
distributed with OLE applications (list) 544
distributing (list) 579
distributing applications 578
DLL files 579
.EXE (executable) 112, 119, 579
executable (.EXE) files 112
form (.FRM) 111
.FRM (form) files 111
.FRX (binary data) 112
.INI (Initialization) files 585
initialization, installing on users' machine 584–585
installing 577
installing on users' machine 584
linking to data within 531
.MAK (project) files 111
Microsoft Visual Basic, included in 2
not compressible (list) 583
operating system components, installing on users' machine 584
pre-installing on users' machine 577
program
application directory 584
installing on users' machine 584
Project (.MAK)
adding text files 117
ASCII (text) format 113
discussed 111, 113
starting 113
removing from Project 116
saving code as text 118
saving in Project 116
saving objects in 551
Setup, compressing 583
shared application resources, installing on users' machine 584–585
text 117
.VBX (custom control) 111
working with 541
FillColor property
defined 369
settings, listed 344
Search Help for FillColor
FillStyle property
selecting grid cells 288, 292
settings, listed 344
First record, moving to 469–470
Fixed-length strings 166–167, 262
FixedAlignment property, aligning text, fixed columns 289
FixedCols property, defined 286
FixedRows property, defined 286
Flags parameters, Pop-up menus (list) 92
Flags property 107
Floating-point values 166
Focus
ActiveControl property 65
ActiveForm property 65
background applications, concerns 415–416
defined 64
detecting keyboard input 419
determining 65
dialog boxes, setting 100
Enabled property 65
GotFocus event 64
label controls and 40
LostFocus event 64
receiving 64–65
SetFocus event 64
setting control focus 408
setting dialog boxes 100
shifting modeless dialog boxes 94
Visible property 65
Font properties
listed 283–284
Search Help for fonts
Font properties dialog boxes 108
FontBold property
defined 380
dialog boxes 107
FontItalic property
defined 380
dialog boxes 107
FontName property
defined 380
dialog boxes 107
order when used 381
Fonts
bundling with applications 378
changing display 379
code example, list of available 382
coordinating screen and printer 378
dialog boxes 107
GetFonts procedure 382
Print method 383
printer dialog boxes 107
printing to forms 383–387
printing to picture boxes 383–387
programming in substitution 379
properties, selecting (list) 107
proportional, guidelines 388
screen dialog boxes 107

Fonts *(continued)*
selecting dialog boxes 107
setting characteristics 379–382
small, tips 381
substitution by Microsoft Windows 378
truncated text 384
types 378
visual characteristics, defined 379
FontSize property
boxes 107
defined 380
small font tips 381
FontStrikeThru property
boxes 107
defined 380
FontTransparent property defined 380
FontUnderline property
boxes 107
defined 380
For...Next, loop structure 146
ForeColor property
defined 358, 369
line color 365
setting with PSet method 340
Search Help for ForeColor
Foreground palette, defined 376
Form arrays, saving child forms 305
Form files, contents 123
Form modules *See* Code modules; Form files
Form-level handling 423–424
Format function
displaying time 414
special characters 390
syntax 389
using 389
using named formats 389–390
Variant variables 159
Search Help for Format
Format$ function
discussed 389–392
Search Help for Format$
Format, ASCII text 116
Formatting
code example, time 392
dates and times 389–392
numbers 389–391
print output 384–386
Forms
adding Bound controls to 464
adding pictures at design time 323
adding pictures at run time 324
adding text files 117
area to redraw 361–362
ASCII text format 116
auto-instantiation, defined 187
Forms *(continued)*
BorderStyle property uses 367
Bound controls, adding to 464
changing font display 379
changing size and shape 333
clearing for print 385
code examples, printing with Print object 397
current focus 64–65
defined 17
detecting keyboard input 419
displaying multiple per container *See* MDI
drawing controls on 13
drawing OLE controls on 525
drawing on, example 339
FontTransparent property 380
.FRM files 114
generating with external tools 116
graphics methods
creating graphics 339
described 338–339
layering graphical controls and methods 363–365
idle loops *See* Idle loops
instance, defined 181
loading for debugging 229
mouse event recognition 266
moving controls with mouse events 266
moving/resizing 318, 325
multiple-form applications, drag-and-drop operations 277
naming conventions 35–36
new, creating 115
object variables *See* Object variables
online lessons 8
output to printer 404
parent, defined 46–47
persistent graphics 358–361
positioning circles 345
print coordinates 385
printing text on 383–387
printing with Printer object 395–399
programming syntax 22
programming techniques 256–257, 258
resizing 333–334
saving as text 116, 117
source *See* DDE
startup 125
terminating applications 126
textual representation of 116–117
timer controls *See* Timer controls
transferring controls between 137
transferring properties between 137
unloading to free memory 103
Forms collection 203–205
FoxPro
databases
accessing 458

FoxPro *(continued)*
external 481
opening 458
external databases 483
FoxPro 2.0 and 2.5 databases 459
Fraction values, single precision arguments 340
Frame control
area to redraw 361–362
multiple option button groups 45–46
naming conventions 36
Toolbox icon 33
FreeFile function, file access uses 438, 446
Freeing memory by unloading forms 103
.FRM (form) files *See also* Form files
FromPage property, Print dialog box 108
Front-end applications, creating with OLE 523
.FRX (binary data) files 112
Function keys
F1, discussed 9
F5, running applications 23
F8, debugging, Single Step 212
F9, debugging, setting breakpoints 216
keyboard handling 419–420
Function procedures
code example 132
data types 132
DLL uses 558
syntax 132
Variant data type 132
vs. Sub procedures 127, 130
Search Help for Function procedures

G

General procedures, .FRM (form) files 114
Generic object types 184–185, 187–188
Get statement
editing random-access records 439
file access uses 438, 446
reading binary-access files 449–450
Search Help for Get
GetAttr function, file access uses 438, 443, 446
GetChunk method 476
GetData method
copying Color palettes 375
selecting data format 408–409
Search Help for GetData
GetFonts procedure, code example 382
GetFormat method 408–411
GetObject function
discussed 526, 552
OLE Automation 551
GetSystemDirectory() Windows API function 585
GetText method 406–407
GetWindowsDirectory() Windows API function 585
Global constants 179
Global procedures, .BAS (code module) files 114
Global statement
control variable arrays 192
global arrays 171
global variables 152, 165
object variables 183
Global variables
circular references 180
defined 152
Global statement 152
lifetimes 152
name restrictions 154
object variables 184
public procedures 154
GotFocus event, described 64
Graphical user interface *See* GUI
Graphics
code example, Cut, Copy, and Paste 409–410
compatible graphics applications 323, 325
coordinate systems
defined 317
Print method 318
data format 409
displaying with Bound Picture boxes 466
drawing 321–322
fundamentals 320–326, 339–341
graphical controls 316–320
graphics methods
creating graphics on forms 339
described 316, 338–339
listed 338
vs. graphical controls 338
graphics properties, listed by category 358
grid controls *See* Grid control graphics
image controls
adding pictures at design time 323
adding pictures at run time 324
animation uses 332
bitmaps 321
defined 321
expanding pictures 325
icon 321
metafiles 321
moving/sizing pictures 325
resizing 333–334
image controls
layering graphical controls and methods 363–365
line controls defined 321
Stretch property 334
stretching pictures 321
Search Help for image control
line controls
measuring 317

Graphics *(continued)*
online lessons 8
output to printer 404
persistent *See* Persistent graphics
pictures 323–324
programming techniques 256–257, 260–261
shape controls
available shapes, listed 322
Search Help for shape control
Search Help for line control
Grid environment options 120
Grid control cells
code example 294–295
defined 285
HighLight property 293
highlighting/selecting 292–295
SelEndCol property 293
SelEndRow property 293
SelStartCol property 293
SelStartRow property 293
Grid control graphics 290–292
Grid control text 287–290
Grid custom control (GRID.VBX)
See also Grid control cells; Grid control graphics; Grid control text
combining graphics with text 292
creating access keys 40
custom control files 284
defined 284
discussed 114
naming conventions 36
rows and columns *See* Rows and columns
Toolbox icon 33–34
using in applications 284
GRID.VBX (grid) custom control
See also Grid custom control (GRID.VBX)
discussed 114
distributing application files 579
LOAN.EXE Setup program 586
startup 121
Grouping Menu items 109
Groups, Windows Program Manager, Setup program 577
GUI (graphical user interface)
described 1
programming goals 124–125
programming techniques 256–257, 260–261

H

Handling
data errors 475
data fields, large 476
Hardware requirements 2
hDC device context property, Print dialog box 108
Headers, code example 403
Height property
described 333
dimension measurement 355
moving/resizing forms 318
resizing controls 333
Help
Contents 9
Search 9
Help Compiler Guide, online Help 5
Help File, Project options 120
HELPREF.HLP online Help 5
Hex function
converting numbers to hexadecimal 49
Search Help for Hex
Hexadecimal numbers
format 134
using to set colors directly 370
Hide method
Dialog boxes, display options 102
hiding/showing controls 334–335
Menu Controls 90
Search Help for Show
HighLight property, defined 293
Highlighting text 405–406
Horizontal scroll bar control
naming conventions 36
Toolbox icon 33–34

I

.ICO (icon) files 321
Icon Library 325
Icons
See also Grid control graphics
animation
creating simple 330–333
toggling two pictures 330
assigning drag icons 278
.ICO files 321
image controls 321
provided in Icon Library 325
toolbar, list of 16–17
Toolbox, list of 33–35
Windows Program Manager, Setup program 577
IconWorks sample application, online Help 5
ICONWRKS.HLP, online Help 5
Idle loops
code example, prime numbers 418
defined 416
DoEvents function *See* DoEvents function
restricting 416
writing 416–418
If...Then, decision structure 141–142

If...Then...Else, decision structure
defined 142–143
Search Help for If
Image control
adding 466
advantages and limitations 320
BUTTERF.MAK, simple animation application 23–26
Click event 38
compared to picture boxes 256, 321
creating access keys 40
discussed 463
displaying 25
displaying graphics 466
multiple controls application 26–29
naming conventions 36
printing considerations 303
programming techniques 256, 260
Toolbox icon 33–34
using to create toolbars 313
vs. command button controls 38
Image property, programming techniques 261
Images
clipping 530
displaying in OLE objects 530
pasting to Clipboard 466
Immediate pane
described 227–231
Search Help for Immediate
Implicit declarations 149–150, 152
In place activation 539
Inaccessible OLE objects 554
Inches, scale measurement 353
Index
collections 556
objects 487
properties
Menu Control Arrays elements 80
printing 490
Index property
control arrays
attributes 336
creating 70–71
settings 71
values 69–70
Indexed sequential access method (ISAM) databases 483
Indexes
collections 487
Data Manager, creating with 485
database, defined 482
discussed 485
mapping 490
primary 487
properties 490
.INI (Initialization) files 584–585
Initialization (.INI) files, installing on users' machine 584–585
Input statement
file access uses 446
program variable uses 444
Input# statement
using in sequential file access 442–444
Search Help for Input#
Input$ function
editing strings for sequential access 442–444
file access uses 442, 446
reading binary-access files 450
InputBox function, predefined dialog boxes
discussed 94
prompting for input 95
Search Help for InputBox
Insert Object command, deleting existing objects 533
Insert Object dialog box
discussed 526, 530
displaying to user 540
Object Linking and Embedding (OLE) 525
Installing
applications 573
Microsoft Visual Basic 3
OLE 2.0, distributing OLE applications 545
printers 108
Instances
auto-instantiation, defined 187
creating 198–199
defined 181
loading/unloading 202–203
Me keyword 201–203
multiple 201–202, 302, 316
Nothing keyword 202–203
reference count, defined 202
scope considerations 202
Instant Watch, debugging 210, 226–227
Integer data type 164
Integrity rules, Microsoft Access databases, violating 465
Interface *See* Microsoft Visual Basic
Interior dimensions, object 355
International formatting
dates/times 391–392
numbers 391
Interval property
application interaction 411–412
interval limit 411
precision 411
setting 412
using system clock 411
Invalid records 468
Invisible Menu Controls 89
Is function 468
Is operator, testing object variables 199–200

ISAM (Indexed Sequential Access Method) external databases 483
IsDate function 162
IsEmpty function 162
IsNull function 163
IsNumber function 158–159
Items *See* DDE
Iterating through numeric collections, limitations in Visual Basic 556

J

Jumps, defined 11

K

Keyboard handlers
access keys in labels 40
code examples
detecting specific keys 420, 423–424
entering uppercase text 420
event interactions 419–420
events, listed 419
form-level handling 423–424
global constants 423
prohibiting key events 424
shortcut keys, using instead of keyboard handlers 420
See appropriate event
Keyboard, detecting input 419
KeyDown event
defined 419
determining ANSI code 421–422
form-level handling 423–424
KeyCode argument 421–422
KeyPress, and KeyUp events 419–421
Shift argument 422–423
Search Help for KeyDown
KeyPress event
code examples 420
converting ASCII characters 420
defined 419
form-level handling 423–424
intercepting keyboard input 18–19, 42
KeyDown, KeyUp events 419–421
Search Help for KeyPress
KeyPreview property
detecting keyboard input 419
form-level handling 423–424
Search Help for KeyPreview
Keys
assigning 85
introduced 86
menu, (&) Ampersands 80
Keystrokes, sending through DDE 521–522
KeyUp event
defined 419
determining ANSI code 421–422
form-level handling 423–424
KeyCode argument 421–422
KeyDown and KeyPress events 419–421
Shift argument 422–423
Search Help for KeyUp
Kill statement, deleting files 442

L

Label control
AutoSize property 334
creating access keys 40
discussed 463
displaying text 38–39
multiple controls application 26–29
naming conventions 36
printing 397
sizing 39
Toolbox icon 33
vs. text box controls 38
Labeling distribution disks 589
Last record, moving to 469–470
LastUpdated property 488
Layering of controls 363–365
LBound function 555
collection restrictions 203
using with arrays of control variables 192
Learning tools, online 8
Left property 326–328
Lessons, online 8
Lib keyword, DLL uses 558
Limitations in Visual Basic 554–556
Line breaks, word wrapping, with text boxes 41
Line controls
advantages and limitations 320
code example 327
moving with coordinates 327
naming conventions 36
Toolbox icon 33–34
Line Input# statement
file access uses 442
reading lines from sequential-access files 443–444
Line labels
considerations when using 243
identifying error-handling procedures 239
On Error GoTo statement 239
Line method
B option, drawing boxes 344
ClipControls property, effects 361
code examples
boxes 344

Line method *(continued)*
lines and circles 269
polygons 342
randomly scattered lines 342
solid, dashed, dotted lines 367
straight lines 268
MouseDown event 268
MouseMove event 271
varied line width 366
defined 338
FillColor property 345, 369
printer output 393
relative coordinates 343
setting colors by pixels 402
syntax 267, 341
Lines
Circle method 350
code examples
drawing polygons 342
drawing randomly scattered lines 342
drawing solid, dashed, dotted lines 367
pie chart with missing slice 350
printing to printer 399
varied line widths 366
color argument 365
drawing solid, dashed, dotted lines 366
drawing with line controls 321
drawing, graphics method 341–343
printing to printer 397
varying style and width 365
width 366
LinesOnPrinter procedure
code example 399
Printer object and forms 397
LinkClose event
destination link uses 506
source link uses 518
Linked and Embedded objects, contrasting 527
Linked objects *See* OLE objects
Objects, linked 526
creating 526, 530, 551
creating at run time 537
data considerations 527
displaying 551
paths, valid 531
vs. embedded objects 527–528
LinkError event
destination link uses 506
trapping for destination links 518
trapping for source links 519
Search Help for LinkError
LinkExecute event
destination link uses 508–509
sending commands through DDE 515
Linking
data 524
Paste Special 532
within files 531
objects, SourceItem property 536
LinkItem property
defined 500
Search Help for links
LinkMode property
defined 500, 504–505
LinkRequest method 500–502
Search Help for links
LinkNotify event
LinkOpen event
destination link uses 506
source link uses 507
LinkPoke method 514
LinkRequest method
LinkMode property 500–502
requesting data, DDE 515
Links, DDE *See* DDE links
LinkTimeout property
error trapping 503
Search Help for links
LinkTopic property
See also Application object (App)
defined 499–500, 503–504
terminating conversations 504
Search Help for links
List box control
accessing list items 54
adding items 52–53
AddItem method *See* AddItem method
capturing user input 49–59
Clear method *See* Clear method
code example 55–57
Columns property *See* Columns property
counting items 55–56
creating access keys 40
display options 49
drop-down list boxes 51
identifying selections 54
List property *See* List property
ListCount property *See* ListCount property
ListIndex property *See* ListIndex property
mouse events 266
multiple-column list boxes 57–59
multiple-selection list boxes 57–59
MultiSelect property *See* MultiSelect property
naming conventions 36
position of list items 54–55
RemoveItem method *See* RemoveItem method
removing items 53
Sorted property *See* Sorted property
sorting list 53

List box control *(continued)*
Text property *See* Text property
Toolbox icon 33–34
vs. combo boxes 51
List boxes, in distribution files, SetupWizard 574
List property 54
ListCount property
and directory list box 428
code example, list boxes 55–57
counting list items 55–56
ListIndex property
and directory list box 428
position of list items 54–55
Load statement
adding/removing controls 335–338
code examples
adding/hiding controls 336
moving shape controls 337
control array uses 71–73
error handling 71
Load statement, custom dialog boxes 101
Loading text files as code 118
LoadPicture function 466
adding pictures at run time 324
assigning drag icons 279
grid control graphics 291
loading bitmap graphics 28
programming techniques 261
removing pictures at run time 324
syntax 291
Search Help for LoadPicture; Picture
LOAN.MAK, LoanSheet application 284–296
Loc function, file access uses 438, 443, 446
Local databases, connecting to 457
Local variables
declaration methods 151–152
defined 151
defining as Static 154–156
Dim statement 151–152
implicit declaration 152
in recursive procedures 152
object variables 184
preserving 154–156
LOF function
file access uses 438, 443, 446
vs. FileLen function 443
Log, error messages 117
Logical palette
background palette 376
described 373
foreground palette 376
increasing Color palette 373
z-order effects, described 376
Long data type 164
LostFocus event
described 64
Search Help for LostFocus
LTrim function
stripping leading spaces 49
Search Help for LTrim

M

Macro languages, accessing OLE objects 548
.MAK (project) files
See also Project files
discussed 111
starting from MS-DOS prompt 113
common dialog (CMDIALOG.VBX) 114
discussed 111, 579
grid control (.GRID.VBX) 114
loading .VBX controls 524
OLE (MSOLE2.VBX) 114
OLE controls 524
Managing records 468
Manipulating
objects using methods 553
other applications' data, Object Linking and Embedding (OLE) 523
Manual links
creating 501–502
defined 496
Mapping
databases 483, 488
fields 489
indexes 490
tables 488
Masked control 463
MaxButton property
and Microsoft Windows version 300
effects of 90
MDI child forms 300
MaxButton standard dialog box property 98
MaxLength property, setting string length 41–42
MDI (multiple-document interface)
ActiveForm property 303
BorderStyle property 300, 303
child forms
defined 298
design time vs. run time 299–300
minimizing results 298
saving 304–305
ControlBox property 300
creating MDI applications
example 301–303
modifying AUTOLOAD.MAK 299
overview 298–299
document windows, defined 297–298

MDI (multiple-document interface) *(continued)*
document-centered, defined 301
focus 315–316
form arrays
code example 306–307
triggering save 305
form-level variables, triggering save 304
forms, using standard 298, 303
initial size and position 307
internal area, defined 298
loading 303
MaxButton, MinButton properties 300
Me keyword 302
menus
changing with focus 308
overview 308–310
Window menu *See* Window menu
WindowList property 310–311
Microsoft Windows version compatibility 300
multiple instances 302, 305
parent window, defined 297
QueryUnload event 307–308
saving contents of child forms 304–305
startup forms 303
status bar *See* Toolbar
toolbar *See* Toolbar
unloading 307–308
MDINOTE.MAK, MDI NotePad application 297
Me keyword
auto-instantiation 201–202
passing as argument 202
specifying current form 302
static, module-level concerns 203
Measurement scales 353
Memory
considerations, OLE Automation 554
freeing by unloading forms 103
limitations in Microsoft Visual Basic 360
persistent graphics, effects upon 360
Menu access keys, (&) Ampersands 80
Menu bar 76
Menu bar, defined 17
Menu commands
disabling 87
enabling 87
Menu control arrays
creating 79
creating in Menu Design window 80
DDE selections 510–511
elements, Index properties 80
mnuFileArray menu control 89
using 79
writing code for 81
Menu control
access keys *See* Access keys
Menu control *(continued)*
adding at run time 89
arrays *See* Menu control arrays
Checked property 87–88
checks marks, displaying 87
click event procedures for 81
code, writing for 81
creating 109
creating dynamically 89
creating in Menu Design window 79
Cut, Copy, and Paste, working with 407
defining appearance 109
dynamically creating 89
equivalents, toolbar icons 16–17
Hide method 90
hiding 89, 90
menu bar 17
Menu Design window *See* Menu Design window
menu items *See* Menu items
menu titles *See* Menu titles
naming conventions 36
separator bars 84
shortcut keys *See* Shortcut keys
toolbar, creating 313
Toolbox icon 35
Unload statement 90
Visible property 89
writing code for 81
Search Help for Menu
Menu Design window
access keys, assigning 85
Caption property 77
creating separator bars 84
displaying 77
menu control arrays, creating 80
menu controls
creating 79
design-time properties 77
Name property 77
separator bars, creating 84
using 77
Menu Design window, Index property *See* Index property
Menu items
adding at run time 89
disabling at run time 86
dynamically creating 89
grouping 76
Menu control position 78
Menu titles
introduced 76
Menu control position 78
Window menu in MDI applications 311-313
Menus
See also Pop-up menus 533
access keys, assigning 85

Menus *(continued)*
adding 75
check marks 86
commands
disabling 87
enabling 87
controlling at run time 86
creating 75
creating at design time 76
creating submenus 82
disabling menu items at run time 86
online lessons 8
Pop-up
CONSTANTS.TXT file 92
displaying 90
displaying with right mouse button 91
Flags parameters (list) 92
OLE controls 533
PopupMenu method 90
separator bars 78
Shortcut keys, assigning 85
Messages, log of error 117
Metafile images of OLE objects 528
Metafiles
colors supported 371
image controls 321
.WMF files 321
Methods
AddNew
aborting 466
described 465, 472–473, 480
AppendChunk 476
Close 473, 476
defined 24
Delete 466
FieldSize method 476
GetChunk method 476
Move 473, 480
MoveFirst 470
MoveLast 467, 470
MoveNext 470, 475
MovePrevious 471
performing 553
Refresh 468
databases, opening 468
discussed 460, 476, 481
first record, moving to 469
opening databases 458
opening new databases 467
Update 465, 472, 474, 476, 481
UpdateControls 466, 481
UpdateRecord 479, 481
Microsoft Access database engine 1, 453
Microsoft Access databases
accessing information in 458
Microsoft Access databases *(continued)*
integrity rules, violating 465
opening 458
queries 460, 462
tables, attached 460
Microsoft C *See* DLLs
Microsoft Excel *See* DDE; OLE
Microsoft FoxPro databases 453
Microsoft Product Support Services 14
Microsoft SQL Server databases 483
Microsoft Visual Basic
See also DDE; Help
Basics, transferring from other 4
hardware requirements 2
installing 2–3
interface
MDI *See* MDI
overview 16
setting properties 20–21
working with controls 18–19
limitations 554
modifying installation 3
new features (list) 13
Professional Edition, custom controls 114
Project Window, automatically loaded 113
projects, building and managing 111
sample applications *See* Sample applications
setting up 2
Standard Edition, online Help 5
starting 15–17
system requirements 2
transferring from other Basics 4
version compatibility
arrays, declaring 172
data types 149
DDE terminology 494, 497
object variables 189
Microsoft Windows
Control Panel settings 161
DDE *See* DDE
font substitution 378
International dialog 391–392
multitasking 416
running executable files 107
setting default system colors 370
system color hexadecimal values 371
system palette 372
version differences and properties 300
Microsoft Word *See* DDE; OLE
Millimeters, scale measurement 353
MinButton property
MDI child forms 300
standard dialog box property 98
mnuFileArray menu control 89
Modal dialog boxes 93–94

Modal forms, custom dialog boxes 101
Modeless dialog boxes 94
Modeless forms, custom dialog boxes, display options 101
Modifying installation 3
Modular programming
components 127
constants, global 249
defined 126
DoColor procedure 63–64
print procedures 400
using general procedures 128–130
Module-level variables
defined 152
Dim statement 152
object variables 184
private procedures 154
Modules *See* Code modules
adding text files 117
.BAS files 114
saving as text 117
Monitors, color adaptation 373
Mouse button, right
displaying pop-up menus 91
pop-up menus 532–533
Mouse events
See also Drag-and-drop operations; MouseDown event; MouseMove event; MouseUp event
Button argument 271–274
listed 266
mouse button status
defining constants for 274
described 271–274
moving controls 282–283
Shift argument 276
MOUSE.MAK, mouse events application
code example, graphics 316
location 265
MouseDown event
Button argument 275
code examples
activating drawing 270
centering controls 267
Click-A-Line application 267–268
drawing lines 268
mouse button status 272
moving controls 267, 283
Scribble application 270–271
testing Shift argument 276
defined 266
DragMode property 280
mouse button status 272
mouse coordinates, initial 282–283
Move method combinations 266
MouseMove event
Button argument 273–275
MouseMove event *(continued)*
code examples
activating drawing 271
drawing curved lines 268
drawing lines and circles 269
mouse button status tests 273
Scribble application 268–271, 274–275
coordinates 266
defined 266
mouse button status 272, 273–274
working with 270
MouseUp event
Button argument 275
code examples
deactivating drawing 270
mouse button status 272
moving controls 283
Scribble application 270–271
defined 266
DragMode property 280
mouse button status 272
mouse coordinates, initial 282–283
Move method 473, 480
and bitmap graphics 24, 26
code examples
centering controls 267, 282
moving controls 267, 282
compared with Left, Top properties 328
MouseDown event combinations 266
syntax 328
Search Help for Move
MoveFirst, MoveLast, MoveNext, MovePrevious methods 470-475
Moving
absolute movement 328
code example, moving line controls 327
controls
at design time 19
dynamically 326–330
Left, Top properties 326
on container 46–47
with Cut and Paste commands 46
line controls with coordinates 327
redrawing screen images 358–361
relative movement 328
speed and smoothness 330
Moving to specific records with Bookmarks 471
MS-DOS prompt
executable files
making from 119
running from 119
.MAK files, starting from 113
MSAES110.DLL, distributing application files 579
MSAFINX.DLL, distributing application files 579
MSAJT110.DLL, distributing application files 579

MsgBox function
predefined dialog boxes 94
Search Help for MsgBox
MsgBox statement
predefined dialog boxes 94, 96
Search Help for MsgBox
MSOLE2.DLL, distributing applications 585
MSOLE2.VBX (OLE) custom control
discussed 114, 524
distributed with OLE applications 544
distributing application files 579
Multidimensional arrays 172–173
MultiLine property, word wrapping, text boxes 41
MULTINST.MAK, multiple instances application 181
Multiple transactions 478
Multiple-document interface *See* MDI
Multiple-form operations, drag-and-drop 277
Multiple-module applications, error-handling guidelines 246
MultiSelect property
file list box uses 429
selecting entire list 59
selecting multiple items from list boxes 57
Search Help for MultiSelect
Multitasking 416

N

Names
database properties 488
Index property 490
Name property 70, 488
Name statement, renaming files 442
Named arguments, limitations in Visual Basic 555
Named ranges, object linking and embedding (OLE) 531
Names, database 462
Naming conventions
errors 135
forms collection, shadowing 203
Me keyword 153–154
misnaming variables 150–151
name conflicts 153–154
object naming conventions 35–36
prefix identifiers, listed 35–36
reserved words *See* Reserved words
rules used 135
shadow variables 153–154
sharing names 153–154
Navigating Recordsets 469
Nested procedures, debugging 221
Nesting, defined 147
New databases, opening 467
New features (list), Microsoft Visual Basic 13
New Form menu command 115
New keyword
creating instances 198–199
declaring object variables 187–188
New MDI Form menu command 115
New Module menu command 115
New Project menu command 115
NewIndex property
vs. ListIndex property 55
Search Help for NewIndex
NewPage method
clearing Printer object 399
code example 403
using with Printer object 393
Next record, moving to 469, 470
Nonscalable fonts, defined 378
Nop, DrawMode setting described 368
Not keyword, testing object variables 200
NotCopyPen, DrawMode setting described 368
Notify links
See also DDE; DDE links
bitmap graphics 502
creating 502
defined 496
Now function
date/time formatting 391
international dates/times formatting 391–392
Search Help for Now
Null
See also DLLs
defined 163
IsNull function 163
testing for errors 163
Search Help for expressions
Null keyword 154, 163
Number formatting, symbols used 389
Number sign (#)
Double type-declaration character 164–165
formatting numbers 161, 389
Numbers
conversion 159
converting to strings 137, 389
converting to Variant data type 389
data types 158, 165–166
decimals 166
floating-point values 166
Format function 159
formatting
dates and times 391
default formatting 391
international 391
international dates/times 391–392
using named formats 389–390
hexadecimal representation 134
IsNumber function 158

Numbers *(continued)*
octal representation 134
plus sign (+), concatenation vs. addition 157
rounding 166
Variant variables 158
Numeric collections
iterating through 556
limitations in Visual Basic 556
Numeric keypad, keyboard handling 419–420

O

OBDC.DLL, distributing application files 579
Object box 20-22
Object class 526
Object Linking and Embedding *See* OLE
Linked objects, inserting placeholders 527
OLE automation 526
OLE controls, pop-up menus 533
Object linking and embedding (OLE)
advantages of 543–544
applications, container 526
applications, front end, creating with 523
comparing OLE and DDE 543
container applications 526
Create Embedded Object command 533
Create Link command 533
data 523–524
discussed 523
distributing OLE applications 545
embedded objects, creating 528, 532
embedding data 524
front-end applications, creating with 523
fundamentals 524
Insert Object dialog box 525
installing OLE 2.0 without SetupWizard or SetupKit 545
linked and embedded, contrasting 527
linked objects
creating 530
discussed 530
vs. embedded objects 528
linking data 524
linking to data within files
discussed 531
R1C1 syntax 531
objects, displayed, determining how 530
OLE automation, previous versions of OLE 544
OLE control, using 529
OLE terms 525
previous versions of OLE 544
program control, transferring 543
ranges, named 531
using OLE control 529
Object property syntax 552
Object variables
See also DLLs
arrays *See* Arrays
assigning values
references, defined 197–198
testing 199–200
declaring 183
defined 181
determining types
code example 194
syntax 194–195
TypeOf keyword 194–195
example 182
generic object types *See* Generic object types
implicit declaration, forms 188–189
instances *See* Instances
Is operator 199–200
New keyword 187–188, 198–199
Not keyword 200
Nothing keyword, defined 196–197
overview 181–182
passing objects
procedure arguments 192
return values, valid 192
scope by declaration 184
Set statement 195–196
Show method 182, 188
specific object types *See* Specific object types
Object, database 484
Objects
accessing linked and embedded 551
activating 539, 551
class 526
creating
Action property 535
Class property 535
design time vs. run time 530, 535
embedded objects 537, 538
invisible 550
new 548
OLE Automation 547
OleTypeAllowed property 535
SourceDoc and SourceItem properties 535
data, editing or playing 539
databases, relational 481
default verbs 539
embedded
accessing 551
creating 526, 528, 532
discussed 525–526
displaying 551
exposing by OLE applications 525
Field 486
Index property 487
invisible, creating 550

Objects *(continued)*
linked
accessing 551
creating 526, 530, 551
creating at run time 537
discussed 525–526
linked and embedded, contrasting 527
linking
SourceDoc property 536
SourceItem property 536
manipulating
methods, using 553
OLE Automation 547, 552
metafile images of 528
new, creating 548
OLE Automation 526
programming other applications' 547
properties, accessing 552
property values, retrieving 553
recordsets, accessing 491
relational database 481
TableDefs 485
using existing 550
verbs 539
See OLE Objects
Objects, dimensions 355
OBJECTS.MAK, object variables application 181
Oct function
converting numbers to octal 49
Search Help for Oct
Octal numbers, format 134
ODBC databases 459, 483
ODBCINST.DLL, distributing application files 579
OLECLIEN.VBX startup 121
OLE (object linking and embedding)
See also Object Linking and Embedding (OLE), OLE objects
applications 525, 543, 544
DDE, comparing with 543
fundamentals 524
MSOLE2.VBX 114
naming conventions 36
registering OLE 2.0 545
server application *See* Server application
Toolbox icon 33, 34
OLE 2.0 registration 545, 548
OLE Automation
accessing objects' properties 552
activating objects from files 551
activating parts of files 552
arguments, variant data type 553
Class parameter 552
Close method 554
discussed 526
distributing application files (list) 548
OLE Automation *(continued)*
files, activating parts of 552
inaccessible OLE objects 554
introduced 548
limitations 554
manipulating objects 552
memory considerations 554
object property syntax 552
objects
See also OLE objects
creating 547
manipulating 547, 552
subjects, containing 553
OLE objects, inaccessible 554
OLE.DLLs, function 552
previous versions of OLE 544
Quit method 554
saving objects in files 551
sharing data 547
variant data type 553
Windows, earlier versions 547
OLE constants in CONSTANT.TXT 535
OLE controls
Action property 536
custom controls 524
data, reading into 542
distributing applications using 544
drawing on forms 525
embedded objects 527, 551
linked objects 551
pop-up menus 529, 532–533
using 529
OLE Dialogs, displaying 540
OLE objects
creating 529, 530, 534, 548, 550
data
accessing 547
manipulating 547
saving from 541
deleting existing 533
discussed 525
displayed, determining how 530
.EXE (executable) files and 529
exposed
containers 548
development tools 548
functionality 550
macro languages 548
inaccessible 554
invisible, creating 550
Paste Special command 533
persistent data 541
saving data from 541
using existing 550
verbs 533

OLE terms 525
OLE DLLs, function 552
OLE_CREATE_EMBED action 538
OLE_EITHER OleTypeAllowed property 535–536
OLE_EMBEDDED OleType 541
OLE_EMBEDDED OleTypeAllowed property 535
OLE_LINKED OleType 541
OLE_LINKED OleTypeAllowed property 535
OLE_NONE OleType 541
OLE2.DLL 544, 548, 579
OLE2.REG 544, 548, 579
OLE2BLS.DLL 548
OLE2CONV.DLL
  distributed with applications 548
  distributed with OLE applications 544
OLE2DEMO.MAK, Object Linking and Embedding (OLE) examples 523
OLE2DISP.DLL
  distributed with applications 548
  distributed with OLE applications 544
  distributing application files 579
OLE2NLS.DLL
  distributed with OLE applications 544
  distributing application files 579
OLE2PROX.DLL
  distributed with applications 548
  distributed with OLE applications 544
  distributing application files 579
OLE2UTIL.DLL
  distributed with applications 548
  distributed with OLE applications 544
  distributing application files 579
OleType property
  OLE_EMBEDDED OleType 541
  OLE_LINKED OleType 541
  OLE_NONE OleType 541
  querying 541
OleTypeAllowed property
  Create Link command 533
  creating objects 535
  creating objects at run time 535
  OLE_EITHER 535–536
  OLE_EMBEDDED 535
  OLE_LINKED 535
Online documentation 8
online Help
  API functions 5
  code samples, copying from 11
  code samples, running from 11
  Code window 10
  context-sensitive 10
  copying code samples from 11
  CRW.HLP 5
  Crystal Reports 5
  CTRLREF.HLP 5
online Help *(continued)*
  discussed 9
  error messages 11
  forms on objects 10
  HELPREF.HLP 5
  IconWorks sample application 5
  ICONWRKS.HLP 5
  jumps defined 11
  objects on forms 10
  Properties window 10
  reserved words 10, 11
  running code examples from 11
  standard edition 4
  toolbar items 10
  Toolbox 10
  VBAPI.HLP 5
  VBKNOWLG.HLP 5
  WIN31API.HLP 5
  WIN31WH.HLP 5
  Windows 3.1 API 5
Online lessons 8
Open Project menu command 115
Open statement
  file access uses 438, 443, 446
  random-access files 436–437
OpenDatabase function 485
Opening Projects 115
Operating system components, installing on users' machine 584–585
Option Base statement, array uses 171
Option button control
  code example 48
  creating option button groups 45–46
  naming conventions 36
  selecting 47–49
  shortcut keys 47
  Toolbox icon 33–34
Options menu
  Environment 119–120
  Project 120
Options property 458, 460–461, 467
Oracle databases 483
OTHERNEEDED, SETUP1.FRM constants 587
Ovals, drawing with shape controls 322

P

PACKING.LST, files included in Microsoft Visual Basic 2
Page numbers, code example 403
PAGEDOWN, PAGEUP keys 419–420
Paint event
  ClipControls property 361
  normal layering uses 363, 365
  persistent graphics 360

PalEdit application, editing bitmaps with Color palettes 375
Paradox databases
discussed 453, 459
external 481, 483
opening 458
Parent window, defined 297
Password text boxes
creating 41
MaxLength property 41–42
PasswordChar property 41–42
Paste Link command
adding to applications 509–510
disabling 510–511
general DDE uses 497–499
user-defined links 509–513
Paste operations 466
Paste Special command
creating objects using 534
discussed 533
linking data 532
objects, pasting 533
Paste Special dialog box
creating objects using 534
discussed 526
displaying to user 540
Pasting
Clipboard object 407–408
Color palettes 375
moving controls 46
pictures, design time 323
selecting text 405–406
Path property
directory list box uses 427
file list box uses 429
Path, Right function, determining root 28
Paths, linked objects 531
Pattern property
file list box uses 429
file-system control uses 426
PDX110.DLL, distributing application files 579
Pending transactions 478
Percent sign (%), Integer type-declaration character 164–165
Performing methods 553
Period (.), separating form, and control references 137
Persistent data
embedded objects 532
OLE objects 541
Persistent graphics 358–362
Pi, relationship to radians 348
Picture box control
Align property 334
AutoRedraw considerations 361–362
AutoSize property 325, 334
changing font display 379
clearing for print 385
Picture box control *(continued)*
code example, printing to printer 401
creating access keys 40
creating option button groups 45–46
Cut, Copy, and Paste 409–410
DDE, using LinkSend to update 516
described 383, 461
destinations *See* DDE
drawing on, example 339
FontTransparent property 380
graphics methods, describing 338–339
mouse event recognition 266
moving with mouse events 266
moving/sizing pictures 325
naming conventions 36
print coordinates 385
printing image to printer 397
printing text on 383–387
redrawing 358–361
resizing 333–334
toolbar uses 313
Toolbox icon 33
Picture boxes 466
Picture property
code example, simple animation 331
deleting pictures at design time 323
grid control graphics 290–292
setting at design time 323
Search Help for Picture
Pictures 323–325, 330–333
PICTVIEW.MAK, Picture Viewer application 15, 26–29
Pixels
changing color values 369
code example, setting colors by 340–341
color setting for picture box controls 402
defined 353
dithered colors 372
line width 366
PrintForm method 404
printing form images 394
setting colors by 340
turning on and off 369
Search Help for Pset
Placeholders, inserting 527
Plus sign (+), concatenating 159–160
Point method
defined 338
example 341
Pointer icon
defined 33
Toolbox display 33
Pointers, reference corollary 198
Points, defined 353
Polygons, code example 342

Pop-up menus
CONSTANT.TXT file, sample constants 92
displaying 90
Flags parameters (list) 92
mouse buttons, right 533
OLE controls 533
PopupMenu method 90
right mouse button. displaying with 91
PopupMenu method, displaying pop-up menus 90
Pre-installing files with Setup program 577
Predefined dialog boxes
discussed 94
displaying information, MsgBox statement 96
error messages 96
InputBox function *See* InputBox function
MsgBox function *See* MsgBox function
MsgBox statement *See* MsgBox statement
prompting for input 95
Preserve keyword, dynamic arrays 175
Previous records, moving to 469, 471
Primary Index property 490
Primary indexes 487
Print dialog box 108–109
copies, specifying
displaying 109
FromPage 108
hDC device context 108
PrinterDefault property 109
ToPage 108
using 108
WIN.INI file 109
Print method
affected by AutoRedraw property 361
changing font display 379
code examples, tables 387, 388
combining items 384
commas used with 387
coordinate systems 318
Immediate pane 227–229
printing tables 387
printing variables, debugging purposes 222
shortcut key 228
syntax 383
using as graphics method 338
Print zone 387
Print# statement, writing text into files 445
Printer
resolution, printing with forms 394
setting margins 395
setup with Printer object 394
testing compatibility 393
Printer fonts, defined 378
Printer object
changing font display 379
Printer object *(continued)*
clearing between forms 399
code examples
multiple forms 399
printing forms with 397
CurrentX, CurrentY properties 395
defined 206
graphics method output 339
printed output 393–394
printed page margins 395
printing controls 399
printing forms with 395–399
using with multiple forms 399
using with Print method 383
Printer output
bitmap graphics 394
fundamentals 393
multiple pages 393
positioning on page 395
print quality 392
Printer object 393–394
Printer object and forms 395–399
PrintForm method 404
setting margins 395
testing compatibility 393
using forms 394
PrinterDefault property, Print dialog box 109
Printers
configuring 108
currently installed, displaying 108
default 108
reinstalling 108
PrintForm method
AutoRedraw property 404
printing by pixels 404
syntax 404
Search Help for PrintForm
PrintFrm procedure
code example 397
Printer object and forms 397
Printing
clearing forms and picture boxes 385
Cls method 385
code examples
command buttons with Printer object 400
end of documents 403
form to Print object 397
formatted output 403
lines with Printer object 399
multiple forms 399
multiple items per line 384
multiple-page documents 403
one item per line 384
picture box controls with Printer object 401
tables, custom tabs 388

Printing *(continued)*
command button images 397
control images 397
coordinates 385
CurrentX, CurrentY properties 385
form margins 397
formatted output, combining items 384
label images 397
lines 397
picture box controls 397
print zone 387
printer output *See* Printer output
specific locations 385–386
tables
custom tabs 388
default tabs 387–388
text to forms and picture boxes 383–387
to form and Print object 398
Printing Index properties 490
Private keyword, defined 133
Procedural applications, vs. event-driven 124–125
Procedure list box, defined 22
Procedure Step, debugging 210, 218–219
Procedures
defined 22
event *See* Event procedures
event-driven 22
exiting 148
form-specific 127
Function *See* Function procedures
general 128–130
recursive, local variables and 152
Sub *See* Sub procedures
switching between 28–29
tasks, listed 130
Procedures, writing event 457
Product Support Services 14
Program control, transferring with OLE 543
Program files, installing on users' machine 584–585
Programming
See also Programming techniques
basic steps 18–23
Color palette
limitations 375
z-order effects, described 376
compatible graphics applications 323, 325
event-driven 22
fonts
bundling, copyright considerations 378
programming in substitution 379
small font tips 381
increasing color selection 374
memory usage, persistent graphics 360
modular programming, print procedures 400
performance, graphic considerations 360–362
Programming *(continued)*
pictures, design considerations 324
printing, testing printer compatibility 393
running applications 23
saving 29
setting default system colors 370
Programming techniques
accessing control values 255
application size, reducing 258–260
arguments 262
arrays 259
AutoRedraw property 256–257, 260–261
calling procedures 255
Clear method 257
Cls method 261
constants 260
DDE conversations vs. size 504
dead code 260
declaring DLL procedures 560
design techniques 233
fixed-length strings 262
GUI considerations 256–257
image controls 256, 260
Image property 261
LoadPicture function 261
object variables 189
optimize, defined 254, 258–260
property values as variables 255
recursion 262
run-time considerations 258
Show method 257–258
specific object types 255
speed/clarity issues 139
stacks 261–262
startup forms 257–258
Static variables 262
variable types 255, 260, 262
Variant variables 255, 259
Project files
See also Code module files; Custom controls; Form files
ASCII (text) format 113
described 112
discussed 113
starting 113
Project window
defined 17
discussed 113
loads automatically 113
Projects
adding custom controls to 114, 524
adding files to 116
adding text files to 117
application name uses 495
applications, creating 112
building 111

Projects *(continued)*
closing 115
component files 111
creating 115
creating applications from 112
default custom controls 524
defined 111
elements 114
.MAK (project) files *See* .MAK (project) files
managing 111
New menu command 115
Open menu command 115
opening 115
options 120
removing files from 116
Save As menu command 115
Save menu command 115
saving 115
updating 115
PromptForNextDisk function call, Setup program 586–587
Prompting for input, InputBox function, predefined dialog boxes 95
Properties
See also Properties window
accessing control values 138–139
accessing objects 552
data-aware 463
Database Object, listing 488
programming techniques 255, 260, 262
retrieving values 137
setting values 136–137
Properties list
defined 20
selecting from 21
Properties window
defined 17, 20
Object box, defined 20
online Help 10
Properties list, defined 20
Settings box, defined 20
Property settings, examples listed 44
Property values, retrieving from objects 553
Proportional fonts, guidelines 388
PSet method
code example, setting colors by pixels 340–341
defined 338
drawing end points 342
relative positioning of objects 343
setting background color 341
setting colors pixel by pixel 340
setting ForeColor 340
setting line width 366
syntax 340
using with Printer object 393
Search Help for PSet
Put statement
adding random-access records 440–441
editing random-access records 440
file access uses 437, 446
reading binary-access files 450
Search Help for Put

Q

QBColor function
setting colors 319
specifying color values 318
Search Help for QBColor
Queries
databases, defined 482
names, Microsoft Access databases 462
SQL (Structured Query Language) 457, 462
Structured Query Language) SQL 457
Querying, OleType property 541
QueryUnload event
unloading MDI forms 307–308
Search Help for QueryUnload
Question mark (?)
Print method shortcut key 228
wildcards in file list boxes 429
Quit method, OLE Automation 554

R

R1C1 syntax. Object Linking and Embedding, linking to data within files 531
Radians
converting from degrees 349
defined 348
vs. degrees 348
Radius argument, Circle method 345, 347
Random-access files
accessing records 449
adding records 440–441
data types 435, 437–438
defined 436
deleting records 440–442
editing records 439–440
functions, listed 437
Get statement 439
Open statement 438–439
opening 438–439
overview 435
Put statement 440–441
record length 435–436
statements, listed 437
Type...End Type statement 437–438
vs. binary-access files 449, 451
writing text to files 445

Ranges, named, Object Linking and Embedding (OLE) 531
RDim statement
control variable arrays 192
object variables 183
Read-only operations, specifying in File Open 104
README.TXT file 3
ReadOnly property
databases, opening 458
discussed 462, 467
RECEDIT.MAK, Record Editor application 434
Record number, defined 436
Records
adding new
DB-UPDATABLEFIELD bit 472
discussed 472
Updatable properties 472
adding to databases 464
appending 472
current
defined 468
editing field values 473–474
managing 468
databases, defined 482
deleting 464, 466, 472, 475
editing 472
first, moving to 469
invalid 468
last, moving to 469
manipulating with 467
moving from record to record 469
new, adding 465, 472
next, moving to 469–470
previous, moving to 469, 471
relational database objects 481
retrieving 457
specific, moving to 471
valid 468
Recordset methods 469
Recordset Objects
accessing 491
data, manipulating 460
Recordsets
appending records 472
closing 476
Data Controls, opening with 457
moving around 469
moving from record to record 469
navigating 469
opening with Data Controls 457
records, appending 472
updating 471
RecordSource list 460
RecordSource property
Connect 467
DatabaseName 467
RecordSource property *(continued)*
databases, opening 458
discussed 454, 462, 467
Exclusive 467
Options 467
ReadOnly 467
RecordSource 467
setting 454
SQL queries 462
SWL queries 462
Rectangles, drawing with shape controls 322
Recursion, programming techniques 262
Recursive procedures, local variables 152
ReDim statement
dynamic arrays 173–174
Search Help for ReDim
Redrawing, screen images 358–361
Refresh method
databases
opening 467–468
opening manually 458
discussed 460, 467, 476, 481
first record, moving to 469
vs. DoEvents function 418
Search Help for Refresh
Registering OLE 2.0 on users' machine 548
Reinstalling printers 108
Relational database objects 481
Relational model 483
Relative movement
code example 330
defined 328
Remote databases, connecting to 457
Remove File menu command 116
RemoveItem method
code example, list boxes 55–57
removing grid rows 296
syntax 53
Removing files from Project 116
Requirements, Microsoft Visual Basic 2
Reserved words
brackets ([ ]), using with 135–136
defined 135
Help on 10
online Help 11
Search Help for reserved words
Resizing, controls 19
Resources, freeing with Close method 476
Resume statement
error handling 239
line label considerations 243
options defined 240–243
program execution and 245
RGB colors, hexadecimal values 371

RGB function
code examples 61–64, 319
DoColor procedure 63–64
increasing Color palette 373
setting colors 319
specifying color values 318
Search Help for RGB
Ribbon *See* Toolbar
Right function
determining root directory 28
Search Help for Right
Right mouse button, pop-up menus 91
Rollback command
discussed 477–478
transactions processing 477
Rolling back transactions 477
Rounding considerations 166
RowHeight property, defined 287
Rows and columns
adding rows 295–296
AddItem method 295–296
changing size 287
Cols property 285
ColWidth property 287
creating 286–287
default colors 285
fixed, defined 285
FixedCols property 296
FixedRows property 296
non-fixed, defined 285
RemoveItem method 296
removing rows 296
RowHeight property 287
Rows property, defined 286
scrolling options 285
Rows property, defined 286
Rows, relational database objects 481
RTrim function
fixed-length strings 167
Search Help for RTrim
RTRV110.DLL, distributing application files 579
Run time
bound controls, using at 464
changing AutoRedraw property 361
changing control state 67–68
creating embedded objects at 537–538
creating linked objects at 537
creating objects at 535
creating OLE objects 529
defined 39
embedded objects, creating at 537–538
embedded objects, empty, creating at 538
errors, trappable 475
grid controls
Run time *(continued)*
adding text 287–290
graphics 291
sizing 287
hiding controls 334–335
Menu Controls, creating dynamically 89
Menu items, creating dynamically 89
menus, controlling at 86
moving controls dynamically 326–330
objects, creating at 535
pictures, adding and removing 324
resizing forms/controls 333–334
Run time vs. Design time, creating objects 529
Run-time errors
code
returning with Err function 237, 239, 251
user-defined 246–247
DDE-generated 519–520
messages
returning with Error function 237
user-transparent 241
Search Help for trappable errors
Running code samples from Help 11
Running Setup 3

S

Sample applications
ALARM.MAK, timer controls 405
BUTTERF.MAK, simple animation 15, 23–26
CALC.MAK, Calculator 32, 69–70
CALLDLLS.MAK, DLLs 557
Click-A-Line 267–268
CONTROLS.MAK, controls 32
DDE.MAK, DDE 493
discussed 7
LOAN.EXE, Setup program 586
LOAN.MAK, LoanSheet 284–296
MDINOTE.MAK, MDI NotePad 297
MOUSE.MAK, mouse events 265
MULTINST.MAK, multiple instances 181
OBJECTS.MAK, object variables 181
PICTVIEW.MAK, Picture Viewer 15, 26–29
RECEDIT.MAK, Record Editor 434
Scribble
single-button version 268–271
two-button version 274–275
TEXTEDIT.MAK, Text Editor, file access methods 434
TIME.MAK, TimeCard 377
WINSEEK.MAK, File Seeker 425, 430–433
Samples
applications *See* Sample applications
BIBLIO.MDB database 454

Samples *(continued)*
code
discussed 7
running from Help 11
copying from Help 11
database (BIBLIO.MDB) 454
database application (DATAMGR.EXE) 454
database project (BIBLIO.MAK) 454
OLE2DEMO.MAK, Object Linking and Embedding (OLE) 523
SEUP1.MAK Setup program 573
Save argument 479
Save File As menu command 116
Save File menu command 116
Save Project As menu command 115
Save Project menu command 115
Save Project on Run, Environment options 120
SAVE_TO_FILE Action property 541
SavePicture function 466
Saving
applications 29
changes to tables 477
code as text file 118
code modules as text 116
data from OLE objects 541
forms as text 116
Projects 115
Scalable fonts, defined 378
Scale method 352, 357
ScaleHeight property 353–357
ScaleLeft property 353–357, 395
ScaleMode property 353, 386, 402
Scales
clearing standard settings 353
custom 353–357
defaults 352
defined 318
measurement *See* Measurement scales
standard settings 353
ScaleTop property 354–357, 395
ScaleWidth property 345, 353–357
Screen fonts, defined 378
Screen images, redrawing 358–361
Screen object
DDE uses 509
defined 206
moving/resizing forms 328
Search Help for Screen
Screen, default coordinates 318
Scribble application
single-button version 268–271
two-button version 274–275
Scroll bar control
See also Horizontal scroll bar control; Vertical scroll bar control
Scroll bar control *(continued)*
capturing user input 59–64
Change event *See* Change event
code example, setting colors with 61–64
events, listed 61
input value 60
Mouse Control Panel example 60
Scroll event 61
Value property *See* Value property
vs. ScrollBars property 41, 59–60
Scroll event 61
ScrollBars property
vs. scroll bar controls 41
word wrapping, text boxes 41
Searching Help 9
Seek function, file access uses 438, 443, 446
SelectCase statement, decision structure 143–144
Selected property, determining status 58
Selecting text 405–406
SelEndCol property, defined 293
SelEndRow property, defined 293
SelLength property
defined 406
Search Help for text
SelStart property
defined 406
Search Help for text
SelStartCol property, defined 293
SelStartRow property, defined 293
SelText property
defined 406
Search Help for text
Semicolon (;), using with Print method 384
SendKeys statement 521–522
Separator bars
creating in Menu Design window 84
Menu Controls 84
menus 78
Sequential-access files
defined 436
editing 443–444
functions, listed 442
opening 443
overview 435
statements, listed 442
text box controls 444
Set statement
assigning objects to object variables 195–196
code example 199
creating instances 198–199
SetAttr function, file access uses 438, 443, 446
SetAttr statement, setting file attributes 429
SetClientPrint procedure, Printer object and forms 397
SetData method
pasting Color palettes 375

SetData method *(continued)*
selecting data format 408–409
Search Help for SetData
SetFocus method
receiving focus 64
Search Help for SetFocus
SetFocus method, setting focus in dialog boxes 100
SetText method 406–408
Setting properties in code at run time 535
Settings box 20–21
Setup program
CopyFile function 586–587
CreateProgManGroup 586
CreateProgManGroup function 588
CreateProgManItem 586
CreateProgManItem function 588
customized look 577
described 2
discussed 577
distributing applications 573
files, compressing 583
installing application files 577
installing files 577
LOAN.EXE sample application 586
pre-installing files on users' machine 577
PromptForNextDisk function 586–587
running 3
SETUP1.FRM constants 587
SETUP1.FRM module 586
SETUP1.MAK
customizing 586
sample 586 code
sample program 573
template 582
SetupWizards 573
template, SETUP1.MAK 582
writing 586
Search Help for Setup Kit
Setup Toolkit, Visual Basic 577–583
SETUP.EXE
distributing applications 578
distribution disks, layout 584
installs files in SETUP.LST 582
Setup Toolkit 577
SETUP.LST
distributing applications 578
distribution disks, layout 584
SETUP.EXE installs files in 582
SETUP1.BAS, Setup Toolkit 577
SETUP1.EXE
distribution disks 574
distributing application files 578
Setup Toolkit 577
SETUP1.FRM constants (list) 587
SETUP1.FRM module
CopyFile function 586
CreateProgManGroup 586
CreateProgManItem 586
PromptForNextDisk function call 586
Setup program 586
SETUP1.MAK
CopyFile function 586
CreateProgManGroup 586
CreateProgManItem 586
customizing 586
distributing applications
discussed 582, 586
sample code 585
GetSystemDirectory() Windows API function 585
GetWindowsDirectory() Windows API function, calling 585
PromptForNextDisk function call 586
sample Setup program 573
SetupKit, installing OLE 2.0 545
SETUPKIT.DLL
distribution disks, layout 584
Setup Toolkit 577
SetupWizard
distributing applications 548, 578
distributing OLE applications 545
Distribution Files list box 574
Search online Help 574
Shape control
naming conventions 36
Toolbox icon 33–34
SHARE.EXE
distributing OLE applications on users' machine 545
on users' machine 548
Shared application resources, installing on users' machine 584–585
Sharing data between applications 547
Shell function
launching other applications 513–514
Search Help for Shell
SHELL.DLL, distributing applications 585
Shift argument
code example 276
defined 266
mouse events and 276
testing
ALT-key status 276
SHIFT-key status 276
using with mouse events 271–274
SHIFT key
keyboard handling 419–420
testing status 276
Shortcut Keys
assigning to menu items 85–86
defined 420

Shortcut Keys *(continued)*
introduced 86
selecting option buttons 47
Show method
dialog boxes 94
displaying object variables 182, 188
hiding/showing controls 334–335
programming techniques 257, 258
Search Help for Show
Simple combo boxes 50–51
Single data type 164
Single Step, debugging 210, 218–219
Size of arrays 555
SizeMode property, displaying OLE objects 530
Snapshots 485
Solid lines 366–367
Solutions, custom, with OLE Automation 547
Sorted property 52–53
Source argument, DragDrop event 279
Source code, managing 116
SourceDoc property 533–537
SourceItem property 531, 535–537
Sources 494–496
Spaces
stripping leading 49
trailing 167
Specific object types
control types, listed 186–187
defined 184
forms and 185
generic vs. 187
naming conventions 186
New keyword 188
passing as arguments 193
Splitting files between distribution disks 584
SQL (Structured Query Language)
discussed 482
queries
discussed 457
RecordSource property 462
SQLSETUP.DLL, distributing application files 579
SQLSRVR.DLL, distributing application files 579
SQORA.DLL, distributing application files 579
Squares, drawing with shape controls 322
Stacks, programming techniques 261–262
Start Up Form, Project options 120
Startup
GRID.VBX 121
OLDCLIEN.VBX 121
Startup forms, programming techniques 257–258
State argument, using with mouse events 281
Statements, End 476
Static colors, defined 372
Static statement
control variable arrays 192
Static statement *(continued)*
declaring variables 165
local arrays 171
local variables 151–152, 154–156
object variables 183
Static variables
declaring local variables 151–152, 154–156
defined 154
programming techniques 155–156, 262
Static statement 151–152, 154–156
Status bar, creating 315
Step keyword 341–345
Stop statement
entering break mode 214, 217
error-handling uses 246
executable files and 217
terminating applications 126
vs. breakpoints 217
Search Help for Stop
STORAGE.DLL
distributed with applications 548
distributed with OLE applications 544
distributing application files 579
Str$ function, compared with Format$ function 389
Stretch property
described 333
expanding pictures 325
String data type 164
String$ function, reading binary-access files 449
Strings
See also .DLL; Text box controls
comparing 167
concatenation vs. addition 159–160
converting into numbers 49
converting numbers into 137
data types 159–160, 166–167
default, variable-length 166
fixed-length strings 166-167, 262
stripping leading spaces 49
trailing spaces 167
variable-length strings 166
Variant variables 159–160
Strings, collections containing, limitations in Visual Basic 556
Structure, databases
examining 457
working with 483
Structured Query Language (SQL) 482
Sub procedures
Call statement 131
code example 131
data types 131
DLL uses 558
MDI NotePad application 315
passing user-defined types 566–567

Sub procedures *(continued)*
startup alternatives 125
syntax 130–131
Variant data type 130
vs. Function procedures 127, 130
Search Help for Sub procedures
Submenu controls 83
Submenu item, Menu control position 78
Submenu title, Menu control position 78
Submenus 82, 84
Subobjects, objects contained in 553
Symbols, number formatting uses 389
Syntax
defined 123
object property 552
System objects
See also Application object (App)
Application object (App) defined 206
Clipboard object *See* Clipboard object
Debug object *See* Debug object
object variables and 206
passing to procedures 206
Printer object *See* Printer object
scope 206
Screen object *See* Screen object
Search Help for App
System palette
Color palette limitations 375
defined 372
static colors 372
z-order effects, described 376
System requirements 2

T

Tab character, delimiting cell data 289
Tab function
code example 388
syntax 388
Search Help for Tab
TAB key, keyboard handling 420
Tab order
changing tab order 66
defined 65
removing control from 67
TabIndex property *See* TabIndex property
TabStop property *See* TabStop property
Tab stop width, Environment options 120
TabIndex property
changing tab order 66
control array attributes 336
controls included 66
creating access keys 40
values 66
TableDef objects 485-489
TableDefs Collections 485
Tables
attached, Microsoft Access databases 460
changes, saving 477
Data Manager, creating with 485
databases, defined 482
Deny-Read options 460
Deny-Write options 460
denying access to, Recordsets options 460
mapping 488
names 462
Recordsets Options, denying to 460
relational database objects 481
saving changes to 477
undoing changes to 478
Tables, displaying tabular data 387–388
Tabs
default 387–388
setting 388
TabStop property
option button groups 67
removing controls from tab order 67
Tag property 280
Search Help for Tag
Target argument, DragDrop event 279
Testing distribution disks 590
Text
See also Grid control graphics; Text box control
Clipboard object 406–411
Cut, Copy, and Paste 407–408
formatting
fonts 379–382
printing to forms and picture boxes 383–387
grid control *See* Grid control text
selecting text 405–406
Search Help for text
Text box control
See also Word wrapping
creating access keys 40
destinations *See* DDE
discussed 463
editing files with 444
embedded quotation marks 42–43
intercepting keystrokes 42
naming conventions 36
password text boxes 41–42
Print# statement 445
printing to file 445
selecting text 405–406
SelLength property, defined 406
SelStart property, defined 406
SelText property, defined 406

Text box control *(continued)*
Toolbox icon 33
vs. label control 38
when to use 41
word wrap ability 41
Text boxes, binding to data control 454
Text colors, Environment options 120
Text Editor (TEXTEDIT.MAK) sample application
dialog boxes, About dialog box 93
discussed 75
Edit menu controls 80
UpdateMenu general procedure 90
Text files 117–118
Text property
code example 54
identifying list selections 54
Text, representing forms as 116
Text, saving as
code modules 116
forms 116
TEXTEDIT.MAK, Text Editor sample application 75, 432
TextHeight method
code example 387
described 386
syntax 386
Search Help for Text
TextWidth method
code example 387
CurrentX property 386
described 386
Search Help for Text
Third-party developers custom controls 114
Time
code example, formatting time 392
formatting
international syntax 391–392
syntax 391
using named formats 389–390
Search Help for Now
Time function, displaying time 414
TIME.MAK, TimeCard application 377
Timer control
BUTTERF.MAK, simple animation application 23–26
code example, Alarm Clock 414–416
Enabled property 412
initializing 412–413
Interval property *See* Interval property
naming conventions 36
placement on forms 24, 412
Timer event 413-416
Toolbox icon 33–34
vs. DoEvents function 418
Timer event 413–416
Title bar introduced 76
To keyword, array lower bounds 172
Toggling 335
Toolbar
Align property 313, 315
command buttons 313
controls on 314
creating 313–314
defined 16
icons, listed 16–17
image controls 313–314
MDI NotePad application 314–315
overview 313–315
picture box control 313
user-defined 38
Toolbar items, online Help 10
Toolbox
bound controls 463
defined 17
directory *See* Directory list box controls
drive *See* Drive list box control
file *See* File list box control
file-system icons 425
icons (list) 32–34
online Help 10
Tools, external, for generating forms 116
Top property 326–328
ToPage property, Print dialog box 108
Topics *See* DDE
Transactions 477–478, 488
Transactions processing 477
Transactions property 477
Trappable errors
databases, opening 468
discussed 466–467, 470–471
run time 475–476
Trim function
fixed-length strings 167
Search Help for Trim
TrueType fonts, described 378
Twips
changing with ScaleMode property 386
defined 353
effects on movement 330
inch and centimeter equivalents 386
measuring graphics 317
moving/resizing forms 309
pixel size, monitor-related 402
relative movement 328
Type statement
code example 437–438
Search Help for Type
TypeOf keyword
object variable types 194–195
restrictions 195
using with mouse events 280
Search Help for TypeOf

Types
.BAS (code module) files 114
.FRM (form) files 114

U

UBound function 555
collection restrictions 203
control variable arrays 192
Undoing changes to tables 478
Unique Index property 490
Unload statement
adding/removing controls 335–338
closing forms 126
code example 338
control array uses 71–73
Dialog boxes, display options 102
Menu control 90
removing controls 336
restrictions 71
Updatable database properties 488
Updatable property 471–472
Update method 465, 472–476, 481
UpdateControls method 466, 481
Updated documentation 3
UpdateMenu general procedure, Text Editor sample application 90
UpdateRecord method 479–481
Users' machine
installing files 584
OLE 2.0 registration 545
pre-installing files 577
SHARE.EXE, distributing OLE applications 545
User-defined types
See also .DLL
examples 177, 306–307
explained 176–178
passing as arguments 178
vs. Variant 178
User-defined types, limitations in Visual Basic 554

V

Val function
converting strings to numbers 49
Search Help for Val
Valid records 468
Validate event 476, 479–480
Validating changes to databases 479
Validation
controlling 476
Validation *(continued)*
fields 473, 476
Value property
selecting option buttons 47
using with scroll bars 60
Values, property, retrieving from objects 553
Variable declarations required, Environment options 120
Variable-length strings 166
Variables
See also Data types; Variant data type; Variant variables
.BAS (code module) files 114
command line 113
Const statement 178–180
data types 140
declaring 139–140, 148–149
Dim statement *See* Dim statement
explicit declaration 150–151
.FRM (form) files 114
global 152, 154, 178–180
implicit declaration 149–150
local *See* Local variables
misnamed 150–151
module-level *See* Module-level variables
naming conventions *See* Naming conventions
object *See* Object variables
programming techniques 255, 260, 262
recursion procedures 152
scope, described 151–153
static 151–152, 154–156
symbolic constants 178–180
Variant data type
converting numbers into strings 137
Format function 389
Function procedures 132
OLE Automation 553
Sub procedures 131
Time function 414
variables defined as 140
Variant variables
concatenation vs. addition 157
converting 159
data type 149, 164
date/time values 160–162
default data type 149, 165
Deftype statement 165
empty value 162
Format function 159
internal representation 157–158
IsDate function 162
IsEmpty function 162
IsNull function 163
IsNumber function 158
Null keyword 163
null value *See* Null
numeric values 158–159
overview 156–157
programming techniques 255, 259

Variant variables *(continued)*
String values 159–160, 157, 167
VarType function 49–50
VarType function
determining Variant data type 157–158
Search Help for VarType
VB.INI (environment options) file 119
VBAPI.HLP, online Help 5
VBDB300.DLL, distributing application files 579
VBKNOWLG.HLP, online Help 5
VBOA300.DLL, distributing applications 544, 548, 579
VBRUN300.DLL, distributing applications 578, 584
.VBX (custom control) files *See* Custom controls
VER.DLL, distributing applications 577–578, 584, 589
Verb property
discussed 539
Search Help for Verb
Verbs
displaying an objects' 539
fetch 539
objects' default 539
OLE objects 533
Version stamping resources, Setup program 577
Vertical scroll bar control
naming conventions 36
Toolbox icon 33–34
VGA monitor, pixel size (in twips) 402
VGA palette
256 colors 372
adapting for monitors 373
default 372
View Procedures dialog box, switch between procedures 28–29
Visible property
adding/removing controls 336
changing run-time focus 67–68
code examples, adding/hiding controls 336
control arrays
attributes 336
settings 71
displaying controls 25
hiding/showing controls 334–335
setting focus requirements 65
toggling 335
Visible property, Menu controls 89
Visual Basic *See* Microsoft Visual Basic
Visual Basic Setup Toolkit 577
Visual Basic SetupWizard 573

W

Watch expressions 223–226
Watch pane
defined 222
Watch pane
watch expressions 223-226
Width property
described 333
dimension measurement 355
moving/resizing forms 318
resizing controls 333
Wildcards, using in file list boxes 429
WIN.INI file, Print dialog box 109
WIN31API.HLP, online Help 5
Window menu 311–313
Window, Project 113
Windows
controlling screen image movement 358–361
resizing and graphics 360
Windows 3.1 API, online Help 5
Windows API files, GetWindowsDirectory() 585
Windows applications, interoperability between 523
Windows Program Manager groups, creating with Setup program 577
WINHELP.EXE, Dialog boxes 103
WINSEEK.MAK, File Seeker application 425, 430–433
WINSYSNEEDED, SETUP1.FRM constants 587
Wizards, Setup 573
.WMF (metafile) files 321
Word wrapping, text boxes 41
WordWrap property, vertical label sizing 39–40
Write# statement
delimiting fields 445
writing text to files 445
Search Help for Write#
WYSIWYG, defined 378

X

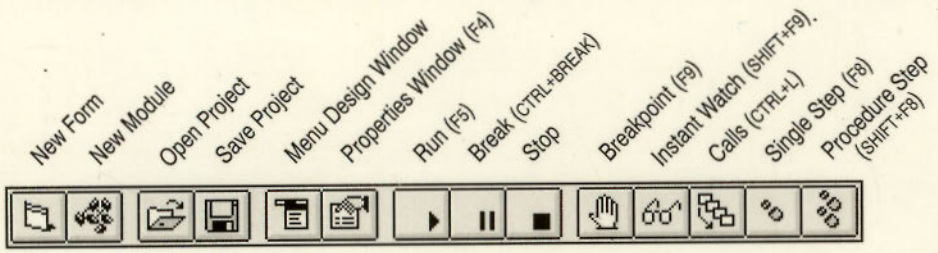
X argument
defined 266
DragDrop event 279
XBS110.DLL, distributing application files 579
XETUP.LST, Setup Toolkit 577
Xor Pen, DrawMode setting 368–369

Y

Y argument
defined 266
DragDrop event 279

Microsoft®

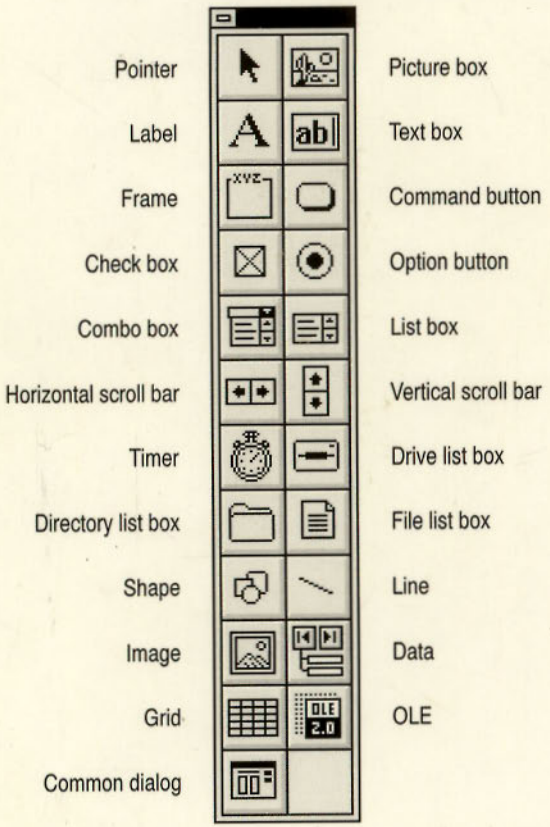
The Toolbar



Getting Help

Press F1 or choose Contents or Search from the Help Menu.

The Toolbox



Keyboard Shortcuts

Description	Key
View Procedures	F2
Go to objects list box	CTRL+F2
Go to procedure	SHIFT+F2
Find	CTRL+F
Find Next	F3
Properties Window	F4
Cut current line	CTRL+Y
Insert blank line	CTRL+N
Next Procedure	CTRL+DOWN
Previous procedure	CTRL+UP
Clear to end of line	CTRL+DEL
Cut to end of line	SHIFT+CTRL+DEL
Shift one screen right	CTRL+PAGE DOWN
Shift one screen left	CTRL+PAGE UP
Beginning of procedure	CTRL+HOME
End of procedure	CTRL+END
One word right	CTRL+RIGHT
One word left	CTRL+LEFT

* 5 1 4 1 0 *

Programmer's Guide

Microsoft® Visual Basic™