FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

A LOS ASISTENTES A LOS CURSOS DE LA DIVISION DE EDUCACION CONTINUA

Las autoridades de la Facultad de Ingeniería, por conducto del Jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia sólamente a los alumnos que tengan un mínimo del 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el periodo de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

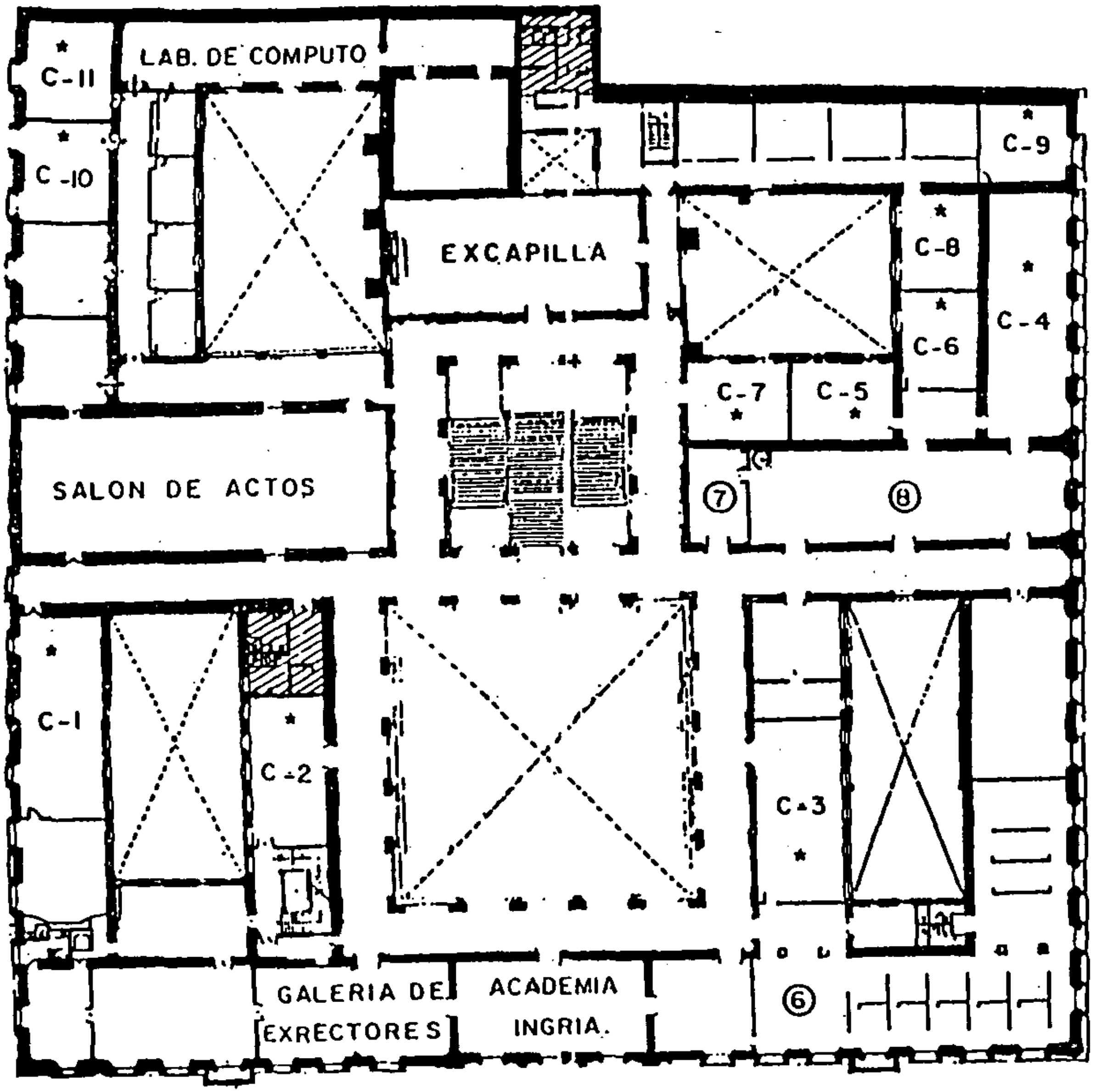
Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases, a efecto de no llenar en la última sesión las evaluaciones y con ésto sean más fehacientes sus apreciaciones.

¡ GRACIAS !



GUIA DE LOCALIZACION

1 – ACCESO

2 – BIBLIOTECA HISTORICA

3 – LIBRERIA U N A M

4 – CENTRO DE INFORMACION Y DOCUMENTACION "ING. BRUNO MASCANZONI"

5 – PROGRAMA DE APOYO A LA TITULACION

★ AULAS

6 – OFICINAS GENERALES

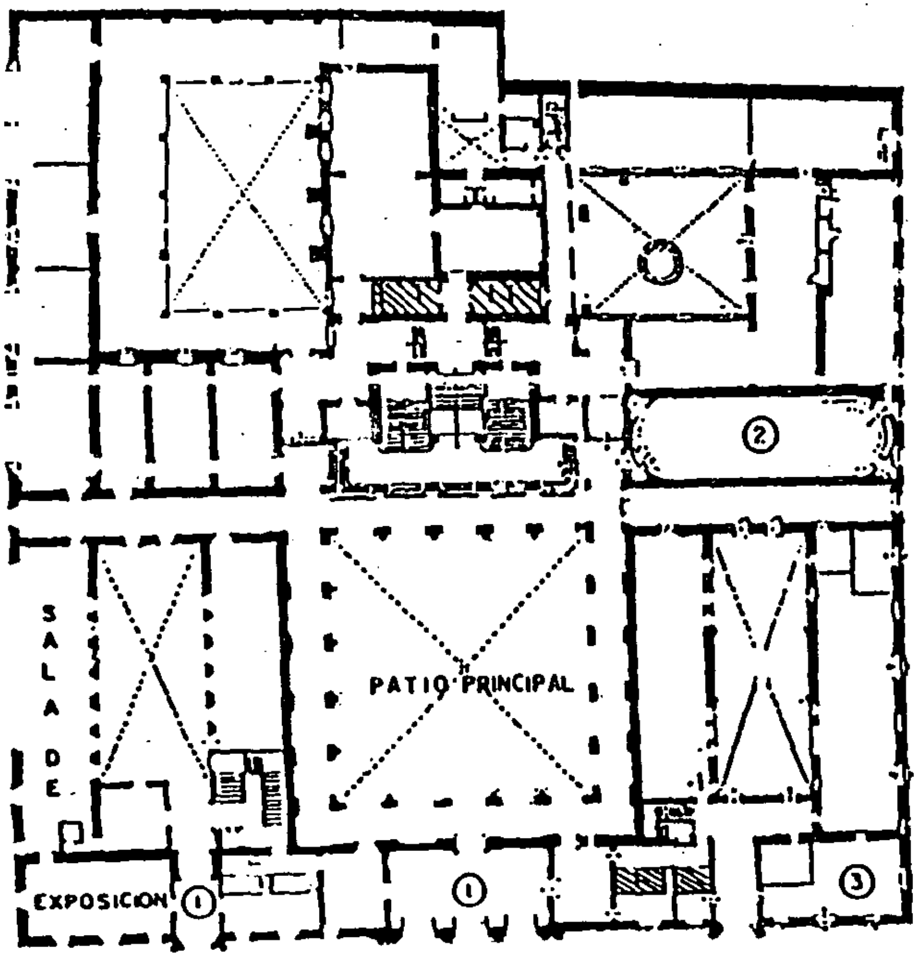
7 – ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA.

8 – SALA DE DESCANSO

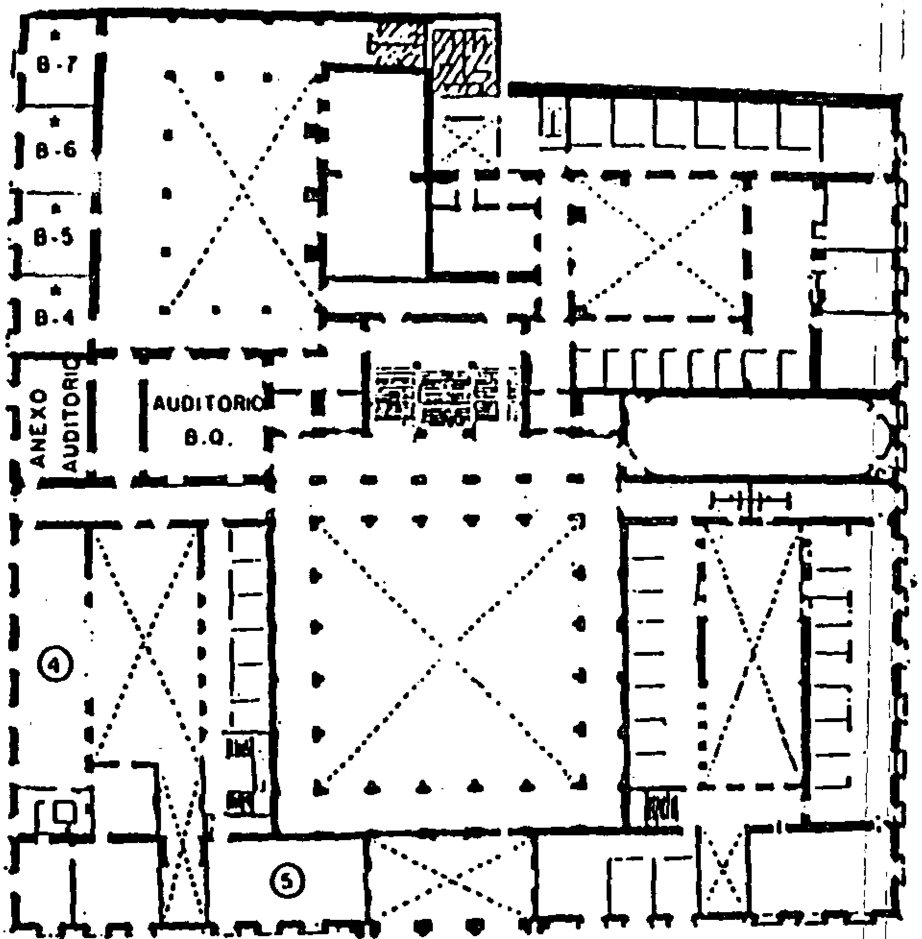
▨ SANITARIOS

1er. PISO

PALACIO DE MINERIA



PLANTA BAJA



MEZZANINNE

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.
CURSOS ABIERTOS

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

INSTITUTO MEXICANO DEL PETROLEO

VISUAL BASIC

Del 7 al 18 de Noviembre

Ing. Andres Monterrubio

Palacio de Minería

1994

Module 1: Using Visual Basic

∑ Overview

- Visual Basic Tools
- Building a Simple Visual Basic Application
- Visual Basic Menu Commands

Overview

The purpose of this module is to introduce you to the key elements of Visual Basic. This is not intended to be an exhaustive review of all functions and tools; rather, it is designed to get you up and running on the product. Later modules will flesh out all of the details that you need to know to become Visual Basic programmers.

The best way to learn about Visual Basic is to review the individual functions in each major portion of the Visual Basic interface and then walk through the steps you should follow to develop an application.

The module is divided into three major sections. The first major section is a lecture about the elements of the Visual Basic application that you use in the creation of applications. The second is a demonstration where the trainer walks you through the application design process and you end up with a compiled executable file. The final section is a lab that gives you hands-on practice using some of the tools that you have just been introduced to.

Prerequisites

There are two key skills that you need for success in this module:

- Practice in using a mouse
- Understanding of graphically based applications

Overall Objective

At the end of this module, you will have successfully created your first Visual Basic application.

Learning Objectives

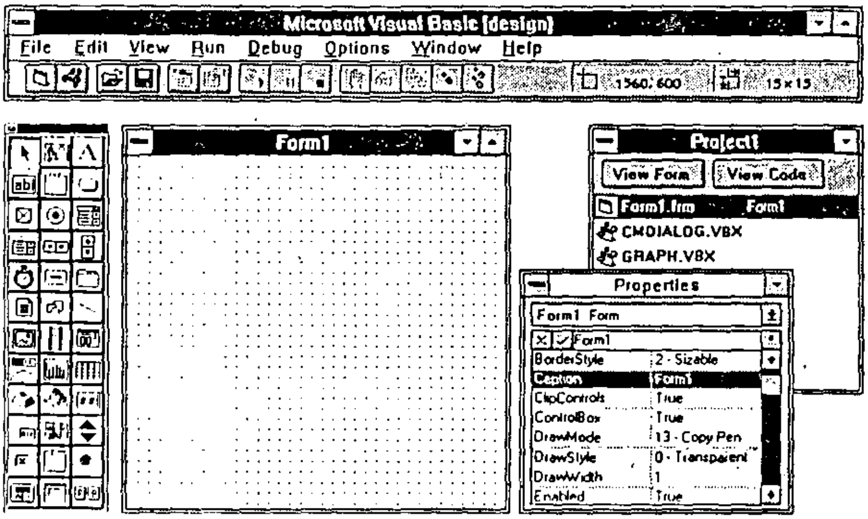
At the end of this module, you will be able to:

- Design an application user interface using the Form window, Toolbox, toolbar, and Property window.
- Name and save the application's forms and project files.
- Start and stop an application from the Visual Basic menu and/or the toolbar.
- Create an executable file and add it to a Windows group.

∑ Visual Basic Tools

- Form and Project Windows
- The Toolbox and Properties Window
- The Toolbar and Code Window

Form and Project Windows



Form1 Form

Form1

BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Transparent
DrawWidth	1
Enabled	True

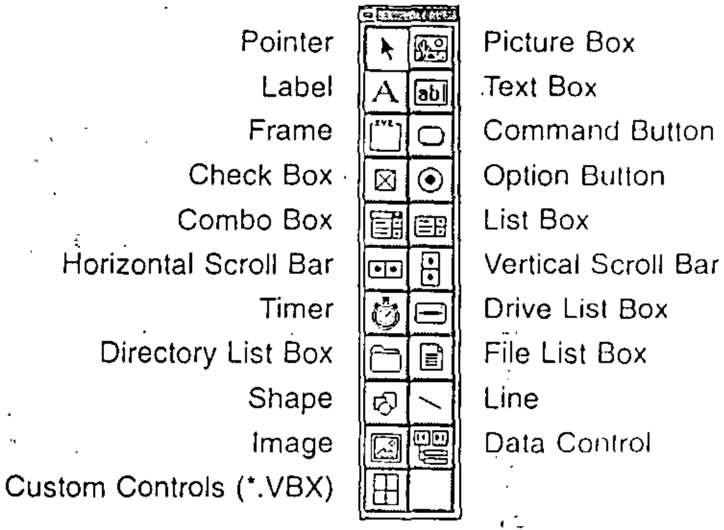
What Is a Form?

Forms are the heart of a graphically based application. They are what the user sees and interacts with in order to accomplish some task. They are also the place where you begin to build applications; on them you place controls—command buttons, list boxes, option buttons—that present the user with the choices that they have.

What Is the Project Window?

The Project window is a list that Visual Basic uses to keep track of the forms that you are using for your application. You will have as many .FRM files listed in the Project window as you have forms in your application. In addition you may have other files—.BAS and .VBX files—but we will hold off talking about those for a little while.

The Toolbox and Properties Window



What's in the Toolbox?

As the name suggests, the Toolbox is where you go to get the basic elements of every Windows-based application you create in Visual Basic.

There are two ways that you can place controls on a form, either by double-clicking the control tool in the Toolbox or by clicking and dragging the control. For the most part, either method has the same result. When you double-click a tool, the control shows up in the middle of the form; so you then have to drag the control to the correct position on the form.

Each control that you place on a form has a set of properties that you can use in order to get the right "look and feel" for your application. For the time being, all you need to know is that there are properties to each control. In later modules you will get a much closer look at the most commonly used properties for most of the tools in the Toolbox.

Custom Controls

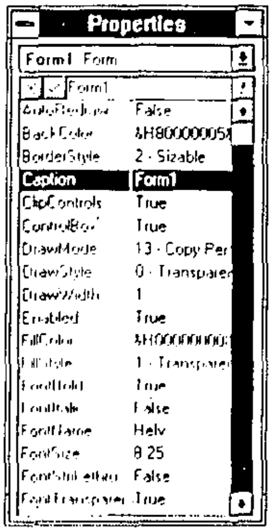
You can add controls to the Toolbox. Some third parties have created custom controls that you can purchase separately. The Professional edition of Visual Basic contains various custom controls.

Σ To Add a Custom Control

1. From the File menu, choose Add File.
2. Select the appropriate file. Custom Control files have a .VBX extension.
3. Choose OK.

What Are Properties and How Are They Set?

Properties for a control are set using the Properties window:



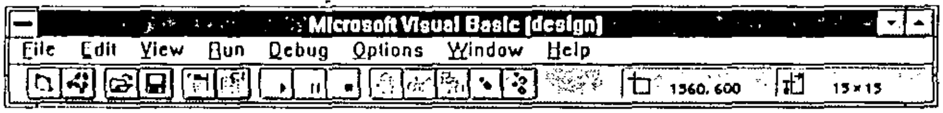
AutoRedraw	False
BackColor	&H8000005
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Per
DrawStyle	0 - Transpare
DrawWidth	1
Enabled	True
FillColor	&H0000000
FillStyle	1 - Transpare
FontBold	True
FontItalic	False
FontName	Helv
FontSize	8.25
FontStrikethru	False
FontTransparent	True

∑ To set properties

1. Click the control whose properties you want set.
2. Scroll the Properties list on the Properties window until you find the property you want to set and select it.
3. Place the value for the property in the Settings combo box.
4. Click the check box to the left of the Settings combo box.

What Is the Toolbar?

The toolbar provides quick access to common commands or functions. These functions—like saving projects and starting the application during the design phase—are also available from the Visual Basic menus as well as through access and shortcut keys.



Nine of the relevant items on the toolbar are.

Toolbar	Menu path	Shortcut keys
New Form	From the File menu, choose New Form.	n/a
New Module	From the File menu, choose New Module.	n/a
Open Project	From the File menu, choose Open Project.	n/a
Save Project	From the File menu, choose Save Project.	n/a

Toolbar	Menu Path	Shortcut Keys
Menu Design window	From the Window menu, choose Menu Design.	CTRL + M
Properties window	From the Window, menu, choose Properties.	F4
Start	From the Run menu, choose Run.	F5
Break	From the Run menu, choose Break.	CTRL + BREAK
End	From the Run menu, choose End.	n/a

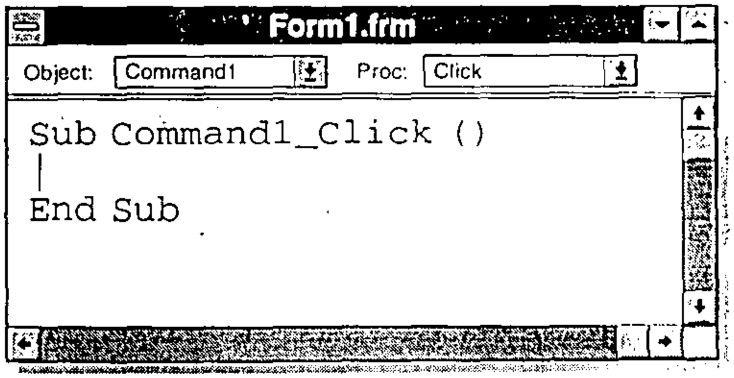
Note The rest of the functions deal with debugging and will be covered during that module.

Position and Size Coordinates

Visual Basic displays the coordinates for the upper-left corner for each control on the form relative to the inside upper-left corner of the form. It also displays the width and height of the control. You can set these values by placing the form or control approximately where you want it, or you can specify Left and Top coordinates from the Properties list. The unit of measurement is in twips (there are 1,440 twips in an inch). You will find out later that you can change this unit of measurement to something you are more familiar with.

What Is a Code Window?

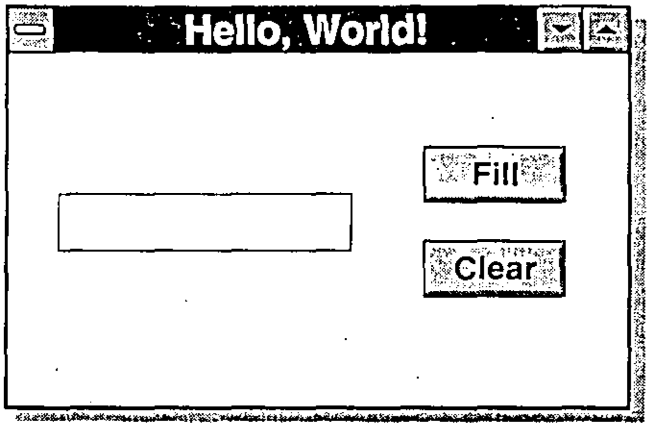
Code windows display the code that implements your application. At first Code windows only contain the template for procedures and functions; you will add more code to them as you develop your application. There are several ways to open a Code window. The easiest way is to double-click the object whose code you want to view. For example, to locate the Click event template for a command button, double-click the command button on the form during design time.



Object: Command1 Proc: Click

```
Sub Command1_Click ()

End Sub
```

Building a Simple Visual Basic Application



Walk Through—Building a Visual Basic Application

∑ To start WORLD

1. From the Walk Throughs program group, start WORLD.

   The purpose of this little application is to give you a chance to walk through most of the major steps needed for developing a Visual Basic application.

   How does the application work?

   It has one text box and two command buttons.

2. Choose the Fill button.

   When the user clicks the button, text is revealed in the text box.

3. Choose the Clear button.

   When the user clicks this button, the text in the text box is cleared.

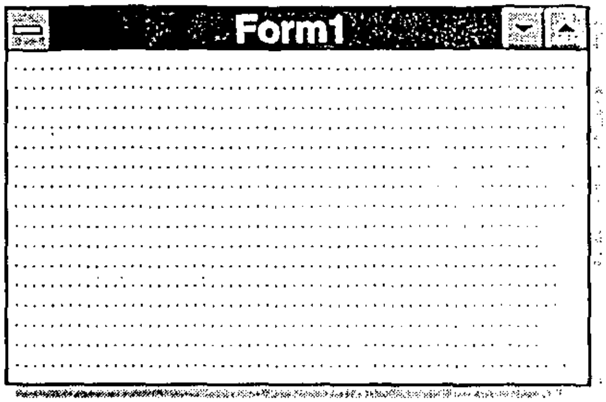
4. Quit the WORLD application.

   From the Control menu, choose Close.

∑ To start Visual Basic

1. Double-click the Visual Basic icon.

   Visual Basic will start with a blank form on the screen.



2. Resize the form to:

Height	2700
Width	4065

∑ To design the base form

The first step in designing the user interface is to set the properties for the application's base form. In this case, set the properties for the Hello, World! form.

1. In the Properties window Properties list, select BackColor.
2. Click the ellipsis (...) at the end of the text box.
3. Select a shade of green.

   The background of the form turns green. Selecting any color will automatically hide the color palette.

4. In the Properties list box, select Caption.
5. In the text box, type **Hello, World!**

   The Form title bar will contain Hello, World!

6. In the Properties window Properties list, select Name.
7. In the text box, type **frmWorld**

∑ To add controls and set properties

To create the form's controls, double-click the appropriate buttons in the Toolbox.

1. Double-click the command button tool in the Toolbox.
2. Set the following properties:

Caption	Fill
Name	cmdFill
FontSize	18

3. Move the Fill button to the upper-right corner of the form.

4. Double-click the command button tool in the Toolbox.
5. Set the following properties:

Caption	Clear
Name	cmdClear
FontSize	18

6. Move the Clear button to the lower-right corner of the form.
7. Double-click the text box tool in the Toolbox.
8. Set the following properties:

Name	txtBox1
Text	Delete any text in the text box for this property.

∑ To add the Basic code to enable the controls

In this step, code will be added to activate the functionality of the controls in the form. The code will enable the text box to be filled with the words Hello World! when the user clicks the Fill button and enable the text box to be cleared when the user clicks the Clear button.

1. Double-click the Fill button that you just created to open the Code window.

When you double-click this command button, Visual Basic brings you directly to the template for a procedure that responds to a Click event for this control. The names of event procedures follow the pattern:

```
Sub objectname_eventname ()
```

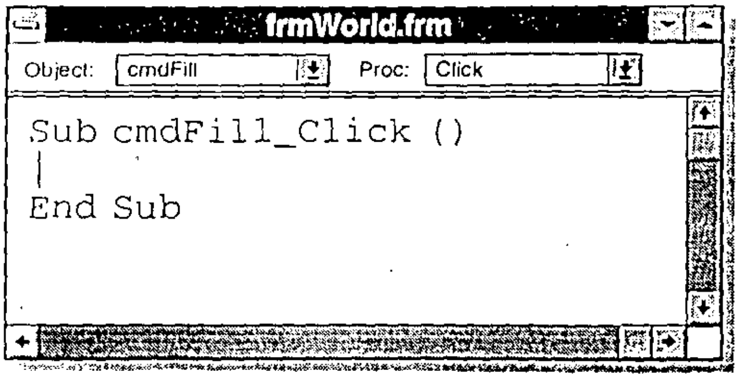
In this case the procedure is named:

```
Sub cmdFill_Click ()
```

Also note that Visual Basic provides the statement that ends the subroutine:

```
End Sub
```

In the graphic below, notice that the object name shows up in the Object list box on the left, and the event name shows up in the Procedure list box on the right.



frmWorld.frm

Object: cmdFill Proc: Click

```
Sub cmdFill_Click ()

End Sub
```

2. Add the following code:

   ```
   txtBox1.Text = "Hello, World!"
   ```

   Add a line of code between the two lines of the template for the Click event procedure. This code gives the application the target for the action (filling the text box) and the contents that you want assigned to the text box. In this case, "Hello, World!" is the contents.

3. In the Object list box, select cmdClear.

4. Add the following text:

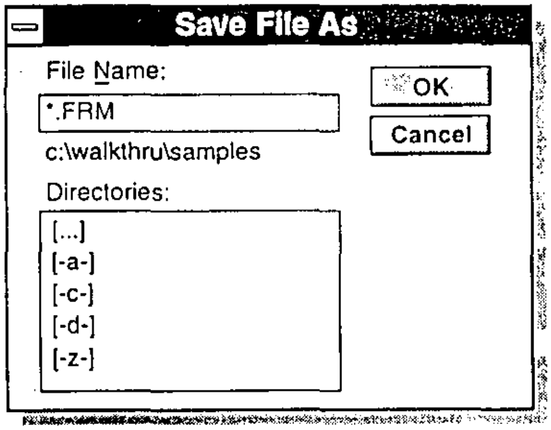
   ```
   txtBox1.Text = ""
   ```

   Inside the template for this Click event procedure, add a line of code to give the application the target for the action (clearing the text box) and the contents that you want assigned to the text box. In this case the contents is an empty string ("").

Σ To save your work

Before you create an executable version of this Visual Basic application that you can run directly in Windows, you should save the source code to disk. You will do this through two actions—saving the form and saving the project.

1. From the File menu, choose Save File As.



File Name: *.FRM

c:\walkthru\samples

2. Make the current directory C:\WALKTHRU\SAMPLES.
3. Save the file as WORLD.FRM.
4. Choose OK.
5. From the File menu, choose Save Project As.
6. Save the file as WORLD.MAK.
7. Choose OK.

∑ To make an executable file

To create an executable file, make sure the source code for the project is open in Visual Basic.

1. From the File menu, choose Make EXE File.
2. Make the current directory C:\WALKTHRU\SAMPLES.
3. Save the file as WORLD.EXE.
4. Choose OK.
5. Minimize Visual Basic.

∑ To start the executable from Windows

You can start an executable file from Windows in two ways: 1) create a program group, and add the name of the file to the group; or 2) start Program Manager, open the File menu, and choose the Run command. We'll use the first method.

1. From the Program Manager File menu, select New.

   The New Program Object dialog box appears.
2. Select the Program Group option.
3. Choose OK.

   The Program Group Properties dialog box appears.

   In the Group Properties Description control, name the group something like SAMPLES.
4. In the Group File control, type **SAMPLES**
5. Choose OK.

   A blank window will appear.
6. From the Program Manager File menu, choose New.

   The New Program Object dialog box appears with the Program Item option selected.
7. Choose OK.

   The Program Items Properties dialog box appears.
8. Name the application.

   In the Description control, name the application WORLD or another name that reflects its function.

   Now comes the tricky part. You need to help Windows locate the executable file that you created.
9. Click the Browse button.

   The Browse window will appear.
10. Select drives and directories to locate WORLD.EXE.

    It should be in C:\WALKTHRU\SAMPLES.

11. Choose OK.

    The Program Items Properties dialog box appears with WORLD.EXE in the Command Line text box.

12. Choose OK.

    The SAMPLES Program Group will appear containing WORLD and a program icon.

∑ To run your application

- Double-click the World icon.

  The Hello, World! application will start in the same screen location where the form was created.

∑ To stop your application

1. Click the Control menu in the upper-left corner of the Hello, World! form.

   A menu will appear.

2. Choose the Close command.

   Windows closes the application for you.

Visual Basic Menu Commands

- Managing Forms and Projects: The File Menu
- Editing Visual Basic Code: The Edit Menu
- Testing Applications During Development: The Run Menu
- Visual Basic Window Management: The Window Menu
- Getting More Information: The Help Menu

In order to develop the simplest of applications, there are a number of Visual Basic commands that you need to know about. Below is a brief listing of these commands. The list is not exhaustive; there are a couple of topics that have been left to a later module.

Managing Forms and Projects: The File Menu

- Adding a New Form
  File menu, New Form
- Adding an Existing Form to a Project
  File menu, Add File
- Deleting a Form from a Project
  File menu, Remove File
- Making an Executable File
  File menu, Make EXE File
- Printing Code
  File menu, Print, select Code option
- Printing Forms
  File menu, Print, select Form option
- Saving a File and Naming It
  File menu, Save File As
- Saving a Form
  File menu, Save File

Note Saving a file does not mean that the project is also saved.

- Saving a Project
  File menu, Save Project
- Saving a Project and Naming It
  File menu, Save Project As

Note A Walk Through showing you how to save a text version of the code is located at the end of this portion of the module.

Editing Visual Basic Code: The Edit Menu

- Searching Code for a Text String
  Edit menu, Find
- Searching and Replacing a Text String
  Edit menu, Replace
- Cutting and Pasting Text
  Edit menu, Cut, Copy, and Paste
- Undoing Changes
  Edit menu, Undo

Testing Applications During Development: The Run Menu

- Starting the Application
  Run menu, Start

  Notice the change in the title bar when you select this option to indicate that you are now in Run mode.
- Stopping the Application
  Run menu, End

Visual Basic Window Management: The Window Menu

- Displaying the Properties Window
  Window menu, Properties
- Displaying the Toolbox
  Window menu, Toolbox
- Displaying the Project Window
  Window menu, Project Window
- Displaying the Color Palette
  Window menu, Color Palette

Getting More Information: The Help Menu

- Using the Help Table of Contents
  Help menu, Contents
- Searching for a Specific Topic
  Help menu, Search
- Locating Product Support Information
  Help menu, Product Support

Walk Through—Saving a Text Version of Code

Procedural programmers are used to seeing all of their code in one place. Remember, however, that the code is not executed in the order it appears in the text file you create here. It is still event-driven code, and the flow of execution still depends on user and system events.

∑ To save a text version of code

1. Restore Visual Basic.
2. From the File menu, choose Open Project.
3. Open WORLD.MAK, located in \WALKTHRU\SAMPLES.
4. Choose OK.

   Load the sample project that you have just completed, but do not run it.
5. From the Project window, select WORLD.FRM.

   Make sure that you select the correct file. When you load a project, the first file in the list is selected by default.
6. From the File menu, choose Save Text.

   Visual Basic will give your text file the same name as the form, in this case WORLD.TXT. That will probably do in most cases. Make sure that you save the file to the \WALKTHRU\SAMPLES subdirectory.
7. Choose OK.
8. Minimize Visual Basic.
9. Return to Program Manager.
10. In the Accessories group window, open Notepad.
11. From the File menu, Choose Open.

    Use the Open dialog box to locate WORLD.TXT.
12. Choose OK.

    This loads WORLD.TXT into Notepad so that you can review the code.

    If you want to print code directly to a printer, choose the Print command from the Visual Basic File menu.

    You can print just the form or code or both for the current file, or you can print all forms or code or both for your entire application. Remember, however, that .BAS files do not have a form associated with them, so in this case you will only be able to print code.

∑ To save a text version of properties and code

You can also save the form with the properties and code in a text file.

1. From Visual Basic, open the project WORLD.MAK in \WALKTHROUGH\SAMPLES.
2. From the Project window, select WORLD.FRM
3. From the File menu select Save File As.
4. Select the check box Save As Text and choose OK.
5. The form is saved in text format. The extension .FRM is used.
6. Use Notepad to compare the two text files.

Walk-Through—Using Visual Basic Help

∑ To use Help

1. From the Visual Basic Help menu, choose Contents.

   The Visual Basic Help window appears. This window provides a brief topical tour of the major components of Visual Basic.

   A topical list that might be of interest to you is Programming Language.

2. Choose the Programming Language topic.

   Information on that topic appears in a window.

3. Choose the Beep Statement.

   Language reference material on the purpose, use, and syntax of this statement appear on screen.

∑ To use the Search command

Like standard Windows Help, the Visual Basic Help system has a Search command on the Help screen. To use Search, simply type in the term you are looking for.

1. From the Visual Basic Help window, choose Search.
2. Type **Toolbox** in the text box.

   The text box is located at the top of the screen.

3. In the Search dialog box, select Show Topics.
4. Select the,topic Toolbox.
5. Choose Go To.

   Visual Basic opens a window with information on the term "Toolbox."

6. From the Help screen, choose Back.

   Visual Basic returns you to earlier topics.

∑ To use the History command

Use the History command to return to topics that you have covered earlier in the session. Topical lists in this option are arranged in reverse chronological order.

1. From the Visual Basic Help window, choose History.

   The Windows Help History dialog box appears.

2. Double-click the topic of your choice.

   The Help screen for that topic appears.

∑ To cut and paste sample code into your code

The most powerful part of the Visual Basic Help system is the large number of code samples that you can paste into a project and run.

The following procedures allow you to copy and paste sample code for a Click event.

1. Visual Basic should already be running.

   Make sure there is a new form on the screen.

2. From the Help menu, choose Search.
3. Type **Click** in the text box.
4. Click the Show Topics button.

   Click Event should be highlighted.

5. Choose Go To.
6. At the top of the Help form, click the word "Example."

   This brings up sample code in a window.

7. From the Event_Click example window, choose Copy.

   This brings up another window with all the sample code in it.

8. Highlight the code you want.

   In this case, take only the lines starting `Picture1.Move...`

9. Click the Copy button.

   This copies the selection to the Clipboard and hides the copy form.

10. Exit Visual Basic Help.
11. Double-click the picture box tool at the top of the Toolbox.
12. Drag the picture box to the lower-left corner of the screen.
13. Double-click the picture box on the form.

    The Code window will open.

14. From the Edit menu, choose Paste.

    Paste the sample code from the Clipboard to your form.

15. From the Run menu, choose Start.
16. Click the picture box once.

    It will move toward the upper-right corner of the screen.

17. Continue clicking the picture.

    It will disappear into the upper-right corner.

18. From the Run menu, choose End.
19. Minimize Visual Basic.

Summary

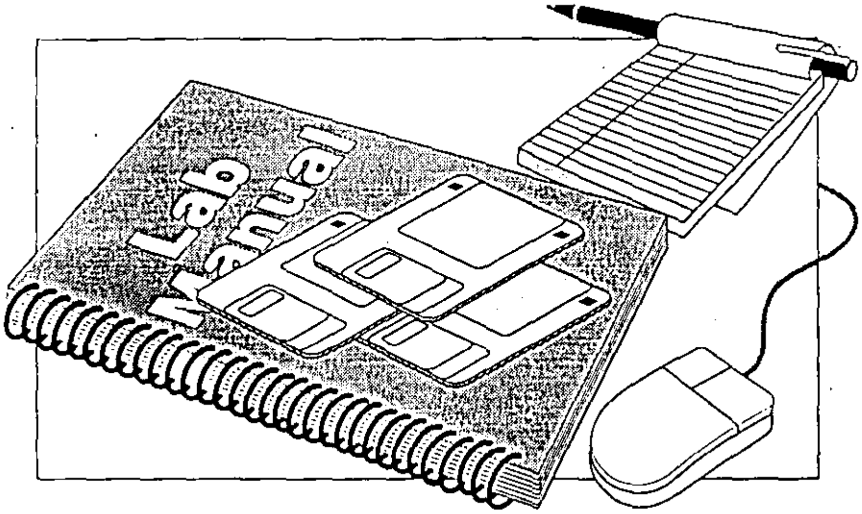
- Visual Basic Tools
- Building a Simple Visual Basic Application
- Visual Basic Menu Commands

Objectives

In this module, you learned to:

- Design an application user interface using the Form window, Toolbox, toolbar, and Property window.
- Name and save the application's forms and project files.
- Start and stop an application from the Visual Basic menu and/or the toolbar.
- Create an executable file and add it to a Windows group.

Lab Time



Go to the Creating An About Box portion of your lab manual.

Module 2: Designing and Building Visual Basic Applications

Σ Overview

- Event-Driven vs. Procedural Programming
- Microsoft Visual Basic Terminology
- Application Development Process
- User Interface Design Guidelines
- Configuring Your Environment

Overview

The purpose of this module is to introduce you to several related concepts that help you make the transition from the procedural world to the event-driven world. This module creates the logical framework for much of the rest of the course. It contrasts programming for Microsoft Windows with MS-DOS and other character-based applications.

It also provides a high-level overview of and establishes relationships between objects and events.

This module also outlines the general process that is used to develop Visual Basic applications and some general suggestions for overall application design.

Prerequisites

None.

Overall Objective

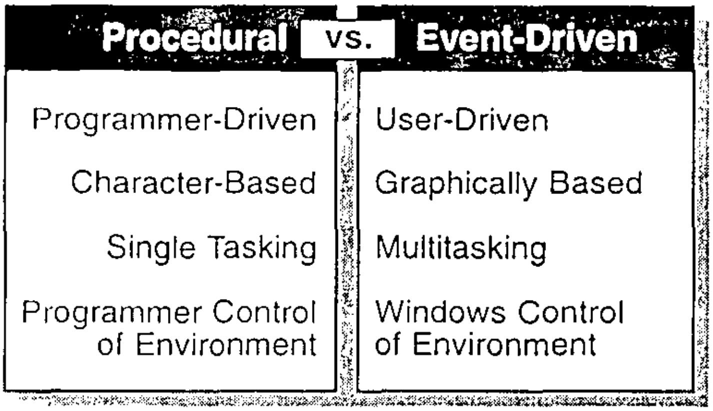
At the end of this module, you will understand the paradigm shift from procedural to event-driven programming.

Learning Objectives

At the end of the module, you will be able to:

- Explain the key differences between graphical and character-based applications
- Provide high-level definitions for some key Visual Basic terms.
- Outline a basic application design and development procedure.
- Explain several fundamental principles of user interface design

Event-Driven vs. Procedural Applications



Procedural	Event-Driven
Programmer-Driven	User-Driven
Character-Based	Graphically Based
Single Tasking	Multitasking
Programmer Control of Environment	Windows Control of Environment

Traditional programming is linear. It has a top-down sequencing that is controlled by the programmer.

Windows-based programming is event-driven. Windows-based events can be triggered in one of two ways. They can be either user-triggered or system-triggered.

Below, in pseudocode, is a very general summary of the structure of an event-driven program for the Windows operating system:

Example

```
Begin MAIN PROGRAM
    Begin Loop
        ' Ask Windows to pass messages to your program
        ' about what events have occurred
        Case Statement GETEVENT
            CASE Click
                ' You can choose to insert code to respond to
                ' click events for the appropriate objects.
            CASE Change
                ' You can choose to insert code to respond to
                ' change events for the appropriate objects.
            CASE ...
              :
              :
            CASE Default:
                ' VB will handle other events internally
                ' without giving your program access.
                ' One of the default cases is the "end
                ' application" message that causes an exit
                ' from the loop and causes your application
                ' to terminate.
        End Case Statement
    End Loop
End MAIN PROGRAM
```

References

For a complete list of the events that Visual Basic recognizes, see the Introduction in the *Microsoft Visual Basic Language Reference.*

Notice that there is not a linear flow to the program because the flow depends on what events are generated and the order in which they are reported to your program by the Windows operating system.

Applications that run in the MS-DOS environment are programmed to be the only application running. MS-DOS–based applications do not handle multitasking very well.

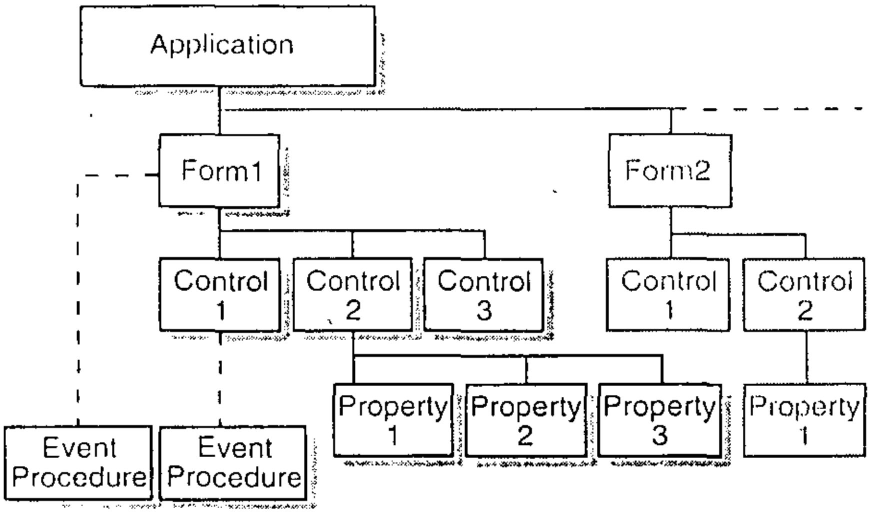
Window applications are multitasking. They can share screen space and computing time.

MS-DOS–based applications are character-based. Windows-based applications are graphically based and typically use proportionally spaced fonts.

Applications programmed to run in the MS-DOS environment are able to control the environment the user operates within. The program can have control over the sequencing and appearance of where the user will go next in the application.

Applications programmed to run under Windows give control of the environment to Windows. How the user moves between events is controlled through the Windows operating system itself rather than through the Windows-based application.

Visual Basic Terminology



The terms on this foil depict two major components of the Visual Basic development environment: the graphical side of it and the code side. Put in tabular form, they look like this:

Application
  Forms
    Properties
    Events
  Controls
    Properties
    Events

What Is an Object?

Consider this example: The dashboard of your car offers users a variety of gauges, dials, and accessories that take input and give output. The thermostat tells you how hot the engine is. The steering wheel lets you change direction. Each one of these "objects" performs a specific purpose, and you use it to attain a given goal or objective. Users learn how to use these objects.

For the most part, users do not learn how to install, maintain, or troubleshoot these things. They do not learn how these things function. The user only needs to know that these objects work and what to expect from them. In a like manner, Visual Basic offers programmers objects—forms and controls—that they can use to build applications. For the most part, the programmer need only use the object without necessarily having a detailed understanding of how the object does what it does.

The Graphical Side

Objects

Components of an application, usually forms and controls.

Forms

Forms are the building blocks of applications. They are the windows that users see when they run your application; they are the major structural units that make up your application. Visual Basic is made up of several forms. The Toolbox, the Properties window, the Project window, and the Save Project As dialog box are all examples of forms.

Walk Through—Forms and What You Get Free

Σ **To open a blank form**

1. Start Visual Basic.

   A blank form should be on the screen.

2. From the Run menu, choose Start.

   Notice all of the things that Visual Basic gives your form, such as:

   - Sizing border
   - Control menu
   - Minimize button
   - Maximize button

   You don't need to write any of the code for painting any of these features of your application. You also don't need to write any of the code for managing these features.

3. From the Run menu, choose End.

Controls

Controls are the tools you place on forms that provide users with application functionality. Examples of controls are the command buttons and labels.

Properties

Forms and controls have properties (attributes) that you can change during design and run time. An example of a property is the caption that appears on a command button. Generally speaking, you set the values for control and form properties when you are designing your application.

Event Procedures

Event procedures are code internal to Visual Basic and Windows that is written for you and provides your application with some of its basic functionality. For example when a user clicks a check box, the Click event knows how to paint an X in the square. However, you must place code inside the Click event for that check box to cause your application to react appropriately when a user places an X in or removes an X from the check box.

The Code Side

Project

Your development project is made up of more than graphical forms and controls; it also has Basic code in it. This is managed and accessed through the Project window.

Walk Through—Visual Basic View of Your Project

∑ To view your project

1. If it isn't already started, Start Visual Basic.
2. From the File menu, choose Open Project.
3. Load the ICONWRKS.MAK file.

   This file is located in \VB\SAMPLES\ICONWRKS.
4. From the Run menu, choose start to run the application.

   Work with the application for a minute, but the point here is that this application is made up of a number of different forms.
5. From the Run menu, choose End to stop the application.
6. Access the Project window.

   The Project window keeps track of the forms and modules that make up your application.

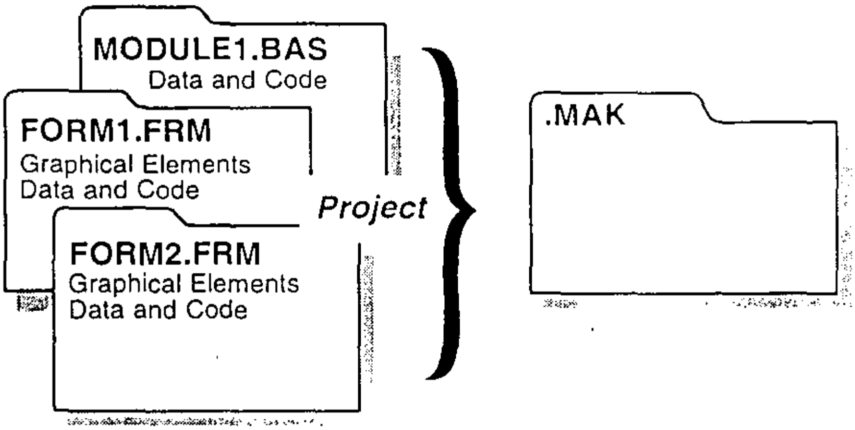
   For our purposes, Visual Basic treats each of your forms as a separate component.

   Visual Basic keeps track of the number and kinds of files that you are using in your application in the .MAK file and loads them when you want to start working on your project.
7. Quit Visual Basic.

Module

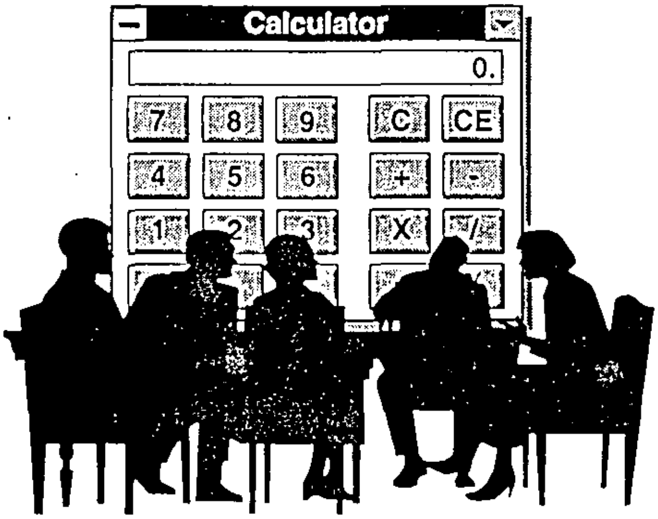
In Visual Basic a module is a file that contains only code. One example of a .BAS file that you could see at the top of a Project window is MODULE1.BAS.

Another way of showing the relationship among forms, controls, properties, and event procedures is this:



The .MAK file contains names of both the .FRM and the .BAS files in the project.

Application Development



Creating an Application

∑ **To create an application in Visual Basic, follow this suggested sequence**

1. Open a new project (or use the new project created when you start Visual Basic) to organize the parts of your application.
2. Create a form for each window in your application.
3. Draw the controls for each form.
4. Create the menu bar for the main form.
5. Set the form and control properties.
6. Write event procedures and general procedures.
7. Save your work.
8. Debug your code.
9. Create an executable file to turn the project into an application.

Distributing the Application

When you distribute the executable of your application to users, you need to distribute a copy of the Visual Basic dynamic-link library VBRUN300.DLL with it. This dynamic-link library (DLL) is a part of the Visual Basic installation files, and you can distribute it royalty free. If you build your application using any of the available custom controls, you will also need to ship all appropriate .VBX files, and in some cases .DLL files. The product documentation for the custom controls will tell you what these files are.

Finally, during design time, you may decide to move the forms for the application to another machine. In this case, you will probably need to rebuild the .MAK file, because it keeps track of all the files that make up your application as well as the fully qualified path to them. To do this, copy all the files to the new machine, then update the paths in the .MAK file by using the Remove File command from the File menu, and then the Add File command.

User Interface Design

General Guidelines

- Design Basics
- Color
- Fonts

User interface design guidelines are an agreement to create consistent user interfaces.

User interface guidelines are important for ease of learning by the user. They also prevent programmers and developers from "reinventing the wheel."

They are guidelines—suggestions on the way you might want to design the user interface. There are no hard and fast rules.

General Guidelines

- Design basics
  - Design for the user, not the system
    Composition and functionality
  - User control
  - Directness
  - Consistency
  - Clarity
  - Aesthetics
  - Feedback
  - Forgiveness
- Color
  - Color as an attention-getter
  - Complementary versus circus colors
- Fonts
  - Serif versus sans-serif
  - Size
  - Number (variety)

For More Information

For more details on user interface design guidelines, see:

- *The Windows Interface: An Application Design Guide*
- *Visual Design Guide* contained within Visual Basic

Walk Through—Designing the User Interface

As you walk through this application, look at the property listed and answer the question that follows.

∑ Print Utility #1

1. From the Walk Throughs program group, start Print Utility #1.

   First impressions are important. Are the function and purpose of the application apparent from a first look at the interface?

   ____

2. Title Bar

   The title bar should contain the name of the application. Does it?

   ____

3. Menus

   The menu structure should reveal something about the contents of the application. Does it?

   ____

   Does the first menu item follow user interface guidelines?

   ____

   Is the Help command in the standard place?

   ____

   Is the Quit command handled suitably?

   ____

   Where does the About command normally appear?

   ____

4. Scroll bar

   Is this the most effective way to ask users for the number of copies they will be wanting?

   Are the size and location of the scroll bar appropriate given the overall purpose of the application?

5. Options

   How many sets of options are there really on this form?

   Are the check boxes presented in red, the best way to get information from the user?

   How should the header, footer, and page numbering questions be implemented?

6. FileName text box

   If the user wants to find a file and print it, what other tools does the user need besides this simple text box?

7. Stop and Print buttons

   Are these buttons appropriately sized for this form and its function?

8. Exit

   Close the application.

∑ Print Utility #2

1. From the Walk Throughs program group, start Print Utility #2.
2. Open the second version of this application.

   Is the overall layout of the form effective? How does the layout suggest how the user might work with the form?

   ____

   How do the frames help structure the user's decisions?

   ____

   How is the menu structure consistent with user interface guidelines?

   ____

   Why were the header and footer options dropped from the form?

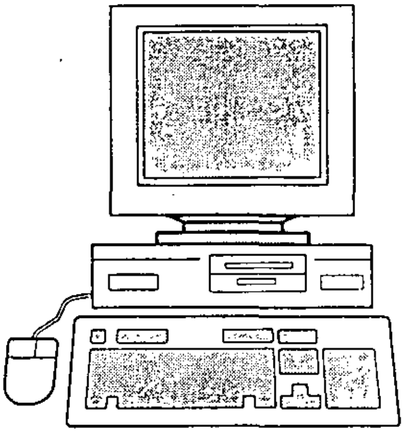
   ____

   Why was the Percent Done label added?

   ____

3. Quit the application.

User Interface Design Guidelines



Device Input

Getting Input from a Mouse

There are a limited number of things that users can do with a mouse:

1. Single-click with either the primary or secondary (left or right, top or bottom button.
2. Double-click with either the primary or secondary button.
3. Drag normally with the primary button.
4. Drag and drop.

For example, a single click with the primary mouse button normally indicates that a choice such as a bold, italic, or underline text format option has been made on the ribbon of a text editor. Selection of a file to be opened is normally indicated by a single click.

Double-clicking selects the object to be acted upon and initiates the action. That is, if you want to select and open a file, you simply double-click a filename. The same thing happens when you double-click an icon.

Dragging is used to resize a window, move the window to a new location, or reposition an icon. Drag and drop is normally used to select an object, for example, the name of a file, and then place it over another object, for example, a printer, so that the file can be printed.

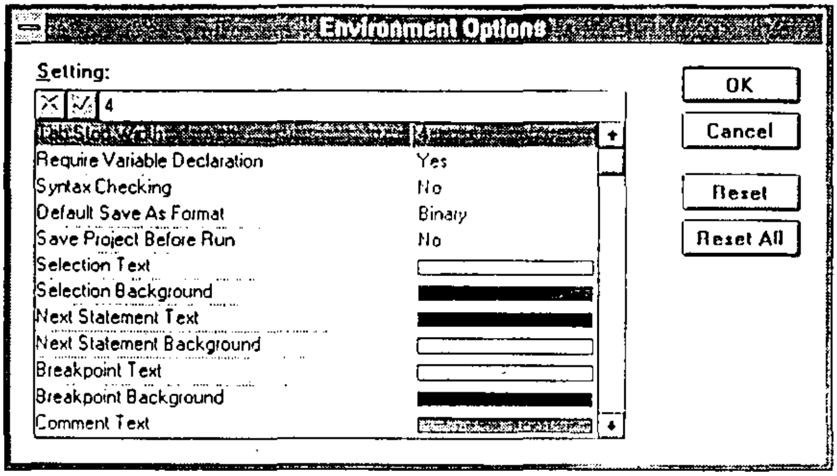
Keyboard and Mouse Functions

Whenever possible, all mouse actions should be duplicated with an equivalent keyboard action. Keyboard shortcuts and access characters should also be used to reduce the number of keystrokes and potential for errors by the user.

For example, in most cases you will want to give a keyboard shortcut for the most commonly used commands. If possible, use the first letter of a menu or control as the access character. If there is a conflict, use another letter in the menu or control name.

When you design the application and its forms, specify the shortcut keys.

Configuring Your Environment



Setting: 4

Tab Stop Width 4
Require Variable Declaration Yes
Syntax Checking No
Default Save As Format Binary
Save Project Before Run No
Selection Text □
Selection Background ■
Next Statement Text ■
Next Statement Background □
Breakpoint Text □
Breakpoint Background ■
Comment Text ▨

You can configure much of the look and feel of the development environment. From the Options menu, choose Environment Options. From this window you can set everything from the default tab stop in code to the foreground and background colors of comments in code.

Setting	Default
Tab Stop Width	4
Require Variable Declaration	No
Syntax Checking	No
Default Save As Format	Binary
Save Project Before Run	No
Selection Text	
Selection Background	
Next Statement Text	
Next Statement Background	
Breakpoint Text	
Breakpoint Background	
Comment Text	
Comment Background	
Keyword Text	
Keyword Background	
Identifier Text	

Setting	Default
Identifier Background	
Code Window Text	
Code Window Background	
Debug Window Text	
Debug Window Background	
Grid Width	120 Twips
Grid Height	120 Height
Show Grid	Yes
Align to Grid	Yes

For a list of the colors available, see the Colors section of CONSTANT.TXT located in the \VB subdirectory.

Note Changing to a larger grid size and then aligning the controls to the grid may change the size of your controls.

Automatically Loading Visual Basic Extensions at Startup

You can control which of the .VBX or Visual Basic Extensions are loaded when starting up a new project by editing AUTOLOAD.MAK with a text editor. This file is located in the \VB subdirectory. The default settings are:

```
GRID.VBX
MSOLE2.VBX
ANIBUTON.VBX
CMDIALOG.VBX
CRYSTAL.VBX
GAUGE.VBX
GRAPH.VBX
KEYSTAT.VBX
MSCOMM.VBX
MSMASKED.VBX
MSOUTLIN.VBX
PICCLIP.VBX
SPIN.VBX
THREED.VBX
ProjWinSize=152,402,248,215
ProjWinShow=9
```

Summary

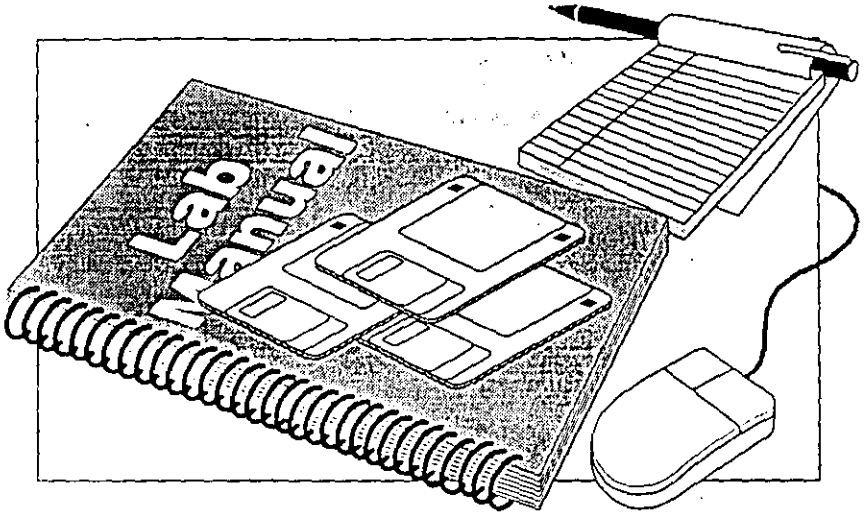
- Event-Driven vs. Procedural Programming
- Microsoft Visual Basic Terminology
- Application Development Process
- User Interface Design Guidelines
- Configuring Your Environment

Objectives

In this module you learned to:

- Explain the key differences between graphical and character-based applications.
- Provide high-level definitions for some key Visual Basic terms.
- Outline a basic applications design and development procedure.
- Explain several fundamental principles of user interface design.

Lab Time



Go to the Employee Database Application Specification portion of your lab manual.

Module 3: Working with Forms

∑ Overview

- Forms and Their Properties
- Message Boxes
- Starting the Forms of the Employee Database
- Multiple Document Interface Applications

Overview

The purpose of this module is to introduce you to the Visual Basic programmer's fundamental tool — the form.

Even though this is a relatively brief module, it introduces some key terms that are required for a full understanding of the programming environment. This module primarily focuses on the various properties associated with forms, separate from the discussion of controls and properties. This module also touches on event procedures and methods associated with forms. A more detailed discussion of these issues will be found in a separate module.

This module also serves as the introduction to the main elements of the class application and walks you through creating, naming, and saving these forms.

Prerequisites

Prior to starting this module, you should have a fundamental awareness of:

- Windowing technology from the user perspective
- The Visual Basic programming environment

Overall Objective

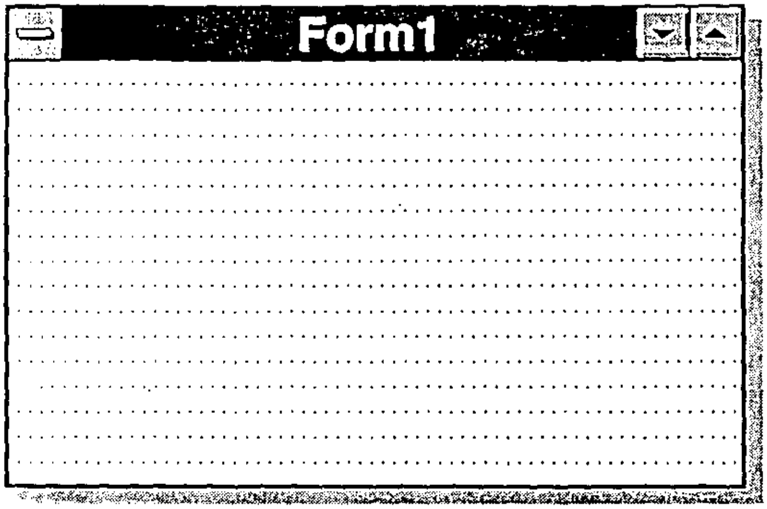
At the end of this module, you will understand the fundamentals of working with forms in Visual Basic.

Learning Objectives

At the end of this module, you will have:

- Set captions on forms.
- Set the Name property for a form.
- Added a form to a project.
- Saved all of the forms in a project.
- Saved the project itself.

Properties for Forms



Forms and Their Common Properties

Forms are the central element of Visual Basic. They are the design of your application; they are the thing that your user interacts with. They are where you place the controls for your application.

You will notice that many of the properties for forms have the same names as those found for controls, but on forms they have a different use.

Property	Default	Comments
BorderStyle	2 - Sizable	Set BorderStyle to "3- Fixed Double" to create a modal form (used for dialog boxes).
Caption	Form1	Appears at the top of the form.
ControlBox	True	Enables the Control menu.
FontSize	8.25	Used in printing to a form.
FontName	Helv	Used in printing to a form.
Name	Form1	The name in code. Also appears on Project list.
Height	4425 twips	
Icon		Default icon for your executable.
KeyPreview	False	Determines whether the form keyboard events occur before control keyboard events.
Left	–	From left edge of screen.
MaxButton	True	Disable for a modal dialog box.

Property	Default	Comments
MinButton	**True**	Disable for a modal dialog box. Also see BorderStyle.
MousePointer	0 Default	Sets cursor shape on form – 13 possible choices. See the *Microsoft Visual Basic Language Reference.*
Top	–	From top edge of screen.
Visible	**True**	False is equivalent to calling the **Hide** method.
Width	7485 twips	

Events

Load A Load event occurs when a form is loaded. Normally, you will use a Form Load event to set initial values for the controls on a form.

Example

```
Sub Form_Load ()
   Top = 1500
   Left = 1000
End Sub
```

Unload An Unload event occurs when a form is about to be removed from memory. This event is normally triggered by the user closing the form using the Control menu.

Methods

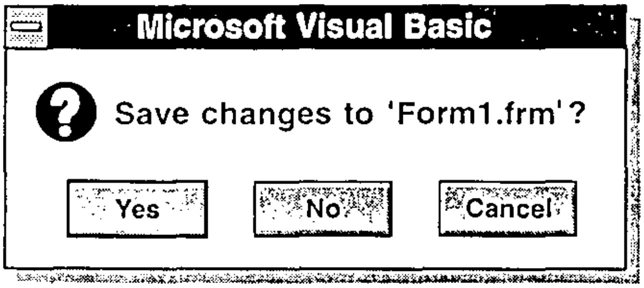
Hide This method is used to remove a form from the screen without unloading it. When you use the **Hide** method, the Visible property of the form is set to **False**.

Show The **Show** method is used to display a hidden form.

Example

```
Sub mnuPrintOptions_Click ()
   frmPrintOptions.Show
End Sub
```

Message Boxes



Microsoft Visual Basic

Save changes to 'Form1.frm'?

What Are Message Boxes?

For the most part, Visual Basic applications are made up of forms. You will find several places in your applications where you want to convey fairly routine kinds of information to users and you really don't want to design and build a form to deliver it.

Visual Basic has a convenient tool that displays a kind of form—called a message box—where all you need to do is to provide the message string and, optionally, a number that indicates the number and types of buttons and icons to be displayed on the form, and a title for the dialog box.

If you wanted to make sure that users had saved all of their new data prior to closing an application that you are writing, you could create a whole new form, or you could use the **MsgBox** function.

Walk Through—Coding Message Boxes

∑ **To inspect the message box sample application**

1. From the Walk Throughs program group, start MsgBox.
2. Click the Exit button on Form1.

   This displays a dialog box that queries users about the status of their data.
3. Choose Yes, No, or Cancel.

   Any one of these choices closes the dialog box. Implementing code that differentiates them will be added in another module.
4. Open the Control menu on Form1, and choose Close.

   This closes Form1.

∑ To code the message box sample application

1. If Visual Basic is not running, start it.
2. Double-click the command button tool in the Toolbox.
3. Place it in the lower-right corner of the form, or any other out-of-the-way place.
4. In the Properties window Properties list for the command button, change the following property:

   Caption: Exit

5. Click anywhere on the form.

   This puts the focus back on the form.

6. Double-click the command button.

   This will automatically open the Code window for the form and take you to the Command1_Click event template.

7. Add the following code:

   ```
   Msg$ = "Have you saved all your work?"
   MsgBox Msg$, 3 + 32, "MsgBox Walk Thru"
   ```

   What does the code do? First you create a string variable with the message you want to display, and then you call the **MsgBox** statement. You can pass the **MsgBox** statement three arguments. Although the second and third arguments are optional, in this case we have you include them.

   The first argument is the message string that is displayed inside the message box.

   The second is a sum of values that indicates to Visual Basic the type of message box you want. More about these numbers in a minute.

   The third argument is the title of the message box.

8. From the Run menu, choose Start.
9. Test the dialog box.

   Use the mouse pointer to choose the Exit command button. The dialog box will appear in the middle of your display.

10. To close the Message WalkThru dialog box, choose the Cancel button.
11. From the Run menu, choose End.
12. In the Code window, highlight MsgBox.

    Use the mouse pointer to highlight this word in the code.

13. Press F1.

    Make sure you are in the design mode, not the run mode.

    By adding the appropriate key values listed in the second table in the Visual Help topic for the **MsgBox** statement, you can control the number and type of buttons, control the icon displayed in the message box, and even set a default button.

    In this walk through, you add the value 3 (to request a message box with Yes, No, and Cancel buttons) and the number 32 (to request a message box with a Warning Query or question mark icon).

    There are more sophisticated ways of implementing and using message boxes, but this gives you a start.

    Did you notice that the Yes, No, and Cancel command buttons do much, yet?

    If you read further down in the Help window, you see that each of the different buttons returns a different value. You will use those values a little later in the course, along with some conditional logic to code the three buttons on the dialog box. For the time being...

14. Minimize Help.

15. From the File menu, choose Save File As.

    Save this form in \WALKTHRU\STUDENT1 as MSGBOX.FRM.

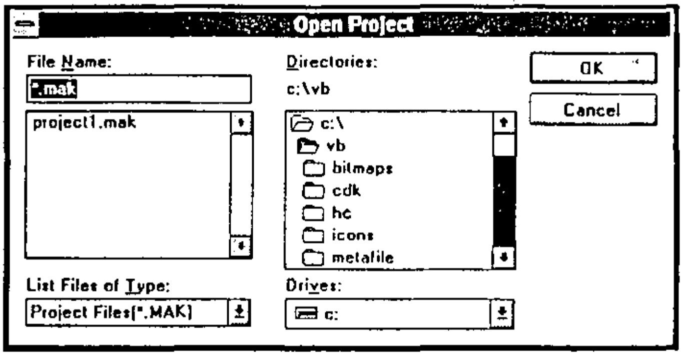
16. From the File menu, choose Save Project As.

    Save the project in the same subdirectory as MSGBOX.MAK.

17. Select the Visual Basic Control menu, and choose Close.

    Quit Visual Basic and Help.

Modal and Modeless Dialog Boxes


Open Project

File Name: *.mak

project1.mak

Directories: c:\vb

c:\
vb
bitmaps
cdk
hc
icons
metafile

List Files of Type: Project Files(*.MAK)

Drives: c:

Modal, Modeless, and System Modal

You can designate a form as being either modeless or modal. The default is modeless, which means that the user can open the form and still get to other forms within the application to do work. Modal forms, in contrast, require that the user do something—click a button, check to make sure they really do want to delete all those files—before they can do any work.

Walk Through—Modal Dialog Boxes

∑ **To use a modal dialog box**

1. From the File menu, choose Open Project.

   Bring up the Open Project dialog box.

2. Access the Project window.

   Place the cursor on any other portion of the Visual Basic interface. Press either one of the mouse buttons.

   A beep sounds and the cursor continues to flash in the text box. In this case the user must choose one of the three choices presented: Locate the name of a project using the list boxes, input a valid project name, and choose OK or Cancel.

3. Click the Cancel button.

   Close the dialog box.

Application Modals and System Modals

Modal dialog boxes do have a limit; they only guide the user within their own application. They have no effect if the user switches to Windows or to another application. In order to make a form system modal, all you need to do is to use the *style%* parameter for both MsgBox statements and functions.

The syntax looks like this:

[form.] Show [*style%*]

In addition, you would need to declare three global constants:

```
Global Const MODAL = 1
Global Const MODELESS = 2
Global Const SYSTEMMODAL = 4096
```

Walk Through—Making the Forms of the Employee Database Application

∑ **To make the forms of the Employee Database application**

1. If Visual Basic isn't already running, start it.
2. Set the following properties for the form:

Caption	Employee Database
Name	frmEmpDB
Height	5760 (approximately)
Icon	\VB\ICONS\MISC\MISC28.ICO
Width	7600 (approximately)

Note When you set the Icon property for a form, you will need to know the full path to the source file, but the Visual Basic interface will only tell you that you have an icon by displaying the property as (Icon).

3. From the File menu, choose Save File As.

   Make sure that you save the file in the appropriate subdirectory. If you don't, the .MAK file will not be able to find this part of your project. This first pass at the application should be saved in \STUDENT1\FORMS.

4. Name the file EMPDB.FRM.
5. Choose OK.

   This saves the form.

6. Click the Project window.

   Notice that the filename is in the left column on this list, and the name of the form is in the right column.

   So far, you have given the form the names that are known to the file system and to Visual Basic.

7. From the Control menu for the Employee Database form, choose Close.

   Close EMPDB.FRM.

8. From the File menu, choose New Form.

   Add a second form to the application.

9. Set the following properties for the form:

Caption	Employee Record
Name	frmEmpRec
Height	4545 (approximately)
MaxButton	**False**
Width	5485 (approximately)

10. From the File menu, choose Save File As.

    Now, rename and save the file, making sure that you save it in \STUDENT1\FORMS.

11. Name the file EMPREC.FRM.
12. Choose OK.
13. From the Control menu on your newly created form, choose Close.

    Close the form, and notice that the name of this form has been added to the list in the Project window.

    User interface guidelines suggest that applications have an About box — a form that indicates that the application is copyrighted. You have already created an About box; now you need to add it to this application.

    Note You will also want to make some changes to the About form, but the directions for doing this are not included here. You would, for example, change the caption for the About box so that it contained the name of the application, but doing that here distracts from the point of this walk through.

14. Copy ABOUT.FRM from \STUDENT1\ABOUT to \STUDENT1\FORMS.

    Before you add the About box form to the project list, you should make a copy of the About box form that you have already completed and place that in \STUDENT1\FORMS.

15. From the File menu, choose Add File.

    ABOUT.FRM should appear in the \FORMS subdirectory, so all you need to do is select the name so that it appears in the text box.

16. Choose OK.

    The name of the added form should appear in the Project list.

17. From the File menu, choose Save Project As.

    Now that you have made most of the basic forms for the Employee Database application, you need to rename and save the file that keeps track of the files in the project.

    The Save Project As dialog box should appear center screen, with the correct subdirectory already listed in the current working directory and a suggested filename with the appropriate file extension.

18. Name the project as EMPLOYEE.MAK.
19. Choose OK.

    This saves all of your current build information.

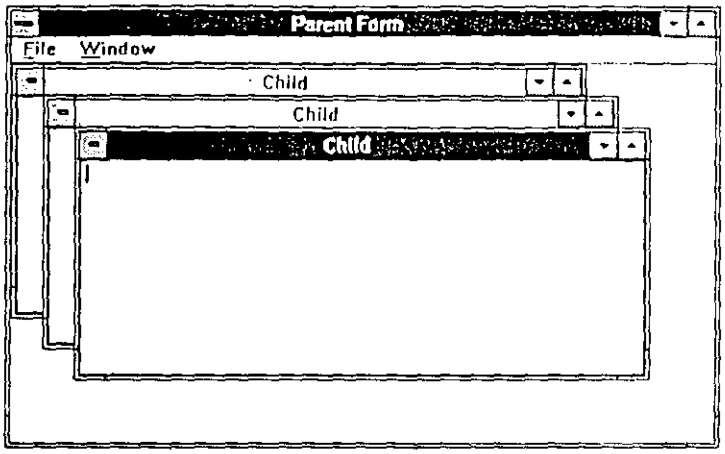
20. From the Run menu, choose Start.

    If you want to test the application, you should see the Employee Database form appear on screen. This form appears only because that was the first form you created. If you try to open the Employee Record form, you will see that you have not implemented code for that task yet.

21. From the Run menu, choose End.

22. Quit Visual Basic.

Multiple Document Interface Applications



Placing Forms Within Forms

Visual Basic allows you to write applications that can create multiple copies of a form and display all the forms within a single container form. Microsoft Word for Windows and Microsoft Excel are both applications that allow users to do this.

Walk Through—Creating MDI Applications

∑ **To see the final version of your MDI application**

1. From the Walk Throughs program group, start MDI.

   For the most part, in this course, we will focus attention on placing controls on forms. However, Visual Basic has the capacity to place forms on forms and multiple instances of forms on forms.

   The purpose of this walk through is to give you the fundamentals of implementing an MDI application. We don't include all the enhancements that you might want to implement. Detailed coverage can be found in the follow-on *Programming in Microsoft Visual Basic 3.0* course.

2. From the File menu, select New.

   Do this three times. You may not see much happening at first, but each time you select New, another instance of the child window is being drawn on screen, one atop the other.

3. From the Window menu, choose Tile.

   This organizes and displays all of the child windows that you have created.

4. From the Control menu on the Parent form, choose Close.

   In order to create this application framework, follow the steps below; but there is a warning required here. You will be typing in a number of things that haven't been explained yet. Don't worry; they will be. For the time being, accept that things work and that the details will follow.

∑ To create a simple MDI application

1. Start Visual Basic.

   A new, blank form will appear on screen.

2. From the File menu, choose Add File.
3. Use the browser to add PARENT1.FRM located in \WALKTHRU\FORMS.

   Note Under normal circumstances, you wouldn't go this route. You'd select the New MDI Form command. If you did that, a second form would appear on screen, the entry named MDIFORM1.FRM would be added to your project list, and the New MDI Form command on your File menu would be disabled. From there, you would need to add all the menu items; but since you haven't done that yet, we will give you the completed form, so that you can concentrate on the MDI capabilities.

   Now you should have two forms in your project.

4. In the project list, put the focus on FORM1.FRM.

   That is, make sure that you see FORM1.FRM on top of all the other forms within Visual Basic.

5. From the File menu, choose Save File.

   Save the file as FORM1.FRM.

6. From the Window menu, select Properties.

   This displays the Properties window for FORM1.FRM.

7. In the Properties window Properties list, locate the MDIChild property.

   The default value for this property is **False**.

8. Double-click the value in the table.

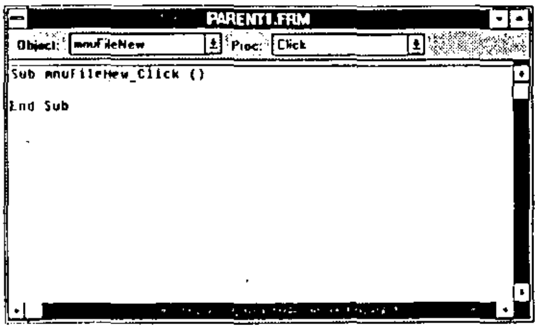
   This toggles the value to **True**.

9. Select the Control menu for the Properties window and choose Close.
10. In the Project window, highlight PARENT1.FRM and choose View Form.

    This places the focus on this form.

11. Open the File menu on the Parent Form and choose the New command.

    This is the shortcut for getting from the form to the code that supports it. You should now be in a Code window that looks like this.



```
Sub mnuFileNew_Click ()

End Sub
```

12. Place your cursor on the line between the Sub and End Sub lines and add this code:

    ```
    Dim NewDoc As New Form1
    NewDoc.Show
    ```

    That's all there is to it. Your questions about Dim and New and Show will all be answered in a little while.

13. From the Object drop-down combo box, choose General.

    This moves you to the General Declarations section of the code, where you can declare a couple of constants that Visual Basic needs for arranging the windows that your application creates.

14. Add the following code:

    ```
    Const CASCADE = 0
    Const TILE_HORIZONTAL = 1
    ```

15. From the Object drop-down combo, select mnuWindowCascade.
16. Add the following code on the blank line between the Sub and End Sub lines:

    ```
    MDIForm1.Arrange CASCADE
    ```

17. From the Object drop-down combo, select the mnuWindowTile.
18. Add the following code on the blank line:

    ```
    MDIForm1.Arrange TILE_HORIZONTAL
    ```

    That should do it. Now all you need to do is to run your application.

Σ To run your new application

1. Choose the Run icon on the toolbar.
2. From the File menu on the Parent form, choose New.
3. Do this a couple of times so that you have several windows to work with.
4. From the Window menu on your application, choose Tile.

   This tiles all of your windows.
5. From the Window menu on your application, choose Cascade.

   This re-arranges all of the windows.
6. From the Run menu, choose End.
7. From the Visual Basic File menu, select Save Project As.

   Now that you have the framework for an MDI application finished, save your work.
8. Save the project as MDI.MAK and make sure that it is in \WALKTHRU\SAMPLES.

Summary

- Forms and Their Properties
- Message Boxes
- Starting the Forms of the Employee Database
- Multiple Document Interface Applications

Objectives

In this module you learned to:

- Set captions on forms.
- Set the Name property for a form.
- Add a form to a project.
- Save all of the forms in a project.
- Save the project itself.

Module 4: Laying Out Menus

∑ Overview

- Menu Guidelines
- Microsoft Visual Basic Implementation

Overview

Previous modules gave you a chance to see and use the processes you should follow for developing Visual Basic applications. In those modules you developed a single-form application. In another module, you developed all the forms that are needed to implement the database front-end, but there was a part that was missing—the menus on some of the forms. This module begins with a discussion of the general layout of menus and the user interface specification. It ends with a demonstration/walk through of the Menu Design window.

Prerequisites

To successfully complete this module, you must be able to use a mouse. Experience with Windows-based applications is useful but not required. You should also be able to start Visual Basic and create a form.

Overall Objectives

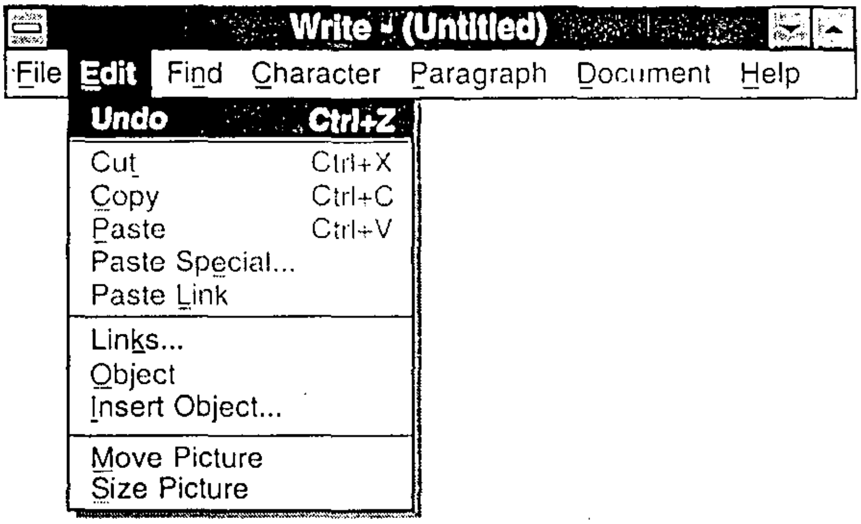
The overall objective of this module is to teach you how to design user-friendly menus and create them using Visual Basic.

Learning Objectives

At the end of this module's lab, you will be able to:

- List and explain the use and value of all of the key menu elements:
  - Access or hot keys
  - Shortcut keys
  - Menu bar
  - Separator bar
  - Ellipsis
- Create an application that uses at least one menu on an application window and uses more than one form. It should include a least one functionality menu as well as a Help menu that includes choices to display a simple Help form and an About box.

Menu Guidelines



Menus and Their Standard Properties

Under user interface guidelines, menus are one of the primary ways for programmers to structure application functionalities for users. Normally, menus do one of two things: explicitly invoke a command, such as closing an application using Exit, or invoke a dialog box that offers users more options, such as the bold and italic text-formatting options.

Menu Structure

Under normal circumstances, most applications will start out life on the development side with three main menus—File, Edit, and Help. The File menu normally will contain commands that are related to file manipulation:

- New
- Open
- Save
- Save As
- Print
- Print Setup
- Repaginate
- Exit

Style Guidelines

Quitting the application, by convention, is done from the bottom of the first menu. Help is normally the last menu on the right.

The general Edit menu normally contains at least these four commands:

- Undo
- Cut
- Copy
- Paste

It may contain several other commands as they are needed for specific functions within the application.

Finally, Help is normally implemented with at least these four choices:

- Contents—Table of contents for Help
- Search for Help on...—Index to Help
- How to Use Help—Directions for using Help
- About *application name*—Copyright notice

Special Features of Menus

Period Ellipses

The three dots at the end of a command on a menu indicate that a dialog box will appear offering you more choices related to the command. A lack of three dots indicates that as a result of choosing the command, an action will be carried out. For example, if you choose Centered from the Paragraph menu, all highlighted paragraphs will be centered on the page.

Checking Options

A check mark next to a menu item indicates to the user that the option is currently invoked.

Separating Clusters of Related Commands

Separator bars are used to visually cluster related commands so that the user sees a short list of related items, rather than a long list of seemingly unrelated items.

Access or Hot Keys

Access keys are marked by an underscore beneath a single letter in the menu item or command. Access keys are the keyboard (as contrasted with the mouse) input device for the menu. Access keys are invoked by pressing the ALT key and then the underscored letter in the command. For example, pressing ALT+F opens the File menu. Pressing ALT+F+O opens the File menu and then chooses the Open command.

Style Guidelines

For access keys, you should try to select the letter that will be most memorable for most users. Normally this is the first letter in the word, but that might not always work. For example, Minimize and Maximize both begin with "M," so a better set of options might be to use the "n" from Minimize and the "x" from Maximize for the access keys.

Shortcut Keys

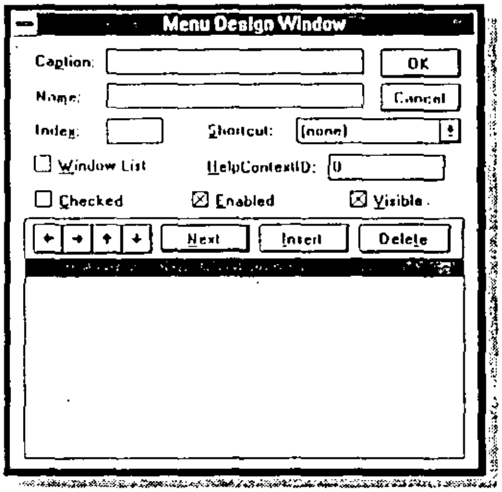
Shortcut keys are a second form of keyboard access to the menu. To use shortcut keys, the user need only press a function key or some other key combination (such as CTRL+A) in order to execute a command.

Style Guidelines

Some examples of standard shortcut keys are as follows:

F1	Help
CTRL+X	Cutting selected text
CTRL+C	Copying selected text
CTRL+V	Pasting selected text
CTRL+Z	Undo

Visual Basic Implementation



Caption:

Name:

Index:

Shortcut: (none)

☐ Window List

HelpContextID: 0

☐ Checked ☒ Enabled ☒ Visible

The Menu Design Window

From the Window menu, choose Menu Design window. If the Menu Design window command is unavailable (dimmed), click any of the forms in your Visual Basic application. You will then be able to access the Menu Design window. The Menu Design window dialog box appears.

The remainder of this topic divides the Menu Design window into two units—Menu Layout Items and Layout Manipulation Items.

Menu Layout Items

Item	Default	Description
Caption		Name that appears on the menu.
Name		Name used in code. Use mnu as a prefix.
Index		Used for adding menu items dynamically.
Shortcut		CTRL + shortcut keys.
Window List		Specifies whether a menu control will include a list of open MDI child forms.
HelpContextID		Specifies an identifier for the menu item in a Help system.
Checked	Not checked	Indicates whether a check mark will be displayed to the left of a menu choice.
Enabled	Checked	Indicates whether a menu choice is available to the user (disabled choices are dimmed).
Visible	Checked	Indicates whether a menu item is visible to the user.

Layout Manipulation Items

In order to use most of the Layout Manipulation Items on the Menu Design window, you must first select the control (menu) and then press the manipulation item to alter the control

Item	Function
LEFT ARROW	Raises a menu item one level —for example, makes a submenu into a main menu
RIGHT ARROW	Lowers a menu item one level for example, makes a main menu into a submenu.
UP ARROW	Moves the menu item up one position in the menu list.
DOWN ARROW	Moves the menu item down one position in the menu list.
Next	Moves the cursor down one item in the menu list. Also, clears the Caption and Name so that you can add a new menu item.
Insert	Inserts a blank line in the list of menus so that you can add a new menu item.
Delete	Deletes the selected menu item from the list.

Style Guidelines

In order to implement access or hot keys, place an ampersand (&) in front of the letter to be used in the ALT+ combination. For example, to implement ALT+F as the access key combination for the File menu, you would type **&File** in the Caption control of the Menu Design window.

Separator bars are implemented by entering a single hyphen (-) as a Caption.

Note You must assign a menu name to the separator even though it is not really acting as a control.

Walk Throughs—Startup Application with Menus and Click Events

∑ **To see the final version of the menus Walk Through**

1. From the Walk Throughs program group, start Startup.

   This sample application serves two purposes. It introduces you to the methods for implementing menus in Visual Basic applications, and it gives you a simple example of how Click events work in Visual Basic. Incidentally, this sample application also shows you how to use the **Shell** function.

2. Choose the Cardfile command button.

   This application starts up Cardfile or Windows Paintbrush™.

3. Close Cardfile.
4. From the Edit menu, choose Colors.

   The application shows the implementation of menus, cascading menus and checked menus. It also uses the ampersand for hot keys.

5. From the Colors cascading menu, choose Red.

   Finally it shows the use of a Click event to reset the background color of the application.

   Now, the question is: How did we do all of that?

6. Close the Startup application.

∑ To implement menus on the Startup application

1. If Visual Basic isn't running already, start it.

   A new blank form should be on the screen.

2. From the File menu, choose New Project.

   Start a new project.

3. Set the following properties for the form in the Properties window:

Name	frmStartup
Caption	Startup Applet
Height	2860 (approximately)
Width	2670 (approximately)

4. Open the Menu Design window.

5. Make the following changes.

Menu item	Caption	Name	Indentation
Edit	&Edit	mnuEdit	0
Colors	&Colors	mnuColors	1
Red	&Red	mnuRed	2
White	&White Select the Checked option.	mnuWhite	2
Blue	&Blue	mnuBlue	2
	–	mnuSep1	1
Exit	E&xit	mnuExit	1
Help	&Help	mnuHelp	0
About	&About... Deselect the Enabled option.	mnuAbout	1

Note Notice that the menu names include only the prefix and the menu function. We shortened the names for this exercise to facilitate completing the walk through.

6. Choose OK.

7. Double-click the command button tool in the Toolbox *twice*.

   Add two command buttons and place them appropriately on the form.

8. Set the following properties for the command buttons.

Button	Name	Caption
Cardfile	cmdCardfile	&Cardfile
PaintBrush	cmdPaintBrush	&PaintBrush

9. From the Project Window, choose View Code.

   Make sure that STARTUP.FRM is in the foreground.

10. In the Object drop-down combo box, choose General Declarations.

11. Add the following code:

    ```
    Const RED = &HFF&
    Const WHITE = &HFFFFFF        Make sure you type six Fs.
    Const BLUE = &HFF0000         Make sure you type four zeros.
    ```

    Set the hexadecimal values for the colors that you want. You can find these colors and several others in CONSTANT.TXT.

    Note . There is a second method for completing the step above: Open CONSTANT.TXT, and copy and paste the appropriate lines into the General Declarations section of the form.

12. In the Object drop-down combo box, select cmdCardfile and locate the Click event template.

    When the user clicks this command button, use the **Shell** function to start Cardfile in a regular window. Add the following code:

    ```
    x% = Shell("c:\windows\cardfile.exe", 1)
    ```

13. In the Object drop-down combo box, select cmdPaintBrush and locate the Click event procedure template.

    When the user clicks this command button, use the **Shell** function to start Paintbrush in a regular window. Add the following code:

    ```
    y% = Shell("c:\windows\pbrush.exe", 1)
    ```

14. In the Object drop-down combo box, select mnuRed and locate the Click event procedure template. Add the following code:

    ```
    frmStartup.BackColor = RED
    mnuRed.Checked = True
    mnuWhite.Checked = False
    mnuBlue.Checked = False
    ```

15. In the Object list box, select mnuWhite and locate the Click event procedure template. Add the following code:

    ```
    frmStartup.BackColor = WHITE
    mnuRed.Checked = False
    mnuWhite.Checked = True
    mnuBlue.Checked = False
    ```

16. In the Object drop-down combo box, select mnuBlue and locate the Click event procedure template. Add the following code:

    ```
    frmStartup.BackColor = BLUE
    mnuRed.Checked = False
    mnuWhite.Checked = False
    mnuBlue.Checked = True
    ```

17. From the Run menu, choose Start and test your application.

18. From the Run menu, choose End.

    Close the application. You may notice that the Exit and About commands are not implemented yet. You will complete those functions in the next module

19. Save the form as STARTUP.FRM in \WALKTHRU\SAMPLES.

20. Save the project as STARTUP.MAK in \WALKTHRU\SAMPLES.

Summary

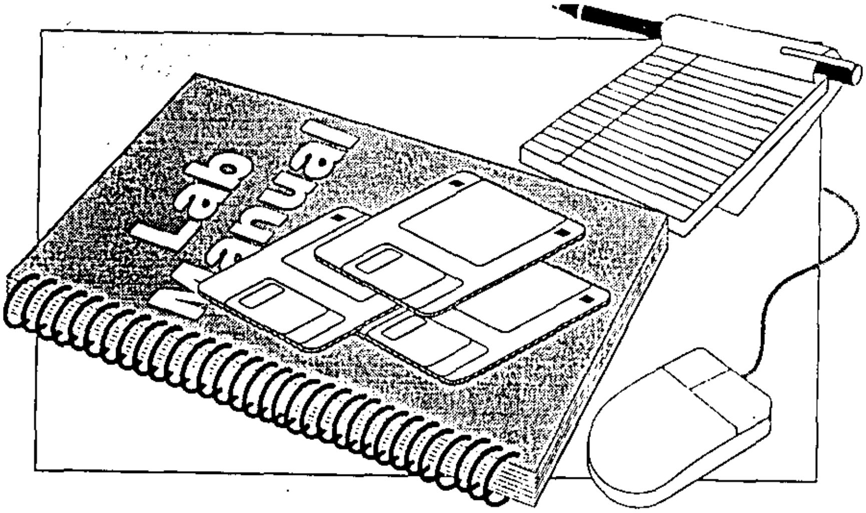
- Menu Guidelines
- Microsoft Visual Basic Implementation

Objectives

In this module you learned to:

- List and explain the use and value of all of the key menu elements:
  - Access or hot keys
  - Shortcut keys
  - Menu bar
  - Separator bar
  - Ellipsis
- Create an application that uses at least one menu on an application window and uses more than one form. It should include a least one functionality menu as well as a Help menu that includes choices to display a simple Help form and an About box.

Lab Time



Go to the Creating Application Menus portion of your lab manual.

Module 5: Connecting Forms

∑ Overview

- **Form Management**

  Statements

  Methods

  Event Procedures
- **Setting the Startup Form**

Overview

This module has two broad goals. First, it provides an introduction to the concepts of event procedures and methods as they relate to managing the forms of your application. Second, it reviews all the important events and methods, and therefore is a crucial lead-in to the controls modules, and the general discussion of functions and statements as they are implemented in Visual Basic.

This module also contains the first actual hands-on experience you will get in writing code that will be implemented by user actions with the application interface.

Prerequisites

Prior to starting this module, you should already be familiar with these terms:

- Controls
- Forms
- Properties

Overall Objective

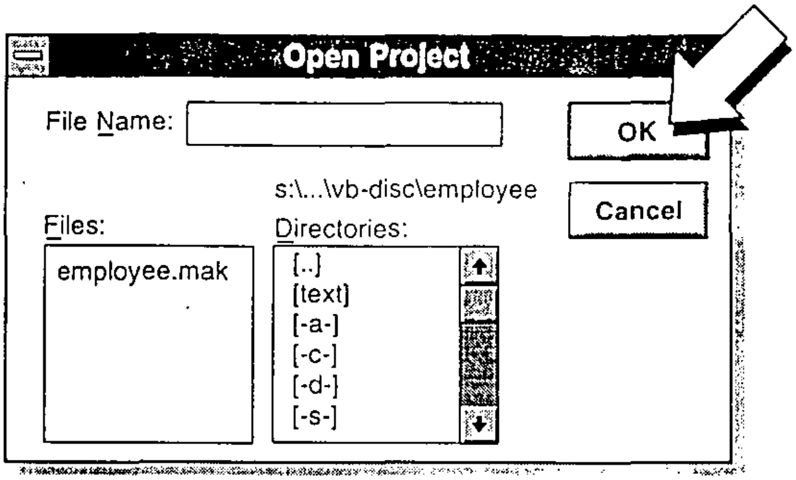
The primary objective of this module is to present the notion of event procedures and to discuss the essential events needed to manage forms.

Learning Objectives

At the end of the module, you will be able to:

- Use the **Unload** statement to unload a form from memory.
- Use the **Load** statement to load a form into memory.
- Use the **Show** method to display a form.
- Use the **Hide** method to hide a form but not unload it from memory.
- Set the startup form.

Statements



The Load Statement

The **Load** statement is used to load a form or control into memory. It normally appears within other event procedures.

Syntax

Load *object*

Example

```
'Event procedure for a control on Form1
Sub Command1_Click ()
   Load Form2
End Sub
```

Important **Load** does not automatically display the form; it just loads it into memory. To make the form visible to the user, call the **Show** method, discussed later in this module. In some cases you may choose to preload all the forms in the application at application startup. Doing this causes them to be displayed more quickly when the **Show** method for the forms is called at the appropriate time. Of course this technique will add to the time it takes to start your application initially.

Further Examples

Application	Form	Procedure/Event
Calculator	CALC.FRM	Form_Load Cancel_Click

The Unload Statement

The **Unload** statement is used to remove a form or control from memory. Unloading a form from memory may be necessary when the memory resources used are needed for some other object or when you need to reset properties to their original value. Another typical use for the **Unload** statement is in an OK button Click event for a form like an About box. There is no reason to keep the About box form in memory, because it is seldom used.

Note that when a form is unloaded, only the display component is unloaded. The code associated with the form module remains in memory.

Syntax

Unload *object*

Example

```
Sub Command1_Click ()
   Unload Form1
End Sub
```

End Statement

This statement ends a Visual Basic program, procedure, or block. By itself, the **End** statement stops program execution, closes all files, clears the values of all variables, and destroys all forms.

You have seen or will see various uses for the **End** statement in other modules to end portions of code. The topic appears here because you can use the **End** keyword alone in code to end your application.

Syntax

End [{Function | Select | Sub | Type }]

Statement	**Description**
`End Function`	Ends a **Function** procedure definition
`End If`	Ends a block **If...Then** statement
`End Select`	Ends a **Select Case** block
`End Sub`	Ends a **Sub** procedure
`End Type`	Ends a user-defined type definition

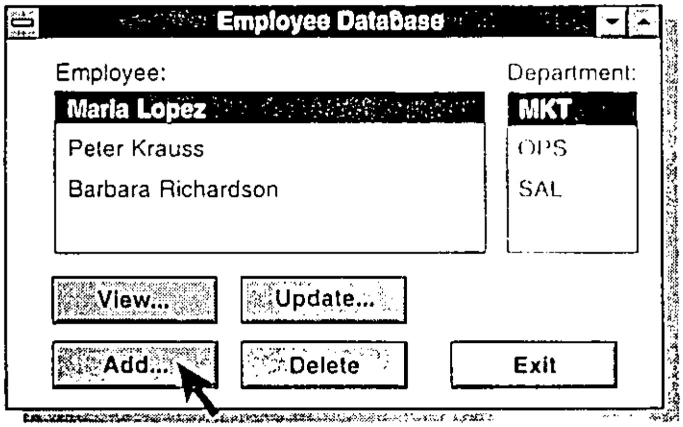
More examples are available in Visual Basic Help.

Example

```
'End Statement Example
Sub EndDemo ()
    Do                                          'Set up infinite loop
        Msg$ = "Enter A, B, C, or X."           'Enter X to exit
        UserInput$ = UCase$(InputBox$(Msg$))        'Get user input
        Select Case UserInput$                      'Evaluate input
            Case "A": Msg$ = "You entered 'A'."
            Case "B": Msg$ = "You entered 'B'."
            Case "C": Msg$ = "You entered 'C'."
            Case "X": End                       'End if user entered X
            Case Else: Msg$ = "You made an invalid choice.
                ^Try again."
        End Select
        MsgBox Msg$                             'Display results
    Loop
End Sub
```

Methods—Hide

EMPDB.HIDE



Employee DataBase

Employee:	Department:
Marla Lopez	MKT
Peter Krauss	OPS
Barbara Richardson	SAL

Hiding a Form

When a form is hidden, it is removed from the screen and its Visible property is set to **False**. A hidden form's controls are not accessible to the user, but they are available to the running Visual Basic application, to other applications communicating with the Visual Basic application through dynamic data exchange (DDE), and to timer events.

If a form is not loaded when the **Hide** method is invoked, the form is loaded but not shown.

Syntax

[form.] **Hide**

Note the optional form in the square brackets. Also note that the period within the brackets is not optional.

The following example is available from Visual Basic Help.

Example

```
'Hide Method Example
Sub Form_Click ()
    Dim Msg$                            ' Declare variable
    Hide                                ' Hide form
    Msg$ = "Choose OK to make the form reappear."
    MsgBox Msg$                         ' Display message
    Show                                ' Show form again
End Sub
```

Further Examples

Application	Form	Procedure/Event
Menu	MAIN.FRM	mnuAppMgr_Click
		mnuEditor_Click
		mnuNum_Click
		mnuToDo_Click

Walk Through—Hide vs. Unload

∑ **To see the difference between Hide and Unload**

1. From the Walk Throughs program group, start Hide vs. Unload.

   When the application starts you see two forms. Form1, on the left, contains two buttons: one to show Form2, the other to unload Form2.

   Form2, on the right, contains nine buttons, Command1 through Command8, and a button labeled Hide Form2. The eight command buttons were placed on the form to make sure that Form2 uses enough memory and Windows resources to illustrate the point.

   The purpose of the application is to demonstrate the difference of the impact on memory and Windows-based resources when hiding a form versus unloading it.

2. Click the Hide Form2 button.

   This hides the form.

3. In Program Manager, choose About Program Manager from the Help menu.

   From the Help menu in Program Manager, choose About Program Manager. Note the values for the two lines at the bottom of the form for amount of Memory and System Resources free.

4. Choose OK.

   Close the About Program Manager dialog box.

5. Press CTRL+ESC to display the Windows Task List.

6. Double-click Hide.

7. Click the Unload Form2 button.

   Unload your Form2.

8. In Program Manager, choose About Program Manager from the Help menu.

   Note the values for the two lines at the bottom of the form for amount of Memory and System Resources free.

   Now that Form2 has been unloaded, not just hidden, you should see more memory and/or resources available. The amount will vary depending on your machine's environment.

9. Choose the OK button.

   Close the About Program Manager dialog box.

10. Press CTRL + ESC.

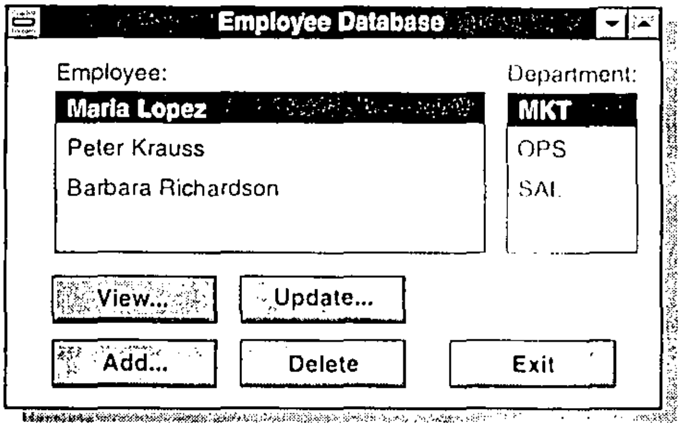
11. Select Hide from the Task List.

12. Click the End Task button.

    Terminate the project.

Methods — Show

EMPDB.SHOW



Employee Database

Employee:

Maria Lopez
Peter Krauss
Barbara Richardson

Department:

MKT
OPS
SAL

Showing a Form

Show is used to display forms that have been loaded. If you use the **Show** method on a form that has not been loaded, the **Show** method will automatically load the form and then show it.

Syntax

[*form.*] **Show** [*style%*]

Note The *style%* parameter is an integer value that determines whether the form is modal or modeless. If *style%* is 0, the form is modeless; if *style%* is 1, the form is modal.

Users must respond to a modal dialog box by clicking any one of a number of buttons. Any dialog box that lets the user continue working with other forms in the application is a modeless dialog box.

This example is available from Visual Basic Help.

Example

```
'Show Method Example
Sub Form_Click ()
    Dim Msg$                                        'Declare variable
    Hide                                            'Hide form
    Msg$ = "Choose OK to make the form reappear."
    MsgBox Msg$                                     'Display message
    Show                                            'Show form again
End Sub
```

Further Examples

Application	Form	Procedure/Event
Hello	GETSTART.FRM	mnuButterfly_Click mnuHello_Click mnuScrollbar_Click mnuShell_Click

Walk Through—Connecting Forms in Startup

∑ **To run the STARTUP.MAK project**

1. If Visual Basic isn't running already, start it.

   A new, blank form should be on the screen.

2. From the File menu, choose Open Project.
3. Open STARTUP.MAK.

   This should be located in \WALKTHRU\SAMPLES. Load the code, but do not run the application yet.

4. From an MS-DOS prompt copy ABOUT.FRM from \STUDENT1\ABOUT to \WALKTHRU\SAMPLES.
5. From the File menu, choose Add File.
6. Choose ABOUT.FRM.
7. Choose OK.

   This adds ABOUT.FRM to the .MAK file for the Startup application.

8. Select STARTUP.FRM in the Project Window and choose View Code.
9. In the Object list box, select mnuAbout.

   Go to the Object list box and locate the template for the mnuAbout_Click event. It should be empty.

10. Add the following code:

    ```
    frmAbout.Show 1
    ```

    Add the code that will show the About box when the user selects the Help menu and the About command. Remember the 1 causes this form to be modal. Notice the syntax here? FormName.Show.

    Important You might be tempted to use the **Load** statement to display the frmAbout form, but you will find that won't work here. Loading a form only means that it is in memory. It doesn't mean that the form is actually displayed on screen. For that you need to use the **Show** method.

11. From the Run menu, choose Start.

    Start the application to test the result.

12. Open the Help menu and choose the About command on the Startup application.

    The About box from the lab appears on screen.

    Note You will need to now change the Caption property on the About box form as well as the Icon and Text properties on the label to fit the current application.

13. Choose OK.

    Try to choose the OK button on the About Box form.

    You haven't coded that yet.

14. From the Run menu, choose End.
15. Select ABOUT.FRM.

    Select the ABOUT.FRM file in the Project window.
16. Choose View Code.

    Bring up the Code window.
17. In the Object list box, select cmdOK.

    Locate the code template for the cmdOK_Click event procedure.
18. Add the following code:

    ```
    Unload frmAbout
    ```

    Add that code on the blank line in the template.

    Notice the syntax here? The **Unload** statement comes first, and the object name follows. Notice also that there is no punctuation.
19. From the Run menu, choose Start.

    Start the application again to test the success of your changes.
20. Open the Help menu and choose the About command on the Startup application

    This should display the About box.
21. Choose OK.

    Choose the OK command button on the About box, and it disappears from the screen.

    Now the question is what do you do to end or close the application itself?
22. From the Edit menu, choose Exit.

    Try exiting from this command.

    In order to close the application entirely, you still need to make one more change to the source code.

    Note This placement of Exit is not standard for Windows. An application that follows interface guidelines for Windows would place Exit as the last item on the File menu.

23. From the Run menu, choose End.

    Stop the application.
24. Select frmStartup in the Project Window and choose View Code
25. In the Object list box, choose the Exit menu Click event procedure, and place the appropriate code there.
26. Add the following code:

    ```
    End
    ```

    Put this code on the blank line in the template.

27. From the Run menu, choose Start.

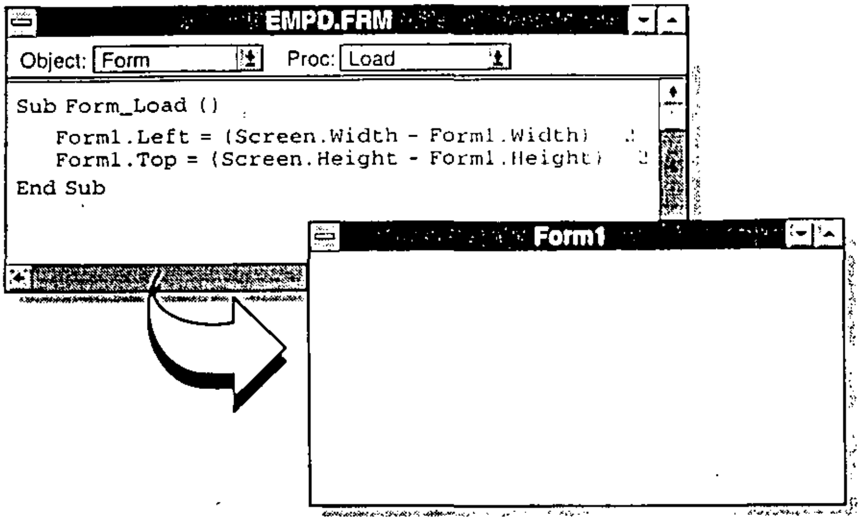
    Test the Exit command on the Edit menu, and test the About box elements.

28. From the Run menu, choose End.

29. From the File menu, choose Save Project.

    Save this project as STARTUP.MAK in \WALKTHRU\SAMPLES.

Event Procedures



```
Sub Form_Load ()
   Form1.Left = (Screen.Width - Form1.Width)   2
   Form1.Top = (Screen.Height - Form1.Height)   2
End Sub
```

Event Procedures

An event procedure is defined as a procedure invoked by a user or system-triggered event. Event procedures are always attached to a given form or control, and the syntax for an event procedure looks like the following:

Syntax

Sub *objectname_eventname*

statementblock

End Sub

Examples of event procedures are: Command1_Click and Form_Click.

Loading a Form

A Form Load event occurs when a form is loaded. It normally occurs when an application is run or as the result of either a **Load** statement or an implicit load that is caused by any reference to an unloaded form's properties or controls.

Typically you place initialization code for a form in a Form Load event procedure. For example, you would specify default settings for controls, initialize the contents of a combo or list box, and initialize form-level variables in a Form Load event.

Syntax

Sub Form_Load ()

'Initialization code

End Sub

Example

```
 Sub Form_Load ()
      ' Center the form on the screen
      Form1.Left = (Screen.Width - Form1.Width) / 2
      Form1.Top = (Screen.Height - Form1.Height) /2
 End Sub
```

Unloading a Form

An Unload event happens when a form is about to be removed from memory. When that form is reloaded, the contents of all the controls are reinitialized. Normally, this event is triggered by a user action, by choosing Close from the Control menu or by an Unload statement.

Many times the Form_Unload event procedure contains a **Select Case** statement to verify that unloading should proceed or to specify actions that need to be taken prior to unloading.

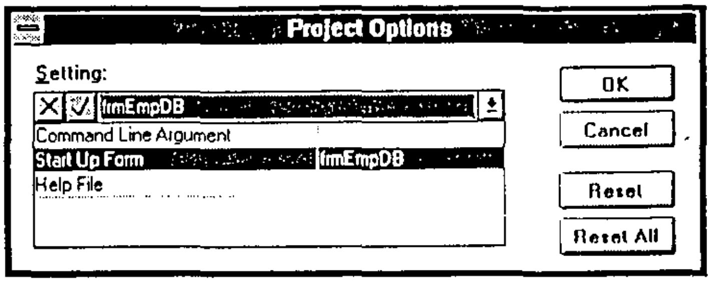
```
Sub Form_Unload (Cancel As Integer)

End Sub
```

Complete information on the Unload event is available from Visual Basic Help.

Further Examples

Application	Form	Procedure/Event
Menu	APPMGR.FRM	mnuDelApp_Click
		mnuExit_Click
	EDIT.FRM	mnuClose_Click
		mnuExit_Click
	NUMSYS.FRM	mnuExit_Click
	TDABOUT.FRM	Command1_Click
	TODO.FRM	mnuEdit_Click

Setting the Startup Form



Setting:

☒ ☑ frmEmpDB

Command Line Argument	
Start Up Form	frmEmpDB
Help File	

Setting the Startup Form

If you have a multiple-form application, by default the first form you create will be the startup form. It is easy to make another form the startup form as you will see in the demonstration.

Walk Through—Setting the Startup Form Comment

∑ **To set the startup form**

1. Start Visual Basic.
2. From the File menu, choose Open Project.

   Locate EMPLOYEE.MAK in \SOLUTION\FORMS
3. EMPLOYEE.MAK.

   Load the Employee Database application into the Project window.
4. From the Options menu, choose Project.
5. Choose the Select Startup Form option, and press the drop-down arrow on the combo box at the top.
6. Select frmEmpDB from the list.
7. Choose OK.

   Set the startup form.
8. Save the project.
9. Quit Visual Basic.

Summary

- Form Management

  Statements

  Methods

  Event Procedures

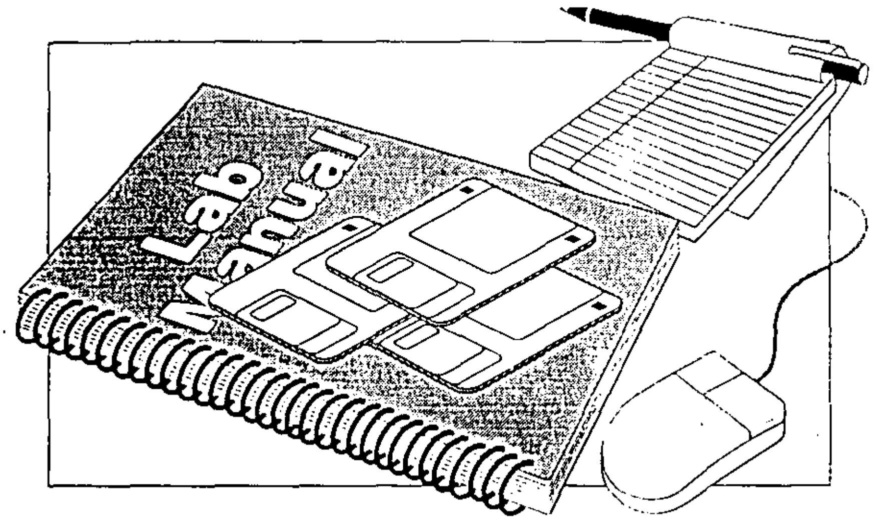
- Setting the Startup Form

Objectives

In this module you learned to:

- Use the **Unload** statement to unload a form from memory.
- Use the **Load** statement to load a form into memory.
- Use the **Show** method to display a form.
- Use the **Hide** method to hide a form but not unload it from memory.
- Set the startup form.

Lab Time



Lab Time

Go to the Loading and Unloading Forms portion of your lab manual.

Module 6: Using Controls

∑ Overview

- Types of Controls
- Properties for Controls

Overview

The purpose of this module is to make sure that you understand the general purpose of control properties and events as well as the essential properties for each of the controls in the Toolbox.

This is not intended to be a comprehensive treatment of all the possible properties and events for all of the controls. In addition, this module covers only half of the total number of controls in Visual Basic. The module following this covers the other half.

The general approach here is to let you practice changing the properties for each control and actually see how the change affects the control.

Prerequisites

Prior to starting this module, you should have a fundamental awareness of:

- The functionality of Microsoft Visual Basic
- The general use for each of the controls in the Toolbox
- The general methods for setting properties on controls

Overall Objective

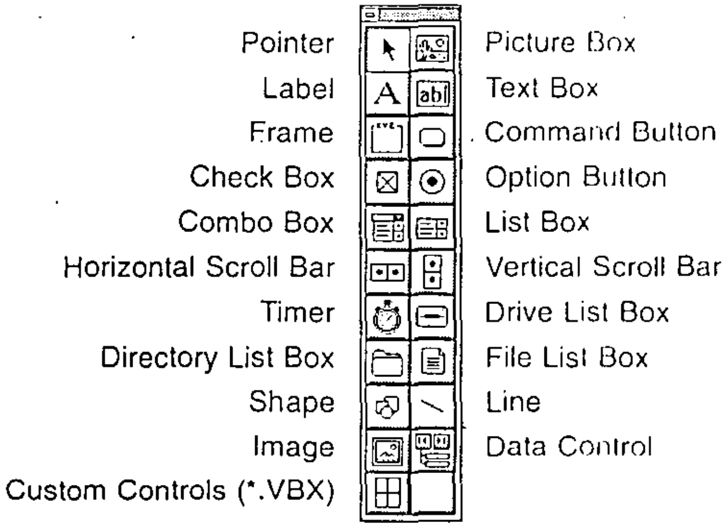
At the end of this module, you will be able to use key properties on half of the controls located in the Toolbox.

Learning Objectives

At the end of this module, you will have set key properties for:

- Labels
- Text boxes
- Frames
- Command buttons
- Check boxes
- Option buttons
- Combo boxes
- List boxes
- Horizontal and vertical scroll bars
- Timers
- Picture boxes

Types of Controls



This module covers the various clusters of controls found in the Toolbox, their most commonly used properties, and the events that are normally used with them. One of the first things you will see is that many of the controls have the same kinds of attributes. In many cases, a property will be covered only once, even though it shows up with almost every control.

Controls will be discussed in order of placement in the Toolbox, with the exception of the picture box, which will be covered last, and the drive, directory, and file list boxes, which will be discussed in a separate module.

A good way to get a handle on properties as they relate to controls is to look at the very front of the *Microsoft Visual Basic Language Reference*. There you can find a comprehensive list of controls and their properties.

Adding Controls to The ToolBox

The graphic on this page displays the default ToolBox. You can add controls to the Toolbox.

∑ To add a custom control to the Toolbox

1. From the File menu choose Add File.

   The Add File dialog is displayed.

2. From the Add File dialog, locate the custom control file that you want to add. For example, locate the file THREED.VBX. This file should be in the C:\WINDOWS\SYSTEM directory. This is a custom control provided by Visual Basic.

3. Select the file THREED.VBX and choose OK.

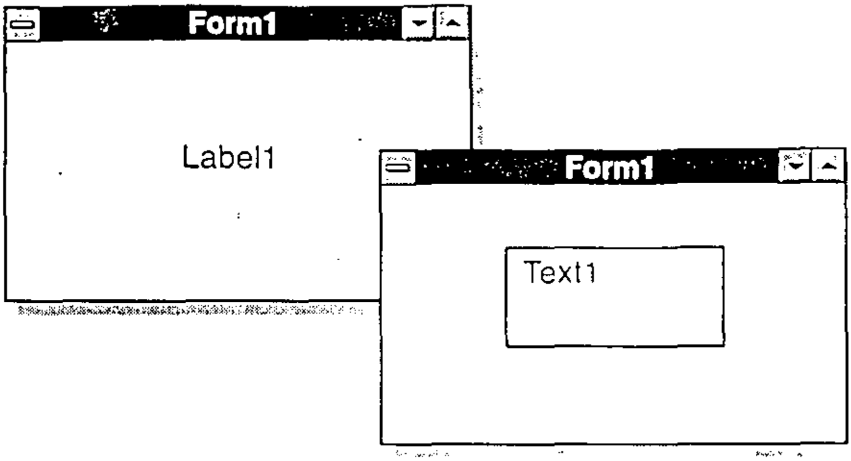
   The new controls are added to the Toolbox. You may get an error message if THREED.VBX has already been added to the Toolbox.

4. Use this same process to add any custom control to the Toolbox.

∑ Properties for Controls

- Labels, Text Boxes, and the Masked Edit Text Box
- Regular and 3-D Frames, Check Boxes, and Option Buttons
- Regular and 3-D Command Buttons
- Combo and List Boxes
- Horizontal and Vertical Scroll Bars
- Timers
- Picture Boxes

Labels and Text Boxes



Labels and Their Common Properties

Labels are most commonly used to display text that you don't want the user to be able to change. Typically this would be a caption under a graphic or captions for drive, directory, and file list boxes.

Property	Default	Comments
Alignment	0 - Left Justify	
AutoSize	False	Determines whether a control adjusts to fit its contents.
BackColor	White	
BorderStyle	0 - None	
Caption	Label1	Title for the control on screen. The caption is the default property for this control. Syntax - Label1 = "Files"
FontName	Helv	
FontSize	8.25	
ForeColor	Black	
Height	500	In twips
Left	–	From left border of form.
Name	Label1	Name used in code. Use lbl as a prefix.
TabIndex	Assigned by Visual Basic	Determines the tab order of the label.
TabStop	True	Determines whether use can use Tab key to move to the text box
Tag	–	User-defined value
Top	–	From top border of form.

Text Boxes and Their Common Properties

A text box control displays either information that is provided by the application or information that the user can type.

Typical uses of text boxes would include the box where users type the string of characters they want the application to find in a search function. If you were creating a front-end for a database, you could use a sequence of text boxes to gather information about members of the database:

Property	Default	Comments
Alignment	0 - Left Justify	
FontName	Helv	
FontSize	8.25	
Height	500	In twips.
Left	--	From left border of form.
MaxLength	0	Controls amount of user input. Ranges from 0 to 64 characters.
Multiline	False	Means no multiline capability.
Name	Text1	Name used in code. Use **txt** as a prefix.
PasswordChar		Specifies the character to be displayed when user types text in the text box.
ScrollBars	0 - None	
Text	Text1	For none, assign "". This is the default property for the control. Syntax—Text1 = ""
Top		From top border of form.
Width		Width of text box.

Walk Through—Text Box with a Scroll Bar

∑ **To use the text box with a scroll bar**

1. From the Walk Throughs program group, start Text Box with a Scroll Bar.

   When you start the application, you will see a form with a text box with a vertical scroll bar in the center of the form.

   The purpose of this application is to demonstrate the behavior of a text box with a scroll bar and how to create a text box with a scroll bar.

2. Type some text into the text box.

   Type enough text into the text box to cause it to start scrolling. Notice that the text will automatically wrap at the right margin. Notice also that you can select text and then cut and paste it just as in a standard Windows-based text box. Use CTRL+C to copy, CTRL+V to insert, CTRL+X to cut and CTRL+Z to undo.

3. Double-click the Control menu.

   Close the walk through.

4. Start Visual Basic.

5. Double-click the text box tool in the Toolbox.

   This will create a text box control.

6. In the Properties window drop-down combo box, make these changes:

Height	1000 (approximately)
ScrollBars	2 - Vertical

   Set the ScrollBars property to 2 - Vertical. Notice that you don't see the control change. There's a little problem here. Setting the ScrollBars property for a text box only becomes effective if the MultiLine property for the text box is also set to **True**.

7. Set the MultiLine property to True.

   Notice that a vertical scroll bar now appears on the text box.

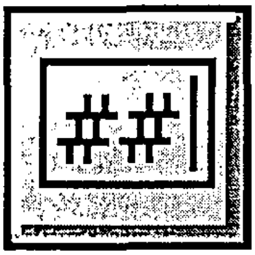
8. From the Run menu, choose Start.

   Test the behavior of the text box to see if it behaves as it did in the demo.

9. From the Run menu, choose End.

   Close the application.

Masked Edit Control



Masked Edit Control

This control is a special form of the text box control. It is used to restrict user input as well as to format data output.

Mask characters	Description
.	Decimal placeholder
,	Thousands separator
:	Time separator
/	Date separator
\	Treat next character as a literal
&	Character placeholder
A	Alphanumeric placeholder
?	Letter placeholder
#	Digit placeholder

Walk Through—Validating User Input

∑ **To monitor user input with the masked edit control**

1. From the Walk Through program group, start Masked Edit.

   The purpose of this walk through is to demonstrate one of the routine uses of the masked edit control—displaying a text box that prompts for and monitors the correctness of a user-added telephone number.

2. Type a telephone number.

   Notice that the use of the parentheses and the underlining prompts the user on the type and amount of information to be typed. Notice also that the masked edit text box automatically tabs the cursor to the next control as you type the telephone number.

3. Close the application.

∑ To create the application

1. Start Visual Basic.
2. Resize the form and set its properties as follows:

Height	2850 (approximately)
Width	4560 (approximately)

3. Place a label control at the top of the form and set its properties as follows:

Alignment	2 - Center
AutoSize	True
Caption	Customer's Telephone Number

4. Place the masked edit control in the center of the form and set its properties as follows:

Mask	(###) ###-####
Width	1635 (approximately)

   Visual Basic calls the parentheses and the hyphen "literals." They are the visual cues that tell the user the type of data that is expected, and you type them as part of the input mask. The number signs (#) are called "digit placeholders," and they specify that a digit is required. As you have seen, when you run the application, the control places an appropriate number of underscores that tell the user the total number of characters in the number as well as the user's relative position within the number.

5. From the Run menu, choose Start.

   Test the behavior of the masked edit control to see if it behaves as it did in the demo.

6. From the Run menu, choose End.

ValidationError Event

If a user types an invalid character, a ValidationError event occurs. You can place code in the ValidationError event to display an error message. For example, you might want to beep if the user types an invalid character.

1. Place the following statement in the MaskedEdit1_ValidationError event:

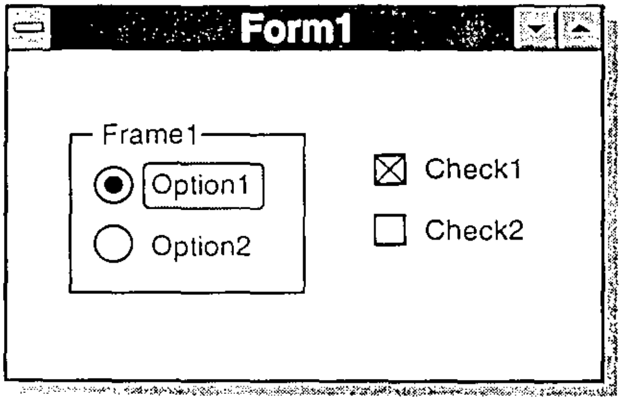
```
Beep
```

2. From the Run menu choose Start. Type a letter in the masked edit field.

   The system beeps.

3. Save the form and .MAK file with appropriate names in \WALKTHRU\SAMPLES.

Regular and 3-D Frames, Check Boxes, and Option Buttons



The frame, check box, and option button controls provide a way to present users with a set of options from which they can choose.

For example, if you start the Solitaire game in Windows, then open the Game menu, and then choose Options, you will see a dialog box that contains two sets of option buttons. Inside the frame labeled Draw are the options affecting how many cards are drawn. Inside the other frame labeled Scoring are the options that affect how the game is scored. The arrangement oñ the screen and the use of frames tell the user and the system that these are independent groups of option buttons. A choice made in the Draw group will not affect the Scoring group, and vice versa.

Frames and Their Common Properties

Frames provide a means for graphically and functionally grouping controls. Frames are most commonly used to group two or more option buttons in a set of mutually exclusive choices.

Note Information presented in frames and option buttons can also be presented in drop-down list boxes.

Property	**Default**	**Comments**
Caption	Frame1	On screen label. This is the default property for this control. Syntax—**Frame = "Definitions"**
Name	Frame1	Name used in code. Use **fra** as a prefix.
Visible	True	**False** hides the frame and any controls inside it.

Caution When Placing Controls on Frames

There are two methods for placing controls on a form. The simplest method is to double-click a control icon. An instance of the control will appear in the center of the form, and you can position the control from there. This is a particularly useful approach when you are placing several controls on a single form.

The second approach is to single-click a tool in the Toolbox, move the cursor to the form, and drag the control on the form until it is the size you want. *This second method is required for placing controls on a frame.* If you do not use this method, you will lose the controls when you try to move the frame around on the form.

3-Dimensional Frames

Added property	Default	Comments
Alignment	0 - Left Justify	You cannot center- or right-justify frame captions with the other controls.
Caption		This is the default property for this control. Syntax—**Frame1 = "Definitions"**
Font3D	0 - None	Sets the amount and direction of shading in the Caption.
Name	Frame3D1	Name used in code. Use **fra** as prefix.

Check Boxes and Their Common Properties

Property	Default	Comments
Caption	Check1	On-screen value
Enabled	**True**	**False** disables all user access.
Name	Check1	Name used in code.
		Use **chk** as a prefix.
Value	0 - Unchecked	1 - Checked; 2 - Grayed.
		(Fills box with gray). This is the default property for the control.
Visible	**True**	**False** hides the check box.

Events

Click Normally a Click event indicates that the user has made a selection. In the case of a check box, the Click event acts as a toggle. If there is an X inside the check box when the user clicks it, the X will be removed. If the check box is empty, an X will be placed inside the check box after the user clicks it. In either case, the Value property will be reset. A typical use of a check box might be to allow the user to choose between bold or normal font.

Option Buttons and Their Common Properties

Option buttons, as noted previously, are used to display an array of choices from which the user may select only one. An example is found in the Microsoft Windows Write Print Setup dialog box, where the user can select either the default printer or a specific printer and either portrait or landscape page orientation.

Property	Default	Comments
Caption	Option1	On-screen value.
Name	Option1	Name used in code. Use **opt** as a prefix.
Enabled	True	User has access to control.
Value	False	This is the default property for this control. Syntax—**Option1 = False**

Style Guidelines

Always have one option button in a group selected as the default for the user when the form is displayed.

Events

Click When the user clicks an option button, the Click event actually invokes several activities. It paints the dot on the selected option button, removes the dot from any other option button within the frame, and resets the appropriate values for all option buttons in the same group.

Walk Through—Option Buttons and Displaying Their Values

Σ **To compare the effect of the Click event for a check box versus an option button**

1. From the Walk Throughs program group, start Check versus Option.

   When you start the application, you will see a form with three option buttons, two check boxes, and a clear screen button on the right side.

   The purpose of this application is to compare the effect of the Click event for a check box versus an option button.

2. Click the Option1 option button.

   The values for all three buttons should be displayed on the left side of the form.

3. Click the Option2 option button.

   The updated values for all three buttons should be displayed on the left side of the form.

4. Click the Check1 check box to place an X in it.

   The value of Check1.Value is printed on the form.

   Compare the number that is printed for the value of a check box when it is selected (1) to the value that is printed for an option button when it is selected (-1).

5. Click the Check2 check box to place an X in it.

   The value of Check2.Value is printed on the form.

   Compare the effect of the Click event for check boxes versus option buttons. Clicking one option button affects the state of the other option buttons in its group. Clicking a check box will not affect the state of other check boxes.

6. Double-click the Control menu.

   Close the walk through.

∑ To code values for option and check boxes

1. If Visual Basic is not running already, start it.
2. Double-click the frame control tool in the Toolbox.

   Add a frame to the form. Click and drag the sizing handles to make it big enough to hold three option buttons. Place it near the lower-right corner of the form so that there is plenty of room to display the values for the three option buttons you will add.

3. Click the option button tool in the Toolbox.
4. Move the mouse over the frame you just created on Form1.

   Note that the mouse pointer has changed from an arrow to a cross hair.

5. Click and drag inside the area of Frame1 to draw an option button there.
6. When the button is the size you want, release the mouse.
7. Draw two more buttons inside Frame1.
8. Double-click the top option button.

   This displays the FORM1.FRM Code window with the Option1_Click event procedure template.

9. Add the following code:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

   This adds the code that prints the values for the buttons.

10. In the Object list box, select Option2 and locate the Click event template from the Procedure list box.
11. Add the following code:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

   This adds the same code that prints the values for the buttons to the Click event for Option2.

12. Using the Copy command, place the code above into the Clipboard.
13. In the Object list box, select Option3 and locate the Click event template from the Procedure list box.

14. Using the Paste command, copy the contents of the Clipboard to this event procedure template.

    The code should look like this:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

    This adds the same code that prints the values for the buttons to the Click event for Option3.

15. From the Run menu, choose Start.

    Test your code.

16. From the Run menu, choose End.

    Close the application.

17. Quit Visual Basic.

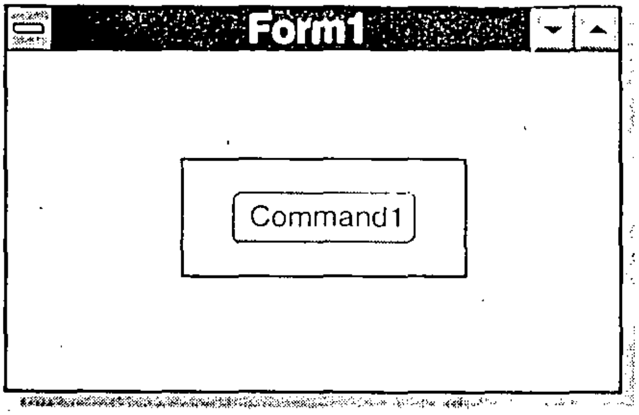
3-Dimensional Frames, Option Buttons, and Check Boxes

For the most part, these three controls are identical to the three controls just discussed. However, there are a few subtle differences that need to be noted as well as the procedures for setting the 3-D qualities for these controls.

3-Dimensional Option Buttons and Check Boxes

Added property	Default	Comments
Name (for check boxes)	Check3D1	Name used in code. Use chk as a prefix.
Name (for option buttons)	Option3D1	Name used in code. Use opt as a prefix.
Font3D	3 - Inset with Light Shading	Sets or returns the three-dimensional style of the caption. For example, you can set Font3D to have font appear raised or inset with light or dark shading.

Regular and 3-D Command Buttons



Command Buttons and Their Common Properties

A command button performs a task when the user presses it.

Property	Default	Comments
Cancel	False	True—button activated by ESC.
Caption	Command1	On-screen value.
Default	False	True—button activated by ENTER.
Enabled	True	False disables the button for users.
Height	500	
Left	–	From left border of form.
Name	Command1	Name used in code. Use cmd as a prefix.
Top	–	From top border of form
Width	1215 twips	
Value		Syntax—Command1 = Enabled

Style Guidelines

Place buttons in a group on a form, either horizontally along the bottom of the form, or vertically along the right side of the form.

Events

Click With command buttons, a Click event normally means that the user wants some kind of action to take place. In the case of an OK button, for example, users might want a form removed from the screen so that they can continue work. In this case, you will be providing code.

When the user uses the mouse pointer to choose a command button, the Click event changes the color of the button (to simulate its being pressed) and executes whatever code you have added to the Click event procedure. A user can also use the keyboard to generate a Click event for a button by tabbing to the button and then pressing the SPACEBAR.

Example

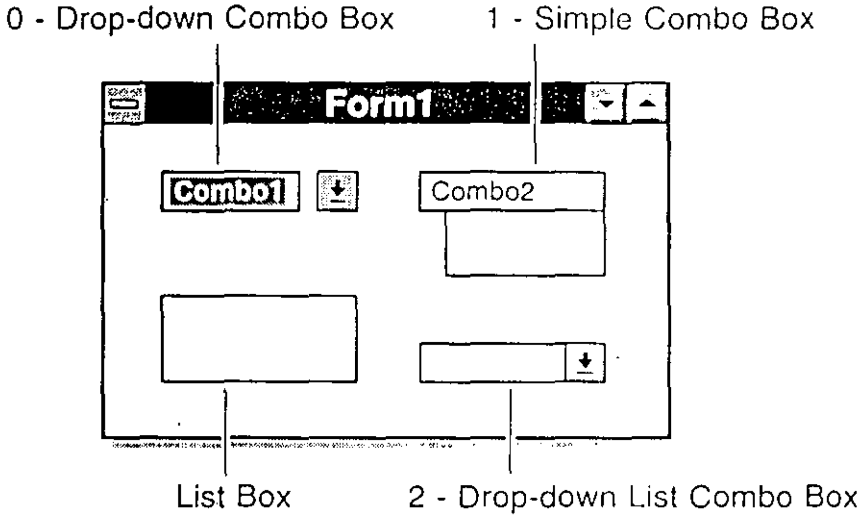
```
Sub Command1_Click ()
    ' Stop all execution of the program
    End
End Sub
```

3-Dimensional Command Buttons

This form of the command button behaves in much the same manner as the other command buttons, but there are a couple of interesting differences.

Property	Default	Comments
Bevel Width	2	Specifies the thickness of the highlight and shadow around the button. Valid range is from 0 to 10.
Name	Command3D1	Name used in code.
Outline	True	False disables the border or line edging the button.
Picture	None	Enables placing a picture or icon on a command button.

Combo and List Boxes



There are actually four kinds of list box controls, each of which has a slightly different functionality.

Both drop-down styles of boxes are designed to save on the initial amount of screen space in use by the control.

Users can only add new items directly into either a drop-down or simple combo box. With a list or drop-down list, they are limited to the choices provided by the application.

List Boxes and Their Common Properties

List boxes display a list of items from which the user can select only one at a time. Through the list box control, users can only select from the given list of articles, but you can write code that allows them to add and delete items during run time through some other control.

Automatic Scroll Bars

All list and combo boxes will automatically add a vertical scroll bar if the number of items contained in the list is greater than what will fit in the maximum display size of the control, which is set by the Height property.

Property	Default	Comments
Columns	Default	Enables multiple column display of data in a single list box.
List		Accessible only at run time. Sets or returns the items contained in a control's list string array.
ListCount		Returns the number of items in the list.
ListIndex		Returns the index of the selected item in the list.

Property	Default	Comments
MutliSelect	0 - None	Other options: 1 - Simple and 2 - Extended. None means that the user can select one within the list. Simple means user can select multiple items using the cursor. Extended means user can press CTRL to select multiple items.
Name	List1	Name used in code. Use lst as a prefix.
Sorted	False	
Text		This is the default property for this control. Syntax—List1 = "Item1"

Events

Click With any of the list boxes, a Click event occurs when users select an item from the list to indicate their preferences.

DblClick A double click normally is used as a shortcut that combines two actions: the selection and the starting up of a process.

Methods

These are used for list management.

Name	Function	Syntax
AddItem	Used to add individual items to list box. Seeding a list box is done from Form_Load.	**List1.AddItem "*name*"**
RemoveItem	Used to remove items one at a time from a list box.	**List1.RemoveItem (ListCount)**
Clear	Used to remove all items in a list box in one stroke.	**List1.Clear**

Walk Through—Adding and Removing List Items

∑ **To add and remove list items**

1. From the Walk Throughs program group, start List Box.

   When the application starts up, the cursor should already be placed in the text box.

   The purpose of this application is to demonstrate a simple procedure for adding items to and removing items from a list box.

2. Type **Harwood, Gene** in the text box.

   Add a new name in the text box.

3. Choose Add.

   You can do this with either the mouse or the ENTER key.

4. Select the text box.

   Tab to or click the text box.

5. Type **Simons, Anna** in the text box.

   Add a second name to the list box, and a vertical scroll bar should appear on your display.

6. Choose Pfeiffer, Terry.

   Single-click this list item.

7. Click the Remove button.

   The list item is removed from the display.

   The question here is this: How does all of this work?

∑ To code list box handling

1. If Visual Basic isn't running already, start it.
2. From the File menu, choose New Project.

> **Note** If you want to do all of the work for this walk through, go to the next step and start changing the properties for the form. If you want to start from a completed interface, you can also find a partially completed form called LIST.MAK in \WALKTHRU\SAMPLES. If you choose this option, skip down to step 12, the point where you start adding code to the Form_Load event.

3. Set the properties for the form as follows:

Caption	Consultants
Name	Form1
Height	3075 (approximately)
Width	3420 (approximately)

4. Double-click the list box tool in the Toolbox.

   Add a list box to the form.

5. Set the following properties:

Name	List1
Height	1005 (approximately)
Sorted	**False**
Width	2775 (approximately)

6. Double-click the text box tool in the Toolbox.

   Add a text box to the form.

7. Set the following properties:

Name	Text1
Height	555 (approximately)
TabIndex	1
Text	Blank
Width	2775 (approximately)

8. Double-click the command button tool in the Toolbox.

   Add a command button to the form.

9. Set the following properties to create an Add button.

Caption	Add
Name	Command1
Default	True
Height	555 (approximately)
Width	1215 (approximately)

10. Double-click the command button tool in the Toolbox.

    Add a second command button to the form.

11. Set the following properties to create a Remove button.

Caption	Remove
Name	Command2
Height	555 (approximately)
Width	1215 (approximately)

Now that you have created the interface, you need to add the code to make it function.

12. In the Project window, choose the View Code button.,

    Make sure that the Form is selected.

13. In the Object list box, select Form.
14. In the Procedure list box, select Load.

    Add the initialization code that loads the names into the list box when the form is loaded at run time.

```
  ' Fill the list box at run time
List1.AddItem "Morton, Joan"
List1.AddItem "White, Don"
List1.AddItem "Marshall, Lynn"
List1.AddItem "Pfeiffer, Terry"
```

15. In the Object list box, select Command1 and locate the Click event procedure in the procedure list box.
16. Add the following code:

```
List1.AddItem Text1.Text
```

This code takes whatever is in the Text property of the text box and adds it to the list box.

17. From the Run menu, choose Start.

    Run your application.

18. Type **Harwood, Gene** in the text box.

    Type Gene Harwood's name in the text box and press ENTER or click Add. His name should be added to the list box but not deleted from the text box.

19. From the Run menu, choose End.

    In order to clear the text box, you need to add a second line to the Click event procedure here.

20. Add the following code to Command1_Click:

```
Text1.Text = ""
```

This takes care of the simple addition of items to a list box.

21. From the Run menu, choose Start.

Try adding Gene Harwood's name one more time. It should work just fine, but what about removing it?

22. From the Run menu, choose End.

Removing items from a list box is fairly straightforward.

23. In the Object list box, select Command2_Click.

24. Add the following code:

```
List1.RemoveItem List1.ListIndex
```

This code removes the item indicated by the value of List1.ListIndex. The **RemoveItem** method removes items one at a time. With small lists, this might be suitable. With larger lists, you would use the Clear method.

The above code will cause an error if no list item is selected. You can avoid this by modifying the code as follows:

```
If List1.ListIndex = -1 Then
    msg$ = "Please Select a Consultant"
    MsgBox msg$, 64, "List Selection"
Else
    List1.RemoveItem List1.ListIndex
End If
```

25. From the Run menu, choose Start.

Add and delete a couple of names to see that the code works.

26. From the Run menu, choose End.

Close the application.

How about sorting this list? Don't you have to add a lot of code to sort the items in the list? No, you don't! You simply set the Sorted property to **True** for List1, and all items inserted in the list will be sorted automatically.

27. Close the Code window.

28. From the Properties list box, select Sorted for the list box.

Click the List1 list box on the form.

Set the Sorted property to **True**.

29. From the Run menu, choose Start.

Note that the names in the list are now sorted. Add and delete a couple of names to see that the sorted order of the list is maintained.

30. From the Run menu, choose End.

Close the application.

31. Save your form and .MAK files with appropriate names in \WALKTHRU\SAMPLES. You can overwrite the file if it already exists.

Walk Through—Combo and List Box Differences

∑ **To add entries to combo and list boxes**

1. From the Walk Throughs program group, start Combos.
2. Type **Fred** in the drop-down combo box.

   If you press the DOWN ARROW key, the list of names is displayed.
3. Press ENTER.

   The name Fred is added to the list. If you add several more names, a vertical scroll bar is also added to the list.
4. Type **Natasha** in the simple combo box.
5. Press ENTER.

   The new name will be added to this list. Notice here that the scroll bar is also added.
6. Select the name Larry in the drop-down list box.

   This name appears in the box at the top of the list, but users cannot add items to the list directly.
7. From the Control menu choose Close.

   Close the application.

Combo Boxes and Their Common Properties

Property	Default	Comments
Name	Combo1	Name used in code Use cbo as a prefix.
Style	0 - Dropdown Combo	1 - Simple Combo 2 - Drop-down List
Text	Combo1	
Height	300	Simple combo—The height displayed both at design time and run time. Drop-down Set height property at design time (or run time), but full height will not be displayed until the user drops down the list at run time.
Text		This is the default property for the control. Syntax Combo1 = "Natasha"

Events

Change A Change event indicates that the contents of a control have changed. In a combo box, a Change event occurs whenever you edit the text box portion of the combo box or when you assign a new value to the Text property from code.

Drop-Down and Simple Combo Boxes

Drop-down combo boxes are an alternative to option buttons within frames; that is, they are designed to display a limited number of logically arranged choices from which the user may select one. A classic example of a drop-down combo box is found in the Windows Control Panel. From the Settings menu, the user can choose Ports and then set the baud rate for the modem at any one of eight possible values, or type a value into the text box portion of the control. The new value typed can then be added to the list. The drop-down combo box enables a scrollable list that allows the user to review more choices than can be easily presented on screen. The user's choice is displayed in the top box.

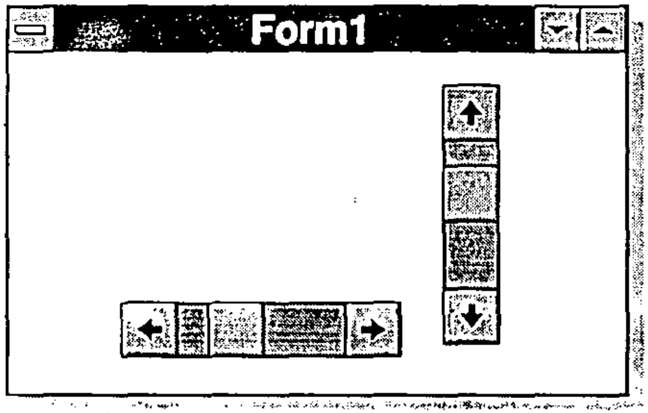
Drop-Down List Combo Boxes

Drop-down list combo boxes are used to present users with a limited number of choices, from which they may select only one.

Key Feature

Both drop-down combo and drop-down list combo boxes display data from which the user may select only *one* choice. But with a drop-down list users can only select from the list; they cannot add to it.

Scroll Bars



Scroll Bars and Their Common Properties

Scroll bars are most commonly used to enable quickly moving across a long list of items, such as a long list of filenames. They can also be used for indicating the current position on a scale, such as shades of coloring for customizing the look and feel of the screen. Or, scroll bars might be used to indicate the volume of an audio system.

Property	Default	Comments
LargeChange	1 (1–32,767)	Amount of change when user clicks scroll bar shaft.
Max	32,767	Maximum value of scroll bar handle in bottom most position.
Min	0	Minimum value of scroll bar handle in topmost position.
Name	HScroll1	Name used in code.
	–or–	Use hsb as a prefix.
	Vscroll1	Name used in code.
		Use vsb as a prefix.
SmallChange	1 (1–32,767)	Amount of change when user clicks a scroll arrow.
Value	0	This is the default property for this control. Can be set by user's direct interaction with the control or through code in response to other events.

Events

Change Occurs when the contents of the control have changed.

Walk Through—Scroll Bars, Properties, and the Change Event

∑ **To see scroll bars, properties, and the Change event**

1. From the Walk Throughs program group, start Scroll Bars.

   You will see a form with a vertical scroll bar on it.

   The purpose of this application is to demonstrate the behavior of a scroll bar and the effects of the properties LargeChange and SmallChange on the Value property.

2. Click the arrows on both ends of the scroll bar.

   Note the change in the Value property.

3. Click on the shaft of the scroll bar both above and below the scroll box.

   Note the change in the Value property.

4. Close the application.

5. Start Visual Basic.

6. From the File menu, choose New Project.

   Start a new project.

7. Set the following form properties in the Properties window:

Height	4425 (approximately)
Width	5430 (approximately)

8. Double-click the vertical scroll bar tool in the Toolbox.

   Add a vertical scroll bar and change some of its properties:

Height	3495 (approximately)
LargeChange	20
Max	100
SmallChange	10
Width	375 (approximately)

9. In the Project window, choose the View Code button.

   Make sure that Form1 has been selected.

10. In the Object list box, select VScroll_Change.

    Locate the Change event procedure for the vertical scroll bar.

11. Add the following code:

    ```
    Print "VScroll1.Value = "; VScroll1.Value
    ```

    Display the Value property each time it changes.

12. From the Run menu, choose Start.

    Test the application you have written.

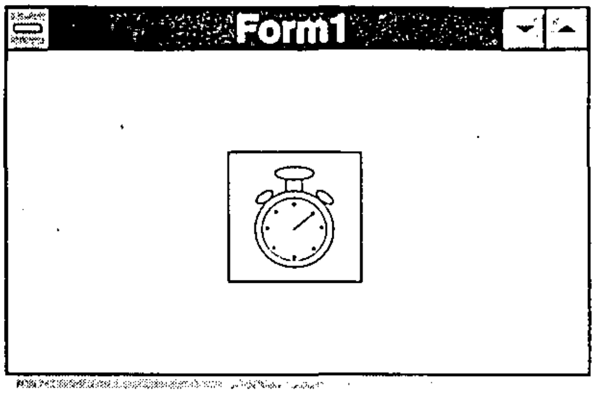
13. From the Run menu, choose End.

    Close the application.

14. Quit Visual Basic.

15. Save your form and .MAK files with appropriate names in \WALKTHRU\SAMPLES.

Timers



Timers and Their Common Properties

Timers are used to run events at a specific time or within an interval that you specify. For example, within a scheduling application, timers would be used to enable alarms for the user.

A special characteristic of this control is that it is never visible at run time.

Property	Default	Comments
Enabled	**True**	This is the default property for this control. Syntax: **Timer1 = Enabled**
Interval	0	Countdown interval for the timer, measured in milliseconds.
Name	Timer1	Name used in code. Use **tmr** as a prefix.

Walk Through—Clock Application and the Timer Control

∑ **To use the timer control**

1. From the Walk Throughs program group, start Clock (Timer).

   When you start the application, you will see a form containing a digital clock displaying the date and time. The clock will be updated every second.

   The purpose of this application is to demonstrate the behavior and use of a timer control.

2. Double-click the Control menu.

   Close the walk through.

3. Start Visual Basic.

4. From the File menu, choose New Project.

   Start a new project.

5. Set the properties for the form as follows:

BackColor	Yellow
BorderStyle	3 - Fixed Double Also removes Min/Max buttons
Caption	Clock
Height	2235 (approximately)
Name	frmClock
Width	4345 (approximately)

6. Double-click the label tool in the Toolbox.

   Add a label to the form.

7. Set the properties for the label as follows:

Alignment	2 - Center
BackColor	Red
Caption	(none)
FontSize	12
ForeColor	White
Height	1340 (approximately)
Left	120
Name	lblDateTime
Top	420
Width	4015 (approximately)

8. Double-click the timer tool in the Toolbox.

9. Set the following properties:

Name	tmrTimer1
Interval	1000

   Place a timer on the form. Set the properties for it. You can place it anywhere. Behind the label will be fine, because timers are not displayed on screen at run time

10. In the Project-window, choose-the-View-Code window.

    Make sure that frmClock has the focus.

11. In the Object and Procedure list boxes. locate the tmrTimer1_Timer.

    Locate the Timer event in the code.

12. Add the following code:

```
lblDateTime.Caption = Format$(Now, "mmmm d, yyyy h:mm:ss am/pm")
```

    Add the code on the blank line in the Click event template.

13. Double-click the Control menu.

    Close the Code window.

14. From the Run menu, choose Start.

    Test the clock application that you have just created.

15. From the Run menu, choose End.

    Close the clock.

16. Quit Visual Basic.

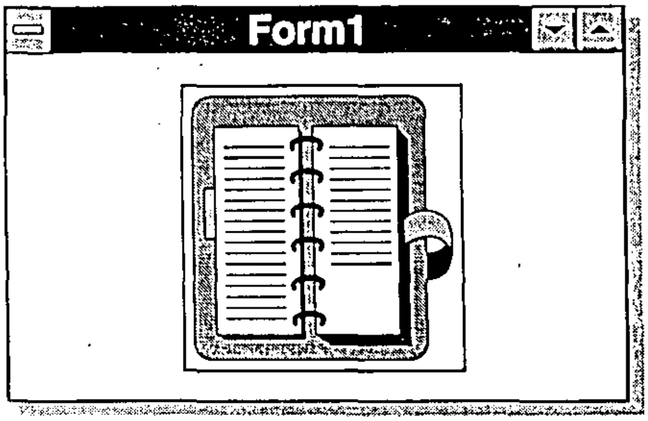
17. Save your form and .MAK files with appropriate names in \WALKTHRU\SAMPLES.

Using a Timer to Initiate a Task at a Specific Interval

You can use the timer to initiate a task after a specific interval. In the following example, the timer event starts a backup procedure. The timer event occurs after at least 60 seconds have elapsed. You could place code in the Timer event to check the current time and start the backup routine only at certain times.

```
Sub Form_Load ()
    Rem Timer will go off after 60 seconds.
    timer1.Interval = 60000
    timer1.Enabled = True
End Sub

  Sub Timer1_timer ()
      x = Shell("c:\mybackup.bat")
  End Sub
```

Picture Boxes



Picture Boxes and Their Common Properties

Picture boxes are used to display a range of graphics, from icons to bitmaps to metafiles. You could use a picture box, for example, to display the icons in the Visual Basic icons library. You could also use the picture control to display the kinds of card backs available for users of a Solitaire game.

Property	Default	Comments
AutoRedraw	False	Will not automatically repaint.
AutoSize	False	Will not automatically resize control to fit its contents.
Name	Picture1	Name used in code. Use pic as prefix.
Picture	(none)	Use Load Picture dialog box at design time to select a .BMP, .WMF, or .ICO file. This is the default property for this control. Syntax—**Picture1 = LoadPicture("\*path*")**

Events

Paint This event occurs when a form or control is moved, exposing parts that weren't initially painted on screen. Note that this applies only when AutoRedraw is set to False. If AutoRedraw is set to **True**, repainting is done automatically, and no Paint event occurs.

Loading a Picture at Run Time

To load a new picture file from disk into a picture box at run time, call the **LoadPicture** function with the name of the file you want to load.

Example

```
Sub Command1_Click ()
   Picture1.Picture = LoadPicture ("\VB\ICONS\ARROWS\POINT12.ICO")
End Sub
```

If you want to copy a picture from one picture box to another at run time, use the Image property of the picture you wish to copy.

Example

```
Sub cmdCopyPicture ()
   Picture2.Picture = Picture1.Image
End Sub
```

Clearing a Picture Box

At design time, clear the Picture property by clicking the picture box you wish to clear. Make sure the Picture property is selected in the Properties list, click the settings box, and then press the DEL key. The settings box will now read (none).

At run time, clear a picture box's Picture property by calling the **LoadPicture** function with an empty argument.

Example

```
Sub Command2_Click ()
   Picture1.Picture = LoadPicture ()
End Sub
```

Walk Through—Picture Box Controls and AutoSizing

∑ **To use picture box controls and autosizing**

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Open a new blank form.
3. Double-click the picture box icon in the Toolbox.

   Place a picture control on the form.
4. Set the following properties for the control:

AutoSize	**True**
Height	1000 (approximately)
Width	1000 (approximately)
5. Place an icon in the picture control.
6. Set the Picture property to:

   \VB\ICONS\ARROWS\ARW01DN.ICO

   The picture box automatically resizes to fit the graphic when it is added.
7. Place a bitmap in the picture control.
8. Set the Picture property to:

   \VB\BITMAPS\GAUGE\CIRCLOCK.BMP

   The picture box automatically resizes to fit this graphic.
9. To remove the Picture property:
   Click the Name property and then click back on the Picture property.
   Click in the Settings combo box for the Picture property.
   Press the DEL key

   The settings box will read (none). This will remove the graphic from the Picture property of the picture box.
10. Quit Visual Basic.

Summary

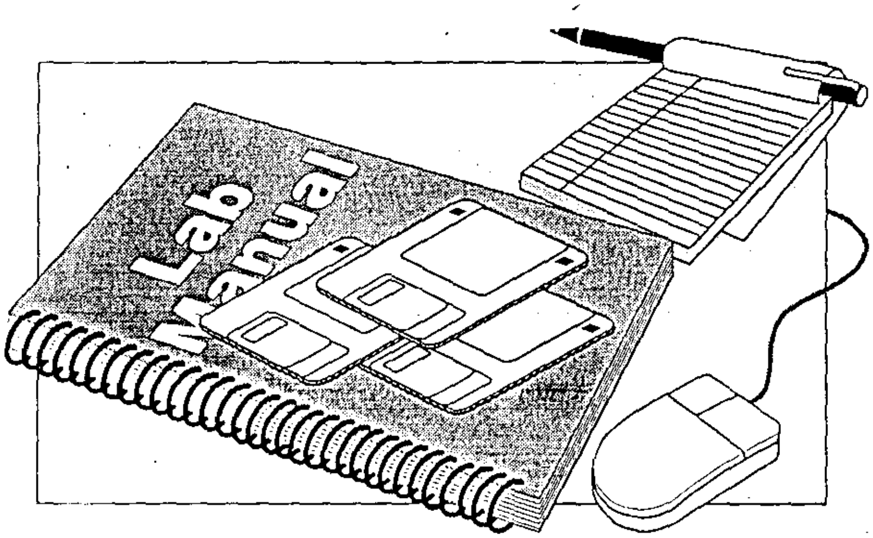
- Types of Controls
- Properties for Controls

Objectives

In this module you set key properties for:

- Labels
- Text boxes
- Frames
- Command buttons
- Check boxes
- Option buttons
- Combo boxes
- List boxes
- Horizontal and vertical scroll bars
- Timers
- Picture boxes

Lab Time



Go to the Adding Controls to Forms portion of your lab manual.

Module 7: File Browsers and Other Controls

∑ Overview

- File Browsing
- Grid Control
- 3-D Panel Controls and Group Push Button

Overview

This module is designed to directly follow the controls module, and it covers most of the rest of the tools in the Toolbox. However, because of time limits it does not cover picture clip, lightweight line and shape, image, and serial communications controls.

The first part of the module discusses file browsing controls—file, directory, and drive list boxes as well as the common dialog control. The first three are specialized controls that act just like regular list boxes; but as you will see, they have special features that make them distinct.

The purpose of this module is to walk you through the flow of events that are needed to connect the various controls. The emphasis here is on talking about the kinds of events that need to be used in order for the changes on one control to be reflected on the other control.

The rest of the module provides you with discussion of and walk throughs for most of the rest of the tools in the Toolbox.

Prerequisites

This module assumes prior experience with:

- List boxes
- Click event implementation

Overall Objective

The overall objective is for you to develop an awareness of the rest of the tools in the Toolbox.

Learning Objectives

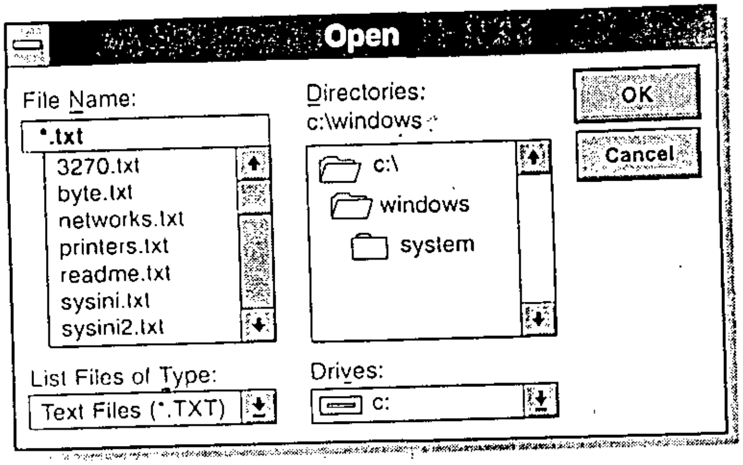
At the end of this module, you will be able to:

- Design and develop a form that lets users browse drives, directories, and files.
- Use the Change and PathChange events.
- Use the grid control.
- Use the group push button and 3-D panel controls.

∑ File Browsing

- Scenario
- Properties and Events for File Browser Controls
- The Common Dialog Control

Scenario



File Name: *.txt

Directories: c:\windows

List Files of Type: Text Files (*.TXT)

Drives: c:

File Browsers

A quick review of the workings of the file browser in Notepad shows that a number of events occur when the user selects a different drive or directory. If the user selects a new drive, the directories and files list boxes are automatically updated, showing the current working directory for the system and any files (of the file type specified in the List Files Of Type list box).

Walk Through—How File Browsers Work

∑ **To use file browsers**

1. In the Accessories group window, open Notepad.

   All of you have used a file browser of one sort or another, but you may not have looked closely at how it works.

2. From the File menu, choose Open.

   Drives list box The Drives list box is a special form of drop-down list box. The user can see options presented in the list but cannot add items to it.

   The more important element to notice at this point, however, is that when the user selects a new drive, the change in selection is reflected in the directory and file list boxes as well as in the current working directory.

   Directories list box The Directories list box is a special form of list box. It behaves like a regular list box but displays only directories and subdirectories. Users cannot add items to or delete items from the list.

   File Name combo box The File Name combo box is a special form of the list box that sends and receives messages from a text box. The text box allows users to add items to the list, something that cannot be done with a list box.

   List Files of Type Combo Box Notice that this, too, is a drop-down list box. Users can select items from it, but they cannot add items to it themselves.

   Current working directory Finally, notice that there is a label that tells the user the current drive and directory.

3. Choose Cancel, and close Notepad.

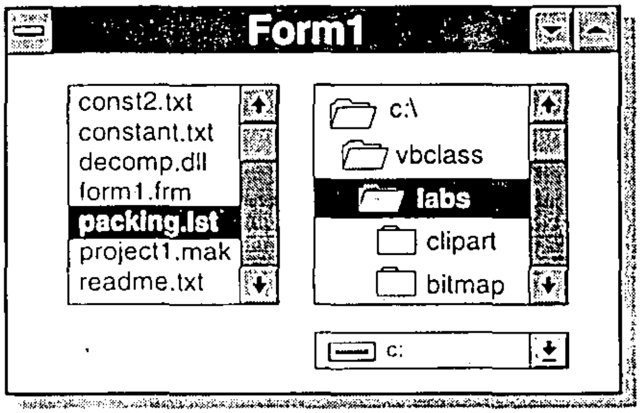
   End walk through.

Style Guidelines

Typically, file browsers can be set to display only a given type of file in the specified drive and directory. Because Notepad is designed to read files that have the .TXT extension, .TXT is the default file type and *.* is the second choice.

Because the Employee Database application is designed to locate .BMP and .WMF files, they will be the default file types.

Properties and Events for File Browser Controls



Drive, Directory, and File List Boxes

All three of these controls are special forms of list boxes. They have many of the same characteristics, but there are also a number of special properties that they use to enable navigating through a file system.

Drive List Boxes and Their Common Properties

The first of these controls is the drive list box. It is a special form of the drop-down list box. Initially it only shows the current drive for the form. However, if the user clicks the drop-down arrow, the control will display all of the other drives currently available on the system.

Property	Default	Comments
Name	Drive1	Name used in code. Use **drv** as a prefix.
Drive		Accessible only at run time, this property returns the selected drive. This is the default property for this control.

Synchronizing the Drive Property

Example

```
Sub Drive1_Change ()
   'When drive is changed, reset the directory list box path
   Dir1.Path = Drive1.Drive
End Sub
```

Directory List Boxes and Their Common Properties

Property	Default	Comments
Name	Dir1	Name used in code. Use dir as a prefix.
Path		Accessible only at run time, this property returns the absolute path for the selected directory. This is the default property for this control.

Synchronizing the Path Property

Example

```
Sub Dir1_Change ()
   ' When directory is changed, reset the file list box path
   File1.Path = Dir1.Path
End Sub
```

File List Boxes and Their Common Properties

Property	Default	Comments
Archive	True	Displays all files with archive bit on.
Name	File1	Name used in code. Use fil as a prefix.
FileName		Sets or returns the selected file from the list portion of a file list box. Accessible only at run time. This is the default property for this control.
Hidden	False	Does not display hidden files.
List		Accessible only at run time; returns the items contained in a list box.
ListCount		Accessible only at run time; returns the total number of items in the list box.
ListIndex		Accessible only at run time; returns the index of the currently selected item.
Normal	True	Displays all files with archive bit set on.
Path		Accessible only at run time.
Pattern	*.*	Display all files.
ReadOnly	True	
System	False	

Accessing the FileName Property

Example

```
Sub File1_DblClick ()
   Label1.Caption = File1.FileName
End Sub
```

Events

Click For all three of the basic controls, a Click event indicates that the user has made a selection. That is, the user is selecting or changing drives, directories, subdirectories, or filenames.

Double-click The only time that a double click is routinely used in a file browser is when the user uses the File Name combo box. In this case, the double click indicates that the user has located a specific file of interest and wants the application that uses the file browser to actually open that file.

Change The real workhorses of a file browser are the Change events. Code that you add to the event procedure template keeps the three controls synchronized whenever the user selects a new item in one of the list boxes.

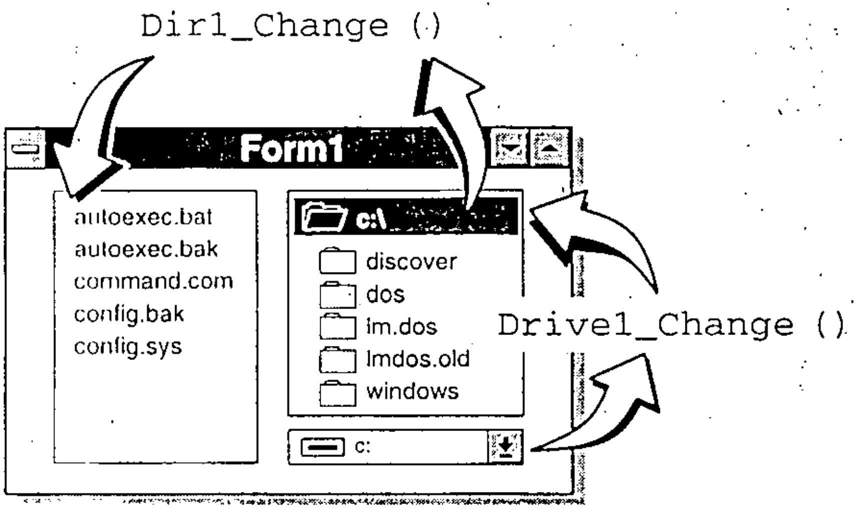
PathChange This event occurs when the selected path has been changed by setting the FileName or Path property from code.

PatternChange This event occurs when the pattern has been changed by setting the FileName or Pattern property from code.

∑ Change Events

- Change Event
- PathChange Event

The Change Event



Changing the Path for Differing List Boxes

Change Events Change events indicate that the contents of a control have changed. Change events are associated with a wide variety of controls-including combo boxes, horizontal and vertical scroll bars, labels, and picture boxes. For our purposes, this module focuses on the Change events for directory and drive list boxes only.

Directory List Box The Change event is invoked when the user clicks a new directory or when the Path property is changed from the code. It indicates that the contents of the control have been changed.

Example

```
Sub Dir1_Change ()
   ' When directory is changed, reset the file path
   File1.Path = Dir1.Path
End Sub
```

Drive List Box The Change event is invoked when the user selects a new drive or when the Drive property is changed from the code. It resets the path for the directory list box.

Example

```
Sub Drive1_Change ()
   ' When drive is changed, reset the directory's path
   Dir1.Path = Drive1.Drive
End Sub
```

Walk Through—Drawing and Coding a File Browser

∑ **To draw and code the file browser**

1. From the Walk Throughs program group, start Browser.

   When this application starts, you will see a simplified form of the file browser used in the Employee Database. This form has file, directory, and drive list boxes only.

   Click a new directory and notice the file list is updated.

   The purpose of the application is to demonstrate the Path and Drive properties and the Change events that are used to synchronize controls.

2. Close Browser.
3. Start Visual Basic.
4. From the File, choose New Project.
5. For Form1, set the following properties in the Properties window:

Height	3930 (approximately)
Width	4290 (approximately)

6. Double-click the file list box tool in the Toolbox.
7. Set the following properties:

Height	2760 (approximately)
Width	1455 (approximately)

8. Double-click the directory list box tool in the Toolbox.
9. Set the following properties:

Height	2175 (approximately)
Width	1455 (approximately)

10. Double-click the drive list box tool in the Toolbox.
11. Set the following properties:

Width	1455 (approximately)

    The first step of implementing a file browser is to add the Change event code that synchronizes changes in the state of the drive and directory list boxes. To do this, you need to:

12. Double-click the drive list box.

    This displays the Code window with the Drive1_Change event procedure in it

13. Add the following code:

```
Dir1.Path = Drive1.Drive
```

    This code resets the Path property of the directory list box to that of the Drive property of the drive list box.

    Now the directory list box will be kept current with changes in the drive list box. But what about the file list box?

14. From the Object list, select Dir1.

    The Code window with the Dir1_Change event template is displayed.

15. Add the following code:

```
File1.Path = Dir1.Path
```

16. Click Form1 to display the interface.

    The code added above resets the Path property of the file list box to that of the Path property of the directory list box.

    The characteristics of the Path property of a directory list box vary depending on whether the user's current working directory is the root directory.

    If the user selects the root directory, the path will be "\\"—that is, a path (in this case the root), which is a backslash. If the user selects any other directory, the path will end in the current working directory, with no backslash at the end.

    This is important when a program needs to build a fully qualified filename. The code must check to see if a backslash must be added to the current value of the Path property before concatenating it to a filename.

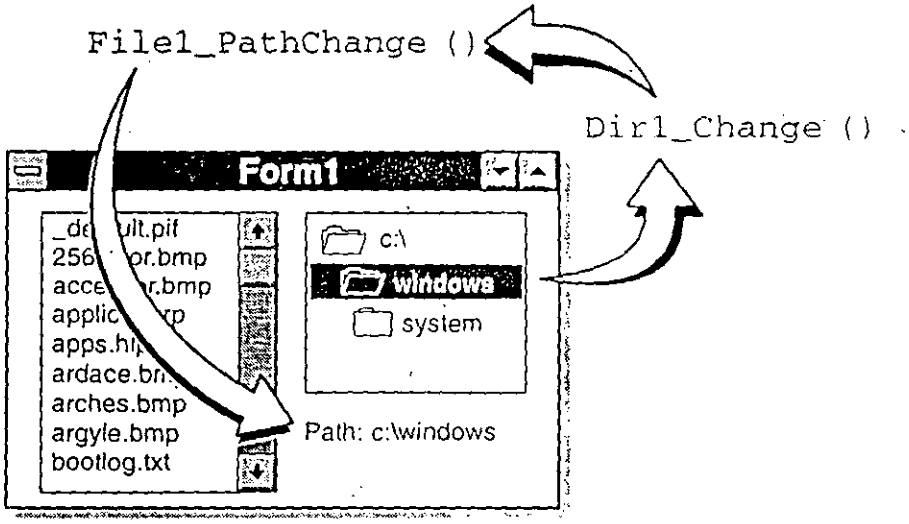
    The code to handle this requires a knowledge of conditional processing. To see how this is done, inspect the cmd OK_click procedure in the AddPhoto.FRM code in \SOLUTION\PRINTING.

17. From the Run menu, choose Start.

    Test to see if your code works.

18. From the File menu, choose Save As.
19. Save the file as BROWSER.FRM in \WALKTHRU\BROWSER.
20. Save the project as BROWSER.MAK in \WALKTHRU\BROWSER.

PathChange



PathChange

The event occurs when the selected path changes by setting either the FileName or Path property from code.

These two examples are from Visual Basic Help.

Examples

```
Sub Dir1_Change ()
   File1.Path = Dir1.Path
End Sub

'PathChange Event Example
Sub File1_PathChange ()
   Label1.Caption = "Path: " + Dir1.Path
End Sub
```

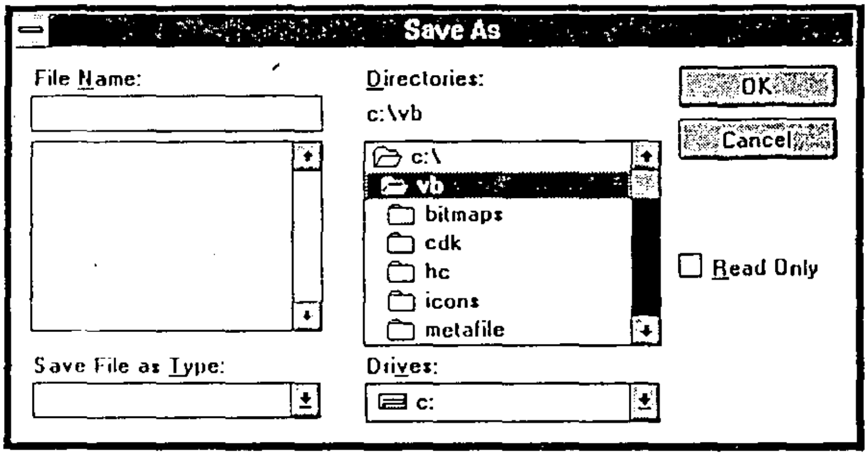
Notice in this example that the PathChange event is generated by code in a Change event in the directory list box control to synchronize the value of the File1.Path property to the new value for Dir1.Path.

For further examples using the PathChange event, see the following.

Further Examples

Application	Form	Procedure
IconWorks	VIEWICON.FRM	File_FileList_PathChange

The Common Dialog Control



The Common Dialog Control

The common dialog control is a versatile tool provided as a standard part of the Professional Edition of Visual Basic. The most common use for it is as a file browser, but as you will see, it has many other possible uses.

Example

```
Sub mnuFileOpen_Click ()
   CMDialog1.Action = 2
End Sub
```

Below is a list of the Action property settings and their associated dialog box type.

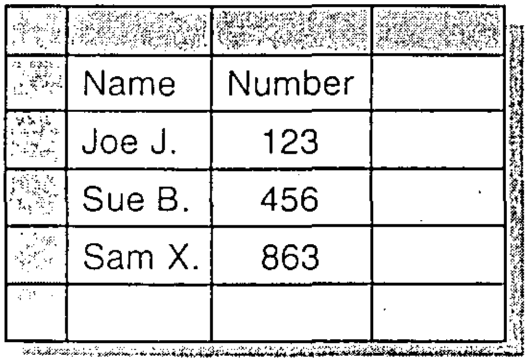
Action setting	Type of dialog box
0	No action
1	File open dialog box
2	File save dialog box
3	Color dialog box
4	Choose font dialog box
5	Printer dialog box
6	Invoke WINHELP.EXE

∑ More Controls

- Grid Control
- 3-D Panel Controls and Group Push Button

This final portion of the module deals, briefly, with each of the remaining tools in the Toolbox.

Using the Grid Control to Display Output



	Name	Number	
	Joe J.	123	
	Sue B.	456	
	Sam X.	863	

```
Grid1.Col = 1
Grid1.Row =1
Grid1.Text = "Name"
```

The Grid control allows you to display output on a grid.

The Cols and Rows properties set the total number of columns and rows you want in the grid. These properties are available at design time or at run time.

The Col and Row properties set or return the current cell in the grid. These properties are available at run time only. The Text property sets or returns the text in the current cell.

```
Sub cmdAddText_Click ()
   'Set the current cell; remember that the first row and
   'col are 0
   Grid1.Col = 1
   Grid1.Row = 1
   Grid1.Text = "Name"
End Sub
```

The Clip property sets or returns the contents of the grid's selected region—for example:

```
Grid1.Clip = "ABC"
```

You can include tabs and carriage returns in the string expression to indicate new columns and new rows, respectively.

The following example places text in a two-by-two range of cells.

Example

```
Sub cmdClipText_Click ()
    Grid1.SelStartCol = 1
    Grid1.SelEndCol = 2
    Grid1.SelStartRow = 2
    Grid1.SelEndRow = 3
    tb$ = Chr$(9)                   'Tab character
    cr$ = Chr$(13)                  'Carriage return
    d$ = "Joe J" + tb$ + "123" + cr$ + "Sue P" + tb$ + "456"
    Grid1.Clip = d$
End Sub
```

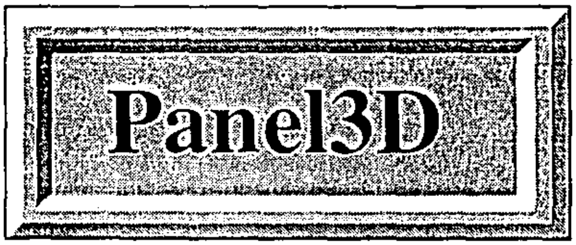
The grid control is used to display output. However, you can simulate input on a grid with code. The following example updates a grid cell when the user types into a text box:

```
Sub Text1_Change ()
   Grid1.Text = Text1.Text
End Sub
```

Some of the key properties that you will need in order to use this control are as follows.

Property	Default	Comments
ColAlignment	0 - Left Justify	Sets or returns the alignment of data in a column
Cols, Rows		Sets or returns the total number of rows or columns in the grid
ColWidth		Sets or returns the width (in twips) of a specified column
FixedCols, FixedRows		Sets or returns the total number of fixed columns or rows for a grid
GridLines	– 1 - True	Determines whether lines between cells are displayed
Picture		Sets or returns a graphic for the current cell

3-D Panel



This control has a number of possible uses. It has, however, two primary uses—the first is to provide greater three-dimensional (3-D) quality to another control or group of controls. For example, you can place other controls on the panel; it can be used in place of a frame. You might also use this control as a background for the entire form. The three-dimensional panel, because it has FloodPercent and FloodType properties, can be used as a status or progress indicator.

Some of the key properties that you will need in order to use this control are.

Property	Default	Comments
Alignment	0 - Left Justified Top	Caption alignment for this control offers nine possible choices.
BevelInner	0 - None	Sets the style of the inner bevel of the panel: none, inset, or raised.
BevelOuter	2 - Raised	Sets the style of the outer bevel of the panel
BevelWidth	1	Sets or returns the width of both the outer and inner bevels of the panel.
FloodColor		Sets or returns the color used to paint the area inside the panel's inner bevel when used as a status indicator.
FloodPercent		Sets or returns the percentage of the painted area inside the inner bevel when used as a status indicator.
FloodType	0 - None	Determines whether and how the panel is used as a status indicator.

Walk Through—Creating a Percent Meter

∑ To set the Percent Meter at work

1. From the Walk Through program group, choose Percent Meter.

   The purpose of this walk through is to demonstrate the status reporting capability of the three-dimensional panel control.

2. On the Percent Meter application, choose Start.

   In a few seconds the panel will progressively change color, indicating along the way the percentage completion of the process.

3. Open the Control menu on the Percent Meter application, and choose Close.

∑ To create the Percent Meter application

1. Start Visual Basic.
2. From the File menu, choose New Project.
3. Set the properties for the form as follows.

Property	Setting
Caption	Percent Meter
Height	4425 (approximately)
MaxButton	False
MinButton	False
Width	2535 (approximately)

4. From the toolbar, double-click the 3-D Panel tool and set its properties as follows.

Property	Setting
BevelInner	1 - Inset
BevelOuter	2 - Raised
BevelWidth	3
BorderWidth	2
Caption	
FloodColor	Yellow
FloodType	4 - Bottom to Top
Height	2655 (approximately)
Left	240 (approximately)
Outline	True
Top	240 (approximately)
Width	1935

5. From the toolbar, double-click the Command button and set its properties as follows.

Property	Setting
Caption	Start
FontSize	13.5
Height	495 (approximately)
Left	480 (approximately)
Top	3240 (approximately)

6. Double-click the Command button in design mode in order to open the Code window.

   Visual Basic displays a Code window with the Command1_Click event procedure template in it.

7. Place the following lines of code between the two parts of the template:

```
Panel3d1.FloodPercent = 0
For i% = 1 To 100
    Panel3d1.FloodPercent = i% * 1
Next i%
```

8. From the Run menu, choose Start.

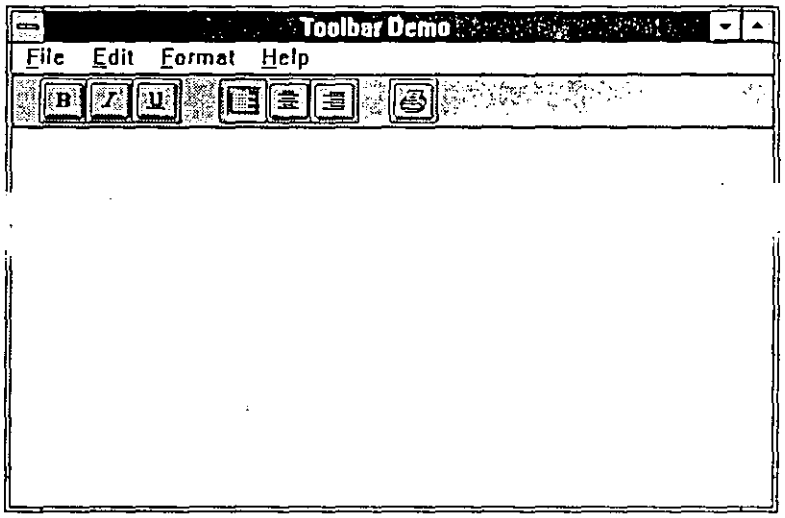
   Test your application.

9. From the Run menu, choose End.

10. Save your form and .MAK file with appropriate names in \WALKTHRU\SAMPLES.

11. Close Visual Basic.

Group Push Buttons



Group push buttons work like combination command buttons and option buttons. They are like command buttons because when the user clicks the button, something happens—a file is opened or saved, or text is right-justified. Group push buttons, however, are also like option buttons because, with the properties set properly, selecting one option can automatically remove the selection from another option.

Some of the key properties that you will need in order to use this control are as follows.

Property	Default	Comments
GroupAllowAllUp	–1 - True	Determines whether all buttons in a logical [illegible]
GroupNumber	1	Sets or returns the group number for a given group push button. This property is used to create logical groups of buttons.
Outline		Sets or returns whether the button has a border around it.
PictureDisabled	None	Specifies the bitmap to display on the 3D group push button when it is disabled.
PictureDn	None	Specifies the bitmap to use when the button is depressed.
PictureDnChange	0	Determines how the PictureUp bitmap is to be used to create the PictureDn bitmap: 2 merely inverts the PictureUp bitmap.
PictureUp	None	Specifies the bitmap to use when the button is up.

Walk Through—Creating a Toolbar

∑ To see the interface you will be creating

1. From the Walk Throughs program group, choose Toolbar.

   This little application doesn't do very much. What it does show is that the user can select any or all of the first three options, whereas they can only select one of the second three. More to the point, it is an example of how you would go about drawing a toolbar like the one found in Microsoft Word for Windows or Microsoft Excel.

2. From the Control menu on the Toolbar form, choose Close.

∑ To create the toolbar

1. Open Visual Basic.
2. From the File menu, choose Open Project.

   Locate TOOLBAR.MAK. It should be located in \WALKTHRU\SAMPLES. This form already contains the basic form—TOOLBAR.FRM with all of its properties already set. In order to add the toolbar, you need to complete the following steps:

3. Place seven group push buttons on the toolbar panel at the top of the form.
4. Use the mouse pointer and the control key to select all seven of the group push buttons, and then set the following properties for all of them.

Property	Setting
Height	420
Top	30
Width	450

5. Place three of the push buttons side by side on the left end of the panel.

   These will become the text formatting buttons for Bold, Italic, and Underline.

6. Select the first button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	BLD-UP.BMP located in \VB\BITMAPS\TOOLBAR3
PictureDn	BLD-DWN.BMP

These changes create the first button on the toolbar and make use of two separate bitmaps to indicate the status of the button to the user.

7. Select the second button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	ITL-UP.BMP
PictureDn	ITL-DWN.BMP

8. Select the last button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	ULIN-UP.BMP
PictureDn	ULIN-DWN.BMP

Notice that all three of these buttons have the same number—0. A group number of 0 means that they are *not* part of a logical group, and therefore changing the state of one will not change the state of others. After all, your user may want to have bold and italic and underlining. Clusters of controls with any other group number will act just like option buttons—only one will be in force at any one time.

For purposes of contrast, we will take a slightly different approach on the second group of push buttons. Here you are giving the user the option of formatting text as either left-, centered-, or right-justified. In this case, you will want these buttons to work as a group because any one bit of text can have only one of these three states.

9. Create the second group of buttons side by side toward the middle of the panel.

In all likelihood, you will not be able to keep the buttons in perfect top-to-bottom alignment. Don't let that bother you. You can select the entire group at the end of this walk through and set the Top property for all of them in one step, as you did above.

10. Select the first button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	LFT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

Notice what we have done here? We have created the PictureDown bitmap by inverting the image using the property PictureUpChange.

11. Select the second button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	CNT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

12. Select the third button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	RT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

What this example emphasizes is that the property GroupNumber is your way of telling Visual Basic how you want individual sets of buttons to be grouped.

13. As a little extra challenge, you set the properties for the last control. It's supposed to represent a printer, so you will need to use the PRT-UP.BMP.
14. Save your work in \WALKTHRU\SAMPLES.

∑ To Code The Toolbar

If you want to implement the toolbar so it functions properly, you can add the following code.

1. Double-click the bold button to open the code window for GroupPush3D1_Click event and add the following code:

```
lblMessage.FontBold = Value
```

2. Double-click the italic button to open the code window for GroupPush3D2_Click event and add the following code:

```
lblMessage.FontItalic = Value
```

3. Double-click the underline button to open the code window for GroupPush3D3_Click event and add the following code:

```
lblMessage.FontUnderline = Value
```

4. Double-click the left-justify button to open the code window for GroupPush3D4_Click event and add the following code:

```
lblMessage.Alignment = 0
```

5. Double-click the center button to open the code window for GroupPush3D5_Click event and add the following code:

```
lblMessage.Alignment = 2
```

6. Double-click the right-justify button to open the code window for GroupPush3D6_Click event and add the following code:

```
lblMessage.Alignment = 1
```

7. Save your work in \WALKTHRU\SAMPLES.

Summary

- File Browsing
- Grid Control
- 3-D Panel Controls and Group Push Button

Objectives

In this module you learned to:

- Design and develop a form that lets users browse drives, directories, and files.
- Use the Change and PathChange events.
- Use the grid control.
- Use 3-D panel controls and the group push button.

Module 8: Using Visual Basic Data Types

∑ Overview

- Key Terms
- Using Variables and Constants
- Scope
- Additional Visual Basic Data Types

Overview

The overall purpose of this module is to introduce already knowledgeable programmers to the data types of Visual Basic and their scoping rules. This module is designed as the first of a series of modules on the implementation of Basic within Visual Basic.

Prerequisites

Prior to starting this module, you should already be familiar with:

- Data types and scoping rules within some other procedural language
- The general Visual Basic programming system

Overall Objective

At the end of this module, you will be able to correctly and efficiently use almost all of the Visual Basic data types.

Learning Objectives

At the end of this module, you will be able to:

- List and describe the seven variable data types in Visual Basic.
- Correctly identify the six type declaration character symbols
- Use the contents of CONSTANT.TXT.
- Distinguish among the various levels of scope in Visual Basic.
- Explain the rules for using each level of scope.
- Declare and use a user-defined data type.
- Declare and use simple arrays.

Key Terms

- Variables
- Constants
- Procedures
- Statements
- Scope
- Module

Variables Changeable values that the program manipulates

Constants Unchanging values that the program manipulates

Procedures Activities that the program performs

Statements Subactivities within procedures

Scope (Accessibility) Which part of a program can access specific data or procedures

Module File containing code and data not attached to a form

∑ Using Variables and Constants

- Data Types of Variables
- Data Types of Constants

Data Types of Variables

▪ Integer	%
▪ Long	&
▪ Single	!
▪ Double	#
▪ Currency	@
▪ String	$
▪ Variant	(none)

There are seven fundamental data types of variables that you can use in Visual Basic.

Type name	Description	Type-declaration character	Range
Integer	Two-byte integer	%	−32,768 to 32,767
Long	Four-byte integer	&	−2,147,483,648 to 2,147,483,647
Single	Four-byte floating-point number	!	−3.40E+38 to 3.40E+38
Double	Eight-byte floating-point number	#	−1.79D+308 to 1.79D+308
Currency	Eight-byte number with a fixed decimal point	@	−9.22E+14 to 9.22E+14
String	String of characters	$	0 to 65,500 characters (approximately)
Variant (Default)	Date/time, floating-point number, string	(none)	Date values: January 1, 0000, to December 31, 9999; numeric values same as double; string values same as string

Declaring Variables

In polite society you always formally introduce strangers to each other the first time they meet. Visual Basic is no exception. You should introduce your variable to Visual Basic by declaring the name of your variable before you use it. You do this in one of two ways: by using the **Dim** statement, or by using one of the two keywords—**Global** and **Static**. As you will see in a couple of pages, not declaring variables can be risky business in some situations because the default data type in Visual Basic is **Variant**.

You have two choices on how to explicitly declare variable data types.

Using As	Using the type-declaration character
`Dim I As Integer`	`Dim I%`
`Dim Amt As Double`	`Dim Amt#`
`Dim YourName As String`	`Dim YourName$`
`Dim BillsPaid As Currency`	`Dim BillsPaid@`

In order to make compound variable declarations, you merely state:

```
Dim X as Integer, Y as Long
```

Caution If you declare three separate variables with the following statement, you will get K$ as a string and I and J as variants, not as strings.

```
Dim I, J, K$
```

Explicit Declarations

To require explicit declaration of variables in all of your code, place the following statement in the General Declarations section of any form or module.

```
Option Explicit
```

If you want to require that all variables be explicitly declared in all of your projects, from the Options menu choose Environment and set the Require Variable Declaration option to YES.

Variable-Length Strings

Sometimes you may need a variable that can hold strings of different lengths at different times. This is called a variable-length string variable. You declare it like this:

```
Dim Message As String
```

Fixed-Length Strings

At other times you know that the strings that will be assigned to a certain variable will never have more than a certain number of characters. In this case, you can declare a *fixed-length* variable like this:

```
Dim FixedLengthString As String * 50
```

In this example, you are declaring a string variable with the name FixedLengthString, and you are telling the application that the variable may contain up to 50 characters.

You can take this one step further. For example, set the length using a **Global** constant with the name FixedLength, set the value of the constant to 50, and then point to that constant name using the following syntax:

```
Global Const FIXEDLENGTH = 50
Global FixedString As String * FIXEDLENGTH
```

Rules When Working with Variables

1. Variable names can be up to 40 characters long.
2. Names can include only letters, numbers, and underscores.
3. The first character in the name must be a letter.
4. You cannot use Visual Basic reserved words. See the list of programming topics in Visual Basic Help for a partial list of reserved words.

As the chart above shows, the **Variant** data type is the default. The **Variant** data type can store numeric, string, or date/time information. You don't need to convert between these kinds of data when assigning them to a variant variable: Visual Basic automatically performs any necessary type conversions for you.

Rules for Using Variant Variables

1. If you perform arithmetic operations or functions on a variant, it must contain a number. If you want to test a variant to see if it contains numeric data, use the **IsNumeric** function.

2. Normally, when you concatenate two strings you would use the plus (+) sign. To avoid ambiguity with variant data, use the ampersand (&) to indicate concatenation.

   Note When concatenating, be careful to leave a space between a variable name and the ampersand. If you don't, Visual Basic assumes that you are typecasting the variable into a **Long** data type.

3. When passing a **Variant** variable as an argument, check the procedure parameters. If the corresponding argument is an explicit data type, you must pass the **Variant** variable with parentheses around it in order to pass it by value.

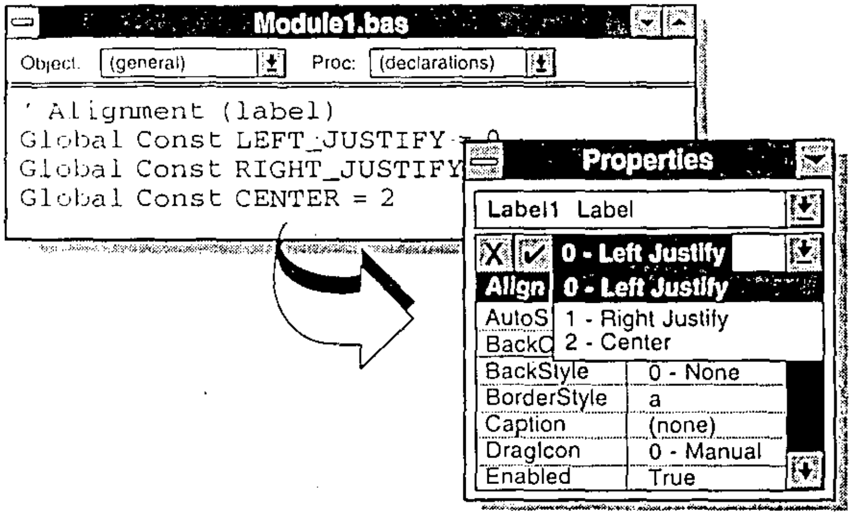
```
Dim V As Variant
V= "Testing"
Debug.Print PrintString((V))        'Extra parentheses to pass by value

Sub PrintString(S As String)
     'Do something
End Sub
```

4. To determine the internal representation for a variant variable, use the **VarType** function. See documentation for specific return values.

5. A variant variable has the **Empty** value before it is assigned a value. The **Empty** value is a special value, different from a zero, a zero-length string, or a NULL value. To test your **Variant** variable for an **Empty** value, use the **IsEmpty** function. To reassign a **Variant** variable back to the **Empty** value, you must assign another empty **Variant** variable to it.

6. To test for a **Variant** variable containing NULL, use the **IsNULL** function.

Note For a listing of the string conversion functions for the **Variant** data type, see the appropriate table at the end of the Visual Basic code module.



```
' Alignment (label)
Global Const LEFT_JUSTIFY = 0
Global Const RIGHT_JUSTIFY
Global Const CENTER = 2
```

What Are Constants?

Constants are just that, entities within your program whose value you need, but which will not be changed during the running of the program. For example, if your program needed to work with circles, you would probably need a constant like PI = 3.1459.

Examples of typical constants can be found in CONSTANT.TXT. If you look at this file, you will find several constant values declared that are familiar to you from the overview of properties for controls.

Example

```
'Border Style
Const NONE = 0
Const FIXED_SINGLE = 1
Const SIZABLE = 2
Const FIXED_DOUBLE = 3
```

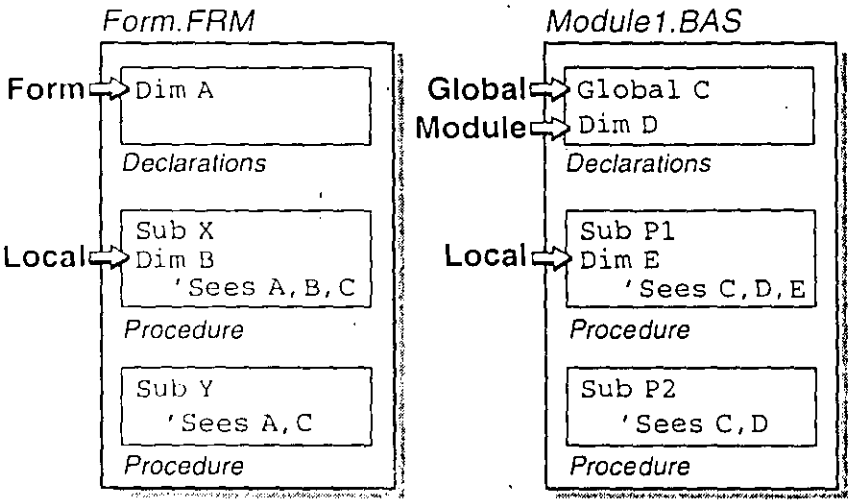
How Are They Declared?

Visual Basic contains a CONSTANT.TXT file, which lists all the common constants. You can copy and then modify this file to meet the application's specific needs. You can then load CONSTANT.TXT into the General Declarations section of a module (.BAS) to be used by the application. Following is a partial listing of the general categories of constants defined in CONSTANT.TXT.

Σ Scope

- Scope of Data
- Declaring and Using Local Variables
- Declaring and Using Form-Level and Module-Level Variables
- Declaring and Using Global Variables

Scope of Data



What Is Scope of Data?

The scope of a data variable or constant is its level of visibility within an application. Scope of data falls into three levels of visibility: local variables, form- and module-level variables, and global variables.

Scope of Data—An Analogy

Scope of data basically deals with who can use what. The easiest way to think of this is by analogy.

Say you had a refrigerator/freezer full of all kinds of goodies—candy, soft drinks, ice cream. Almost everyone likes these goodies and would like to get to them. Your job as the manager of these resources is to make sure that the right people get to them and the wrong people don't.

If you put these goodies out where everyone can get to them, they will be used, right? If that is what you want, then fine, make them *globally* available. In Visual Basic terminology, declare and define variables that all functions will need access to using the **Global** keyword in the General Declarations section of a module (.BAS), not a form.

For those goodies that you only want some people to get to, for example, just the members of your work group, then put the refrigerator/freezer where only they can get to it—for example, in your work group area. In Visual Basic, declare and define these variables using the keyword **Dim** in the General Declarations section of either the form or the module. You'll see an example of how this is done in just a few minutes.

Finally, for those goodies to which only you should have access, put the refrigerator in your office. That is, declare and define those variables locally — inside a procedure. Again, you'll see examples of this is just a minute.

Variable Declaration Keywords and Their Scope

The table below details how the different variable declaration keywords are used.

Scope	Declaration
Local	**Dim**, **Static**, or **ReDim** (within a procedure)
Module	**Dim** (in General Declarations section of a form or a code module).
Global	**Global** (in General Declarations section of a module)

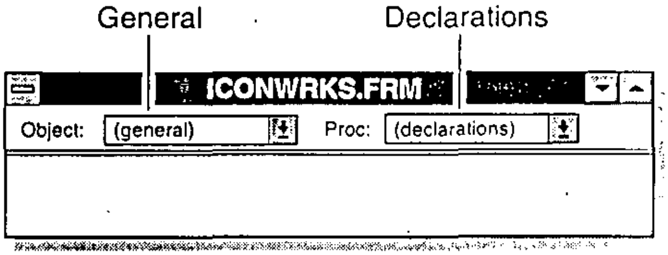
Σ **A simple example of data declarations can be found this way**

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Go to the C:\VB\SAMPLES\CALC subdirectory.
4. Double-click CALC.MAK.
5. Click CALC.FRM in the Project window.
6. Click View Code in the Project window.

Visual Basic displays the form-level variable declarations for the calculator application. Notice the declaration of seven variables and the **Option Explicit** statement.

Note When you are writing code and you want to create module- or form-level variables, make sure that you are really in the general declarations section of the code before you make changes or additions.

Always be sure that the top of your form window has these two options selected.



Object: (general) Proc: (declarations)

∑ **For a more complex listing of global declarations**

1. From the File menu, choose Open Project.
2. Go to the C:\VB\SAMPLES\ICONWRKS subdirectory.
3. Double-click ICONWRKS.MAK.
4. Click ICONWRKS.GBL in the Project window.
5. Click View Code in the Project window.

Scroll down to the General Declarations section of ICONWRKS.GBL and inspect the various values declared and defined.

Rules for Using Local Variables

1. A local variable is recognized and declared only within the procedure in which it appears using the keyword **Dim**. The variable is re-initialized to zero or a NULL string when the procedure begins, unless the local variable is declared as **Static**. A local variable is a good choice for any kind of temporary calculation.
2. If you use a variable without declaring it, then Visual Basic assumes it is local and assumes **Variant** as the data type. However, this technique may waste storage space and is not as reliable as declaring the variable in the appropriate procedure. It is also not as efficient.

Rules for Using Form- and Module-Level Variables

1. A form-level variable is declared in the General Declarations section of the form, using the keyword **Dim**. A form-level variable is available to any procedure on that form and only that form.
2. A module-level variable is declared in the General Declarations section of the module, using the keyword **Dim**. A module-level variable is available to any procedure in that module and only that module.

Rules for Using Global Variables

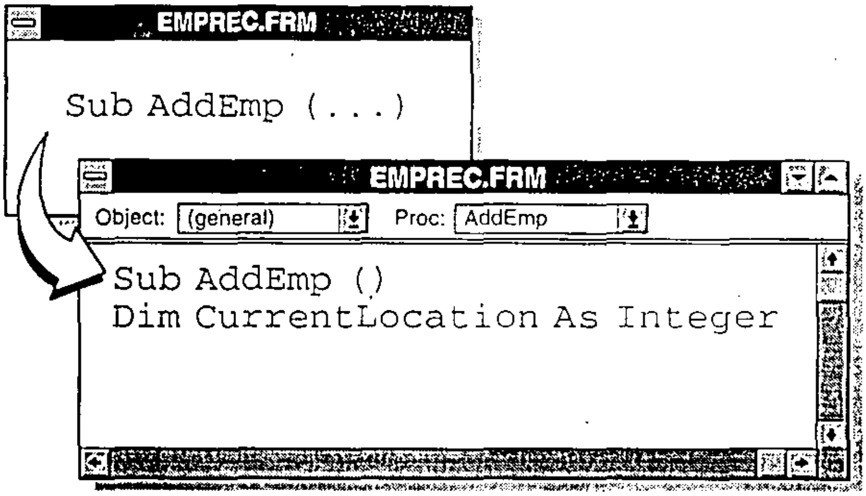
1. A global variable or constant is shared throughout an application. Multiple forms and modules can use a global variable. They are declared in the General Declarations section of a module using the **Global** keyword for variables and **Global Const** for constants.
2. Global variables are persistent; they retain their value throughout the entire application.

Style Guidelines

Good programming practices when using variables and constants include:

- Use form-level and module-level constants and variables as opposed to global-level if possible.
- Declare the variable or constant with the **Dim** statement rather than accepting Basic automatic variables.
- Use **Integer** declarations instead of default **Variant** declarations whenever possible. **Integer** declarations take less space, avoid rounding errors, and consume less CPU time than **Variant** declarations.

Declaring and Using Local Variables



```
Sub AddEmp (. . .)
```

```
Sub AddEmp ()
Dim CurrentLocation As Integer
```

Local variables reset to value zero (0) or an empty string when the local procedure begins.

To declare a local variable, place the Dim data declaration statement inside the local procedure.

Example

```
Sub Form_Click ()
     'The local variable below disappears upon procedure exit
     'and is reinitialized to zero when the procedure is re-entered
     Dim EmployeeNumber As Integer

     EmployeeNumber = Val(Text1.Text)
     Print "The Employee Number is:  " + Str$(EmployeeNumber)
End Sub
```

In the example above, EmployeeNumber is declared as an **Integer**. How would the code be different if you handled it as a **Variant**?

Example

```
Sub Form_Click()
     Dim EmployeeNumber

     EmployeeNumber = Text1.Text
     Print "The Employee Number is: " & EmployeeNumber
End Sub
```

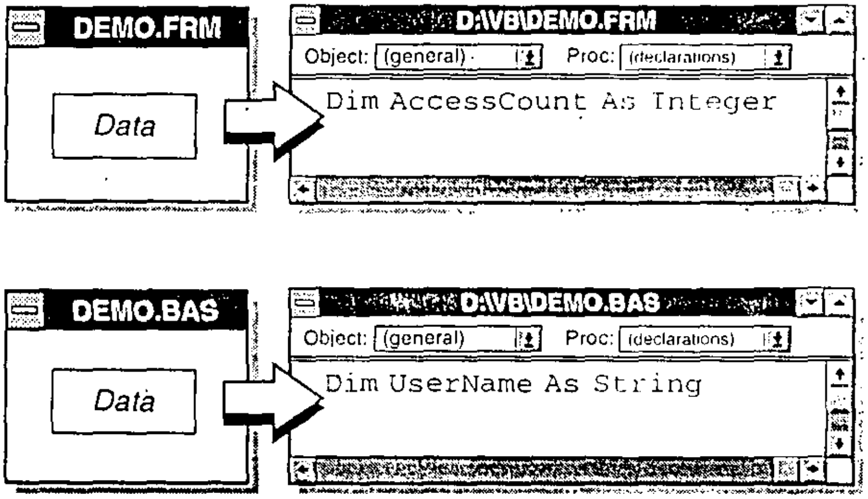
Note Notice the use of the ampersand to concatenate the literal **String** and the **Variant** in the Print statement above?

- If you want a local variable to be persistent, declare it as **Static**. **Static** local variables will retain their data value when the procedure performs a return.

Example

```
Sub StaticDemo ()
    'The static variable below will retain its value across
    ' calls and is used to keep track of how many times
    ' StaticDemo is called.
    Static RunCount As Integer  ' Create a persistent variable
    RunCount = RunCount + 1     ' Use it to keep run count.
    Msg$="The RunCount for the StaticDemo procedure is: "
    Msg$ = Msg$ + Str$(RunCount)
    Print Msg$                  ' Display message.
End Sub
```

Declaring and Using Form-Level and Module-Level Variables



```
Dim AccessCount As Integer
```

```
Dim UserName As String
```

Use form-level variables or module-level variables when sharing data only within a form or a module. The data remains persistent within the form or module but is not accessible to procedures in other forms or modules.

To declare a form-level or module-level variable, place the **Dim** statement in the General Declarations section of the corresponding form or module.

The declaration below appears in the General Declarations section of the module DEMO.BAS:

```
Dim UserName As String
```

When the procedure below is called, it can access this module-level variable. For example, suppose that a text box is supplied for the user to type his or her name:

```
      .
      .
      .
UserName = txtUserName.Text
```

Then the appropriate value for UserName will be printed when the **Sub** procedure WelcomeDemo is called:

Example

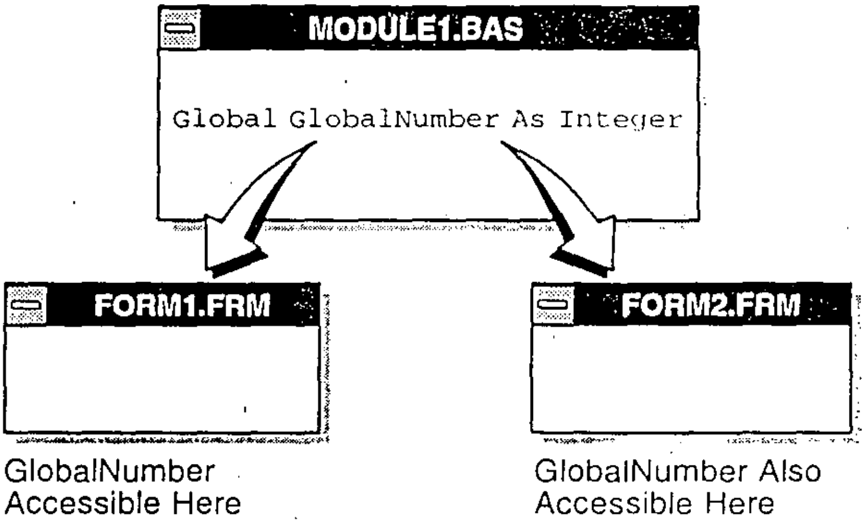
```
Sub WelcomeDemo ()
      .
      .
      .
' Variable declared in module's Declarations section can be
' referenced here
    Print "Welcome " + UserName
End Sub
```

The module-level variable is also available to all other procedures defined in DEMO.BAS.

Example

```
Sub VerifyUserName ()
      .
      .
      .
    Msg$ = "UserName = "
 ' Variable declared in module's Declarations section can
 ' also be referenced here
    MsgBox Msg$ + UserName
End Sub
```

Declaring and Using Global Variables



MODULE1.BAS
FORM1.FRM
FORM2.FRM
GlobalNumber Accessible Here
GlobalNumber Also Accessible Here

```
Global GlobalNumber As Integer
```

Use global variables and constants when sharing data throughout an application. The data remains persistent and accessible from all forms and modules within the application.

To declare a global variable, place the Global statement in the General Declarations section of a module—that is, a .BAS file:

```
Global GlobalNumber As Integer
```

The variable GlobalNumber can now be referenced throughout the application.

Example 1

```
' Code for Form1
Sub Form_Click ()
' Global variables can be referenced here
    GlobalNumber = GlobalNumber + 1
End Sub
```

Example 2

```
' Code for Form2
Sub Form_Click ()
' Global variables can also be referenced here
    Print = "GlobalNumber " + Str$(GlobalNumber)
End Sub
```

Load any constants you want declared for an entire program into the General Declarations section of a module.

Walk Through—Scope

∑ **To demonstrate the concepts of scope of data in Visual Basic**

1. From the Walk Throughs program group, start Scope.

   When the application starts, you see two forms. Form 1, on the left, contains a Command1 button. Form2, on the right, contains a Command1 button.

   The purpose of this walk through is to demonstrate the concepts of scope of data in Visual Basic.

2. Click Form1 (on the form, not the command button).

   The Form_Click event will display values of data items as their scope permits.

   The global and form-level data values are displayed. Note, however, that the Form2 module-level variable and the Form1.Command1_Click and Form2.Command1_Click local variable values are not displayed. Remember, you just requested a Form_Click event, so the Command1_Click variables are not yet in scope and Form2 variables are not in scope until Form2 has focus.

3. Click Form2 (on the form, not the command button).

   The Form_Click event will display values of data items as their scope permits.

   The global and form-level data values are displayed. Note, however, that the Form1 module-level variable and the Form1.Command1_Click and Form2.Command1_Click local variable values are not displayed either. Remember, you just requested a Form_Click event, so the Command2_Click variable is not yet in scope and Form1 variables are not in scope until Form1 has focus.

4. Click the Command1 button on Form1.

   The Command1_Click event will display values of data items as their scope permits.

   You have just generated a Form1.Command1_click event, and this places the Form1.Command1_Click variable in scope. Form1 still has focus, though, so Form2's variables are not accessible.

5. Click the Command1 button on Form2.

   The Command1_Click event will display values of data items as their scope permits.

   You have just generated a Form2.Command1_Click event, and this places the Form2.Command1_Click variable in scope. Form2 still has focus, though, so Form1's variables are not accessible.

6. Double-click the Control menus on both forms to close them.

Additional Visual Basic Data Types

- User-Defined Data Types
- Arrays

User-Defined Data Types

Earlier in this module standard Visual Basic data types were discussed. Visual Basic also allows you to create your own data types. User-defined data types, called records or structures in some other programming languages, must be declared in the General Declarations section of a module, using the Type statement.

Syntax

Type usertype

elementname **As** typename

[elementname **As** typename]

.

.

.

End Type

This feature allows you to create new variables that can be customized to fit the needs of your application. For example, you might want to define a data type to hold customer information as follows:

Example

```
' Place this declaration in a module
Type CustomerRecord
    CustNum As Long
    CustName As String * 38
    CustAddress1 As String * 38   ' Street Address
    CustAddress2 As String * 38   ' City, State
    CustZip As String * 10
End Type
```

You can then declare and use variables of this new type in your application wherever you need them. To refer to a particular element, use the syntax *variablename.elementname*.

Example

```
' Sub GetCustomerInfo ()
' Declare a variable of your user-defined type
Dim NewCustomer As CustomerRecord

' Use contents of various text boxes on a form to fill in values to
' the elements (fields) of your new CustomerRecord variable
NewCustomer.CustNum = Val(txtNumber.Text)
NewCustomer.CustAddress1 = txtAddr1.Text
NewCustomer.CustAddress2 = txtAddr2.Text
NewCustomer.CustZip = txtZip.Text
.
.
.
End Sub
```

Arrays

Visual Basic, like many other programming languages, allows you to create arrays. An array is a group of variables of the same data type that share a common name. Each separate element of the array is identified by a unique index number.

Example

```
' Declare an array
Dim ArrayName(UpperBounds) As DataType
Dim TestScores(23) As Single
```

Note By default, the first element in the array is referred to with the index 0 (zero). This creates an array of 24 (not 23) single-precision numbers in which you could store the scores for 24 students.

If you wanted to print this score out to the form, you would use the following:

Example

```
Print "The test score for the first student is: "
Print TestScores(0)
```

There is an alternate syntax for declaring arrays that allows you to specify the lower (beginning) and upper (ending) indexes, or bounds, of an array. You could declare an array to hold 24 test scores like this:

```
Dim TestScores(1 To 24) As Single
```

You would then refer to the first element in the array as TestScores(1).

If you didn't have arrays, you would have to create 24 separate variables, named something like Score1, Score2, Score3, and so on.

In the module on looping structures, you will discover that loops are a very useful tool for accessing elements in an array.

Note To change the default lower bound to 1, place an **Option Base** statement in the General Declarations section of the module, like this:

```
Option Base 1
```

Variations on Array Declaration Syntax

Depending upon where the array is declared and the special needs of your program, the syntax to declare an array may vary a bit. Here is a table that summarizes the rules.

Scope of variable declaration	Keyword to use
Application-wide	**Global** (Used only in the General Declarations section of a .BAS module.)
Form or module level	**Dim** (Used in the General Declarations section of a form or module.)
Within a procedure	**Static** (If the entire procedure has been declared **Static**, then you may use the word **Dim** to declare the array.)

Multidimensional Arrays

The array described in the last section, TestScores, is an example of a one-dimensional array. You could think of it as representing a single column (or row) of numbers. You can also declare an array of several dimensions. The maximum number of array dimensions allows in a **Dim** statement is 60. For example, to represent a table of numbers you can declare a two-dimensional array as follows:

```
Sub GetNumbers ()
   Static Table(4, 23) As Single
```

–Or–

```
   Static Table(1 To 5, 1 To 24) As Single
      .
      .
      .
End Sub
```

This would create an array of five rows of 24 test scores each and provide room to hold five scores for each of 24 students.

Dynamic Arrays

There are times when you want to use an array, but the size needed will change at run time. Visual Basic allows you to create dynamic, or variable-length arrays by doing the following:

Declare the array without declaring its size with either the **Dim** statement (for a form- or module-level array) or the **Global** statement (for a global array). To do this, just place an empty set of parentheses to the right of the array name:

```
' Place in General Declarations section of a form or module
Dim DynArray() As String * 25
```

–Or–

```
Global DynArray() As String * 25      ' Place in MODULE1.BAS
```

Here is another example:

```
Global DynIntArray() As Integer
```

Then inside a procedure, when you know the size you need for this array in some particular circumstance, redimension the array to the size you need:

Example

```
Sub UseArray ()
    :
    :
    ' Notice you can use a variable from your program to set the size
    ReDim DynArray( List1.ListCount)
    :
    :
End Sub
```

Important Each time you use the **ReDim** statement, all the values currently stored in the array are lost, and each element is reset to zero or a NULL string depending on the type of the elements in the array. If you want to preserve the values during re-dimensioning of the array, use the **Preserve** keyword.

```
ReDim Preserve MyArray(UBound (MyArray) + 10)
```

In this case, you are re-dimensioning an array called MyArray to be 10 elements larger while maintaining existing data.

Control Arrays

Visual Basic has a special type of array called a control array, which has special features and does not follow all the rules of standard arrays.

You will learn to use control arrays in the course that follows this one in the Microsoft University Visual Basic curriculum.

Summary

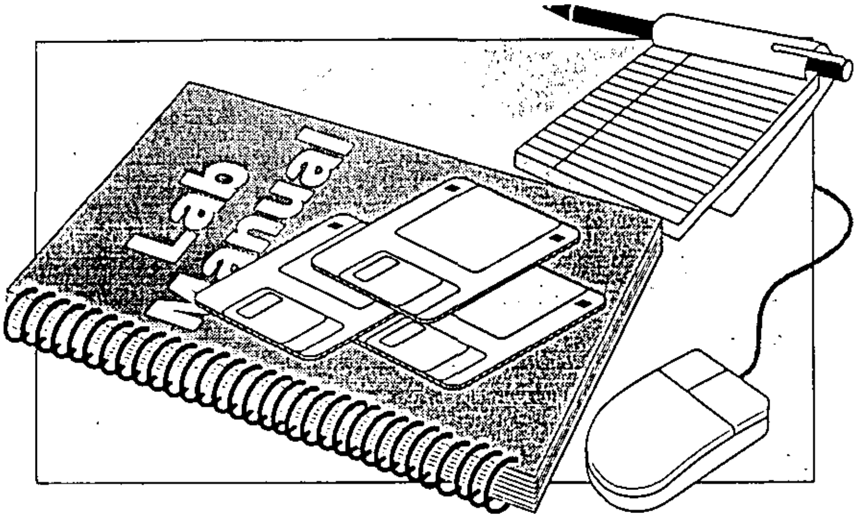
- Key Terms
- Using Variables and Constants
- Scope
- Additional Visual Basic Data Types

Objectives

In this module you learned to:

- List and describe the seven variable data types in Visual Basic.
- Correctly identify the six type declaration character symbols.
- Use the contents of CONSTANT.TXT.
- Distinguish among the various levels of scope in Visual Basic.
- Explain the rules for using each level of scope.
- Declare and use a user-defined data type.
- Declare and use simple arrays.

Lab Time



Go to the Using Constants and Variables portion of your lab manual.

Module 9: Writing Visual Basic Code

∑ Overview

- Visual Basic Procedures
- Scope of General Procedures
- Writing Code in Visual Basic
- String and Numeric Conversion Functions

Overview

This is the "Everything You Ever Wanted to Know About Writing Code...But Were Afraid to Ask" module. It gives you the background and procedures needed to begin writing the Visual Basic code that implements the interfaces.

Prerequisites

To successfully complete this module and its associated lab, you should have a detailed understanding of forms, controls, and properties as they are implemented in Visual Basic.

Knowledge of any block structured programming language is required.

Overall Objectives

There are two overall goals for this module:

1. To quickly review most of the fundamental procedures of Basic as it is used in the Visual Basic product
2. To show students the first steps required for completing the code for an application

Learning Objectives

At the end of this module, you will be able to:

- Define important characteristics of **Functions** and **Sub** procedures and write code that correctly uses both.
- Define and correctly code for appropriate scope of data.
- Write code that uses any of the common Basic data types.

Sample Code

The best way for you to learn how to write Visual Basic code is to read it. Sample code for a four-function calculator application can be found in the \VB\SAMPLES\CALC subdirectory.

∑ Visual Basic Procedures

- Sub Procedures
- Functions
- Arguments and Parameters
- Procedures

  Event Procedures

  General Procedures

  Methods

What Are Procedures?

A procedure is a block of Visual Basic statements that are called as a logical unit.

Two Types of Procedures — A First Pass

Sub Procedures

After the **Sub** procedure completes its work (executes its code), it returns to the procedure that called it.

Syntax

Sub *SubName* ()

statementblock

End Sub

Functions

A **Function** is similar to a **Sub** procedure. In addition, it has a data type just as a variable does. After it completes its work, it returns a value of that type to the procedure that called it.

Syntax

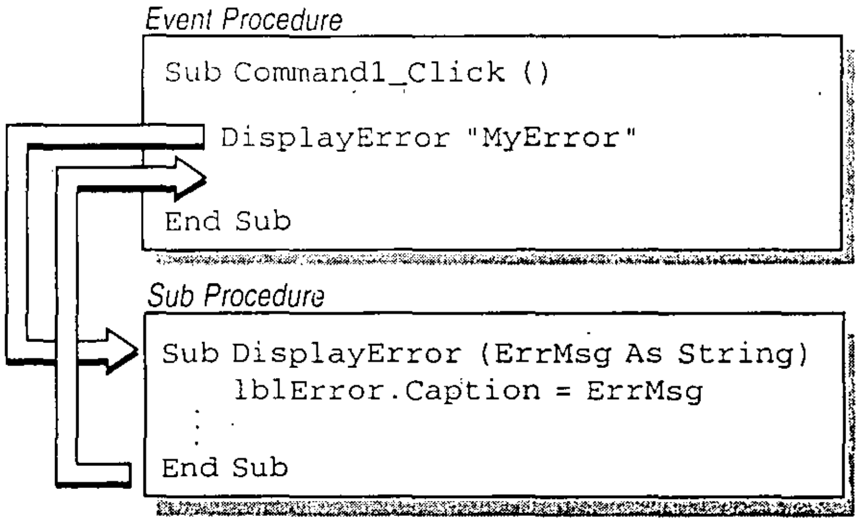
Function *FunctionName*() As *SomeDataType*

statementblock

FunctionName = *SomeValue*

End Function

Calling a Sub Procedure with Arguments



Event Procedure

```
Sub Command1_Click ()
    DisplayError "MyError"

End Sub
```

Sub Procedure

```
Sub DisplayError (ErrMsg As String)
    lblError.Caption = ErrMsg
  :
  :
End Sub
```

Arguments

Any procedure can be defined to receive data when it is called. A piece of data sent to a procedure is called an argument; and an argument is matched to its equivalent parameter entry in the procedure list. Each parameter is declared to be of a specific data type.

The following is a **Sub** procedure that receives two arguments:

Syntax

Sub *SomeSub* (*Param1* **As Integer**, *Param2* **As Single**)

statementblock

End Sub

When you call this procedure you must make sure to pass it the same number and type of arguments in the same order they appear in the **Sub** definition.

Syntax

SomeSub Argument1, *Argument2*

–Or–

Call *SomeSub* (*Argument1*, *Argument2*)

The following calls a **Function** procedure with arguments.

Syntax

Dim *Result* **As Integer**

Result = *SomeFunction* (*Argument1*, *Argument2*)

Function *SomeFunction* (*Param1* **As Integer**, *Param2* **As Single**)
∧ **As Integer**

StatementBlock

SomeFunction = *ReturnValue*

End Function

Note The argument names used when you call the procedure do not have to match the argument names used in the definition of the procedure as they do in the above example. The number, data type, and order of the arguments must match in the procedure definition and call.

An alternative notation for the function return value data type is as follows.

Syntax

Function *SomeFunction%* (*Param1* **As Integer**, *Param2* **As Single**)

StatementBlock

SomeFunction = *ReturnValue*

End Function

Note **Function** procedures with an explicit return value data type are more efficient than those with a **Variant** return value data type.

Remember what we said about **Variant** data types earlier? If your argument is a **Variant** and its corresponding parameter is not, the **Variant** argument must be passed by value. This is accomplished by putting a set of extra parentheses around the **Variant** argument.

You can accomplish the same objective by using the **ByVal** keyword when you declare your parameter in the procedure.

Example

```
Function Reverse (S As String, ByVal N As Integer) As Variant
'...SomeStatements
End Function

Dim U As Variant, V As Variant, W As Variant
V = "Testing"
W = 10
U = Reverse ((V), W)
```

Passing Arguments by Value

Visual Basic passes arguments by reference as a default. What this means is that the procedure can modify the values of the arguments in the procedure list because it knows the address for the data. This in effect allows you to pass back more than one value merely by changing the arguments within the procedure.

In contrast, you can pass an argument to a procedure by value, which means that the procedure receives only a copy of the data and cannot modify the value of the actual argument. Any changes made to the argument within the procedure are local and have no effect on the actual data.

To pass an argument by value, use the keyword **ByVal** in the parameter list or put a set of parentheses around the argument in the calling statement.

Types of Procedures

- Event Procedures
- General Procedures

Types of Procedures—A Second Pass

Visual Basic applications have two categories of procedures: event and general.

Event Procedure

An event procedure is a procedure invoked by a user- or system-triggered event.

Event procedures are always attached to a given form or control. The first part of an event procedure's name indicates which object it is attached to.

Syntax

Sub *objectname_eventname* ()

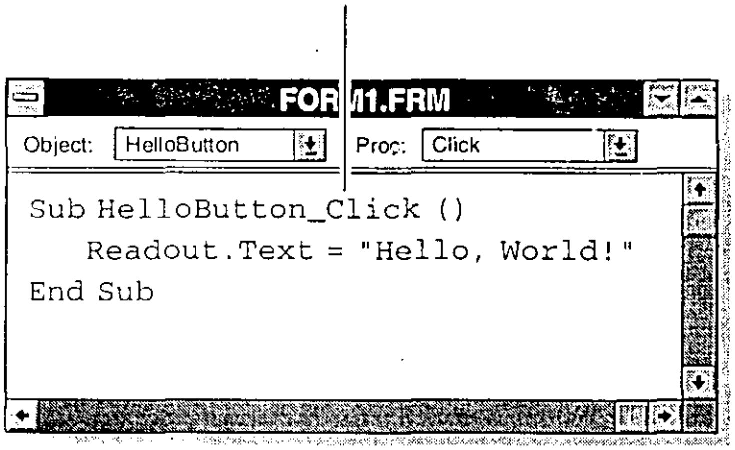
statementblock

End Sub

Examples of event procedures are: Command1_Click and Form_Click. If the user clicks on the command button named Command1, the event procedure Command1_Click will be called; but if the user clicks on the *form*, the event procedure Form_Click will be called.

Creating the Event Procedure

Procedure Name Determines
What Event to Respond To



Object: HelloButton

Proc: Click

```
Sub HelloButton_Click ()
   Readout.Text = "Hello, World!"
End Sub
```

Templates (**Sub** and **End Sub** statements) are supplied for all the events Visual Basic automatically recognizes.

All you need to do is fill in the code:

1. Open the Code window.
2. In the Object list box, select the appropriate object.
3. In the Procedure list box, select the appropriate event.
4. Type code into the template provided by Visual Basic.

Calling the Event Procedure

You don't have to do anything to call the event procedure. Visual Basic automatically recognizes all the events for all Visual Basic objects (forms and controls). As soon as a user or the system triggers an event for an object, the code in the appropriate event procedure will be run by Visual Basic.

Scope of Event Procedures

Event procedures are only available on the form where they were defined.

Note Event procedures may only be **Sub** procedures, not **Function** procedures.

The following is a Click event procedure for a command button.

Example

```
Sub Command1_Click ()
    ' This value can be found in CONSTANT.TXT
    Const RED = &HFF&
    ' Set the background color of the form to red
    Form1.BackColor = RED
End Sub
```

Event Procedures in Visual Basic

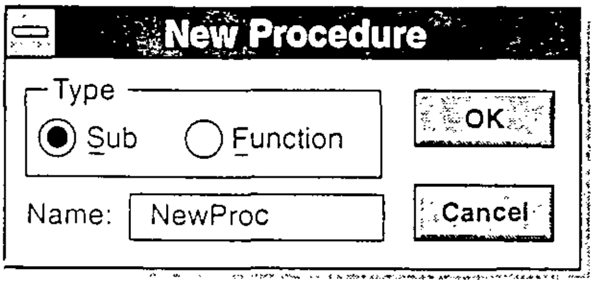
A large number of event procedures are available for your use in Visual Basic. The ones set in italic type below are covered in this course, and the remaining event procedures are covered in the *Programming in Microsoft Visual Basic 3.0* course.

Action	Event
Change to control	*Change*
	DropDown (combo box only)
	PathChange, *PatternChange* (file list box only)
Drag and drop	DragDrop, DragOver
Dynamic data exchange (DDE)	LinkClose, LinkError, LinkExecute, LinkOpen
Keystroke	KeyDown, KeyUp, KeyPress
Mouse	*Click*, *DoubleClick*, MouseDown, MouseUp, MouseMove
Shift in focus	GotFocus, LostFocus
Timer interval	*Timer*
Forms and picture	*Paint*, *Resize*, *Load*, *Unload*

References

Use the online Help or the *Microsoft Visual Basic Language Reference* to find descriptions of the event procedures in Visual Basic.

General Procedure



Type
● Sub ○ Function

Name: NewProc

A general procedure is a procedure executed only when explicitly called by another procedure.

Creating the General Procedure

1. Open the Code window.
2. From the View menu, choose New Procedure.
3. Type the name for the procedure, and choose either Sub or Function.
4. Add the code to the procedure template provided by Visual Basic. If you are creating a new function, remember to define the return type of the function and to include a statement in the code assigning the value you want returned to the function name.

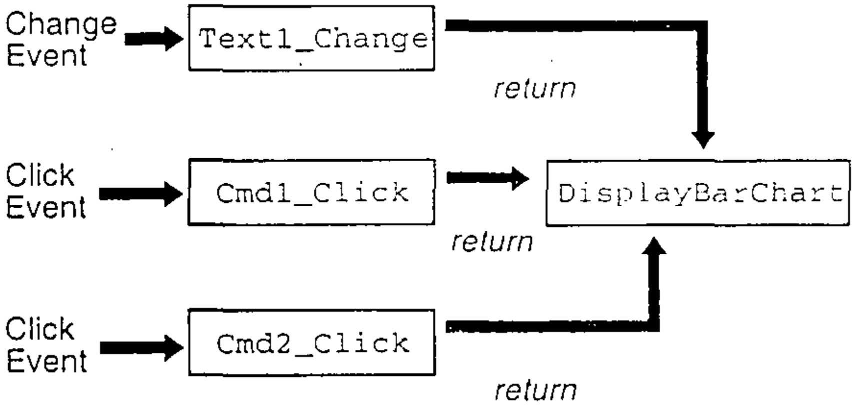
Example

```
Function StockValue (NumShares As Integer, SharePrice As Single)
^ As Single
    StockValue = NumShares * SharePrice
End Function

Sub TotalStock ()
    Print "Total value of all stock is: "
    Print  StockValue(10, 123.25) + StockValue(200, 19.75)
End Sub
```

Calling the General Procedure

You must explicitly call a general procedure, or the code will never be run. Typically, you place a call to a general **Sub** or **Function** inside an event procedure or inside another general procedure.

Scope of General Procedures



Form-Level Scope of Code (.FRM)

The scope of a general procedure depends on where it is defined. If you follow the steps listed above while you are in the Code window for one of your forms, then that procedure is available from anywhere on that form only.

Global Scope of Code (.BAS)

If you have a multiple-form application and need to be able to call certain procedures from anywhere in your application, you will need to create a separate .BAS module to hold those procedures.

Note, for example, if DisplayBarChart is defined as a general procedure for Form1, then Text1, Cmd1, and Cmd2 must all be controls on Form1. If, however, DisplayBarChart is defined in a separate .BAS module, Text1, Cmd1, and Cmd2 could each be on a different form and still be able to call DisplayBarChart.

Private Scope of Code (.BAS)

In some cases, you will want to limit the accessibility of a procedure contained within a module. In order to do this, use the **Private** keyword in front of **Sub** or **Function** definitions.

Declaring a Private Sub Procedure

Syntax

Private Sub *SomeSub* ()

SomeStatements

End Sub

Declaring a Private Function

Syntax

Private Function *SomeFunction* () **As Integer**

SomeStatements

End Function

Method

A method is a special type of procedure provided for you by Visual Basic for specific objects but not associated with a specific event.

Special Characteristics of a Method

1. You cannot create a method; you can only call it.
2. You cannot view or change the code for a method.
3. The names of all Visual Basic methods are keywords. You cannot create a general procedure of your own with the same name as a Visual Basic method.

The following calls a method from an event procedure attached to a form:

```
Sub Form_Click()                ' Form Click event for Form1
    Form2.Show                  ' Display Form2
End Sub
```

Methods in Visual Basic

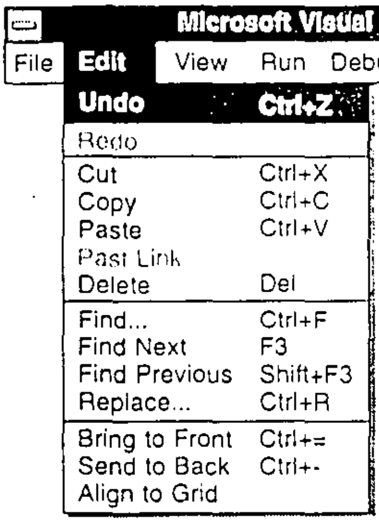
A large number of methods are available for your use in Visual Basic. The ones set in italic type below are covered in this course.

Drawing and graphics	**Circle, Cls, Line, Point, Pset**
Printing	**EndDoc, NewPage, *Print*, *PrintForm*, TextHeight, TextWidth**
DDE	**LinkExecute, LinkPoke, LinkRequest, LinkSend**
List box management	***AddItem*, *RemoveItem*, *Clear***
Clipboard	**Clear, GetData, GetFormat, GetText, SetData, SetText**
Moving controls	**Drag, Move**
Form management	***Hide*, *Show*, Refresh, Scale, SetFocus**

References

Use the online Help or the *Microsoft Visual Basic Language Reference* to find descriptions of the methods available for Visual Basic objects.

Writing Code in Visual Basic



There are several facilities built into Visual Basic that make writing code easy. All of the search and replace procedures discussed below let you search in the current procedure only, in the current module only, or in all modules.

- Cutting, copying and/or pasting code

  Edit menu, Cut or Copy, and Paste
- Finding strings (variable names, for example) in code

  Edit menu, Find
- Finding the next instance of a string in code

  Edit menu, Find Next
- Finding the previous instance of a string in code

  Edit menu, Find Previous
- Finding and replacing a string in code

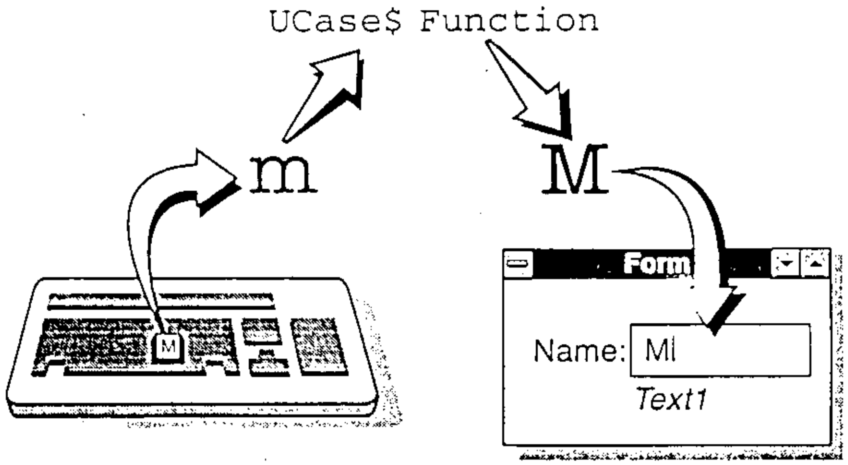
  Edit menu, Replace
- Loading text from the hard disk

  File menu, Load Text
- Saving code as text out to the hard disk

  File menu, Save Text

Note Visual Basic offers a useful feature for people writing code. By default it checks the syntax of your code as you are writing it. This is good news because you get constant feedback on the correctness of the syntax. Because there might be times when some people find this intrusive, this option can be disabled.

∑ String and Numeric Conversion Functions



Visual Basic provides a number of prewritten functions that make your work with strings a lot easier. Below is a partial list of prewritten functions. The focus here is on what many feel are the most important library functions. For a more detailed listing of library functions, see Table 1, "Functions, Statements, and Methods by Programming Task" in the *Microsoft Visual Basic Language Reference.*

Returns Variant	Returns String	Meaning/syntax, example, note
Chr	Chr$	Returns a one-character string for an ANSI code argument. For example, Chr$(13) + Chr$(10) is a carriage return and line feed, which creates as new line.
Format	Format$	A powerful function that displays a number in the format you request.
LCase	LCase$	Returns the lowercase instance of an uppercase character.
Left	Left$	Returns a specified number of the leftmost characters of a string. Left$(*stringexpression, n&*) See online Help for an example.
	Len	Returns the number of characters in a string or the number of storage bytes required by a variable.
LTrim	LTrim$	Returns a copy of a string with leftmost spaces removed.
Mid	Mid$	Returns a string that is part of another string. Mid$(*stringexpression$, start&, [length%]*)
Right	Right$	Returns a specified number of the rightmost characters in a string. Right$(*stringexpression, n&*)
RTrim	RTrim$	Returns a copy of a string with the rightmost spaces removed.

Returns Variant	Returns String	Meaning/syntax, example, note
Str	Str$	Converts a number to a string of digits.
Trim	Trim$	Removes leading and trailing spaces from a string.
UCase	UCase$	Returns the uppercase instance of a lowercase character.
	Val	Converts a string of digits to a number.

String Conversion Functions in the Employee Database

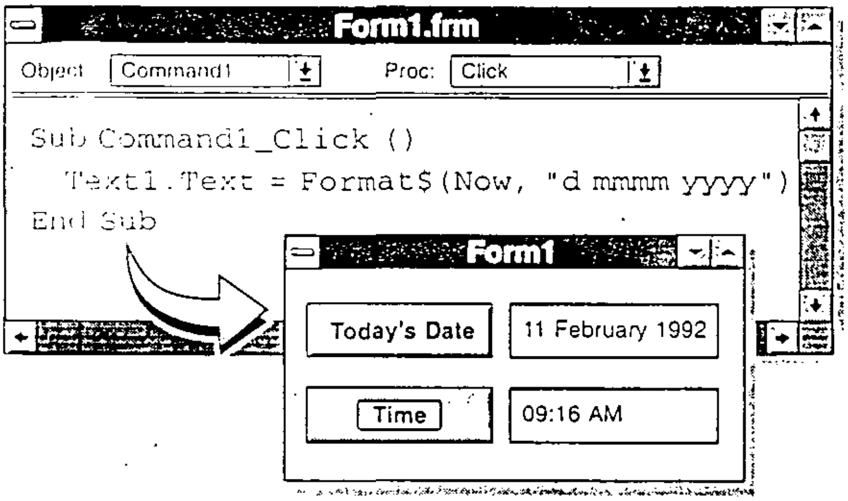
Function	Sub procedure	Form
Chr$	(not used)	
LCase$	(not used)	
Left$	(not used)	
Len	FillFields	EMPDB.FRM
LTrim$	(not used)	
Mid$	FillFields	EMPDB.FRM
Right$	cmdOK_Click	ADDPHOTO.FRM
RTrim$	FillFields	EMPDB.FRM
Str$	Form_Load FillFields	EMPDB.FRM
UCase$	(not used)	
Val	cmdView_Click	EMPDB.FRM

Numeric Conversion Functions

There are a number of functions available within Visual Basic that will convert data types.

Function	Comments
CCur	Converts a numeric expression to a **Currency** value.
CDbl	Converts a numeric expression to a double-precision number.
CInt	Converts a numeric expression to an **Integer** by rounding the fractional part of the expression.
CLng	Converts a numeric expression to a **Long** (4-byte integer) by rounding the fractional part of the expression.
CSng	Converts a numeric expression to a single-precision value.
CStr	Converts a numeric expression to a **String** value.
CVar	Converts a numeric expression or **String** to a **Variant**.
CVDate	Converts an expression to a **Variant** of VarType 7 (Date).

Format$ Function



Object: Command1
Proc: Click

```
Sub Command1_Click ()
  Text1.Text = Format$(Now, "d mmmm yyyy")
End Sub
```

11 February 1992

09:16 AM

This function converts a number to a string and formats it according to instructions contained in a format expression.

Syntax

Format$(***numeric-expression[, fmt$]* **)

fmt$

A format expression is a string of Visual Basic display-format characters that detail how the numeric expression is to be displayed.

Here are several sample format expressions and how the output would be displayed.

Format$(fmt$)	Positive 5	Negative 5	Decimal .5
0.00	5.00	-5.00	0.50
#,##0	5	-5	1
$#,##0.00;($#,##0.00)	$5.00	($5.00)	$0.50

The **Now** function can be used to return the current system date/time as a serial number. Date/time serial numbers can then be formatted with date/time or numeric formats (because date/time serial numbers are stored as floating-point values).

The following are examples of date and time formats.

Format	Display
m/d/yy	12/7/58
d-mmmm-yy	7-December-58
d-mmmm	7-December
mmmm-yy	December-58
hh:mm AM/PM	08:50 PM
h:mm:ss a/p	8:50:35 p
h:mm	20:50
h:mm:ss	20:50:35
m/d/yy h:mm	12/7/58 20:50

Example

```
Sub cmdDisplayDate ()
    txtTodaysDate.Text = Format$(Now, "mm/dd/yy")
End Sub
```

A complete description of the Format$ function can be found in Visual Basic Help.

Further Examples

Application	Form	Procedure
IconWorks	ABOUTBOX.FRM	Form_Load
	COLORPAL.FRM	Txt_RGB_Change
	COLORPAL.FRM	Display_New_Color_and_Elements
	ICONEDIT.FRM	Save_Settings_To_INI_File
	ICONEDIT.FRM	Paste_ClipBoard_Contents
	ICONEDIT.FRM	Save_Colors_To_INI_File
	ICONEDIT.FRM	Prepare_For_New_Icon
	ICONEDIT.FRM	Display_Mouse_Coordinates
	VIEWICON.FRM	Form_Load
	VIEWICON.FRM	Load_All_Icons
	VIEWICON.FRM	File_FileList_PathChange
	VIEWICON.FRM	Form_Unload

IconWorks makes extensive use of the Format$ function. The above table is only a partial listing.

Summary

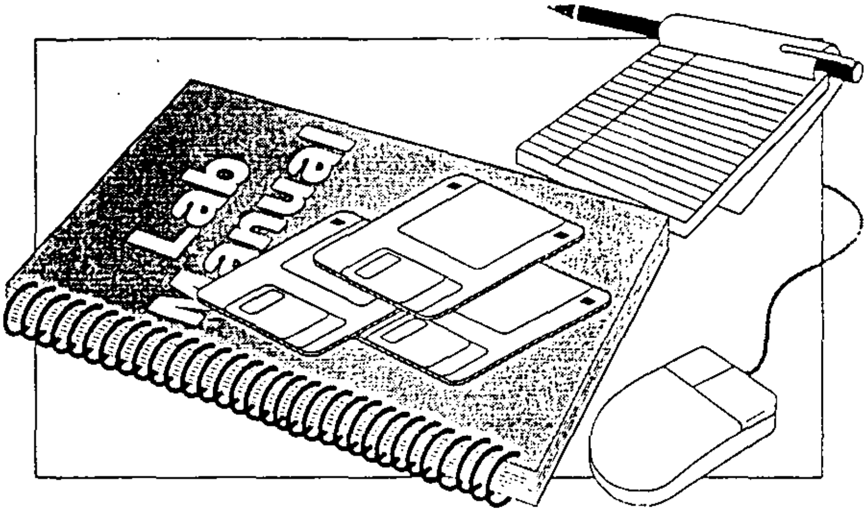
- Visual Basic Procedures
- Scope of General Procedures
- Writing Code in Visual Basic
- String and Numeric Conversion Functions

Objectives

In this module you learned to:

- Define important characteristics of **Functions** and **Sub** procedures and write code that correctly uses both.
- Define and correctly code for appropriate scope of data.
- Write code that uses the common Basic data types.

Lab Time



Go to the Writing Procedures portion of your lab manual.

Module 10: Using Conditional Logic and Loops

∑ Overview

- Control Structures

  If...Then Blocks

  Select Case Statements

  Do While Loops

  Do Until Loops

  For...Next Loops

  GoTo Statements

Overview

Control structures are a crucial part of any computer language because they enable systematic decision making within the code. In this module you will learn how to control the logical flow of your program. You will also learn how to mark off blocks of code that are to be executed if a specified condition is true or false. You will also learn how to specify the number of times that a given block of statements is to be executed.

Prerequisites

This module assumes a fairly detailed understanding of coding mechanisms in Visual Basic. You will also need proficiency in designing and building the user interface for software applications. You should already be familiar with:

- Forms and properties
- Statements, **Sub** procedures, and **Function** procedures
- General and event procedures, as well as methods
- Visual Basic data types
- Variables and constants
- Use of the **MsgBox$** statement and concatenating strings
- String handling functions

Overall Objective

At the end of this module, you will be able to write code that provides systematic decision making within an application.

Learning Objectives

At the end of this module, you will be able to use:

- **If...Then...Else** blocks
- **Select Case** statements
- **Do While** loops
- **Do Until** loops
- **For...Next** loops
- **GoTo** statements

∑ Control Structures

- If...Then Blocks
- If...Then...Else Blocks
- Select Case Statements

If...Then Blocks

If *condition* Then *statement*

If *condition* Then

statements

End If

The foil shows that there are two possible arrangements for **If...Then** blocks. You can use either a single line or multiple lines containing multiple statements. If you have multiple statements, you need to use **End If**.

Operators

There are six operators that you can use in the condition portion of the **If...Then** block.

Operator	Meaning
=	Equal
<>	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Note In this case the = operator is *not* being used for an assignment statement, such as: `ReadOut.Caption = "0."`; instead it is being used for a conditional test.

Example

```
If optFullTime.Value = TRUE Then PositionType = "Full Time"
```

Examples in the Employee Database Code

For a list of example **If...Then** blocks, **If...Then...Else** blocks, and **Select Case** statements in the Employee Database code, see the listing at the end of this module.

If...Then...Else Blocks

```
IF condition1 Then
   statementblock1
ELSEIF condition2 Then
   statementblock2
ELSE
   statementblockn
ENDIF
```

Here you get a chance to test for several different situations and react appropriately to each one. Adding Else to the If...Then statement provides much more flexibility in the response.

Walk Through—If...Then...Else Example

Σ **To use the If...Then...Else statement**

1. From the Walk Throughs program group, start IfElse.

   When the application starts, there will be a form with a check box labeled Bold with an X in it and a command button labeled Print Something.

   The purpose of the application is to show an example of where an **If...Then...Else** block might be used.

2. Click the Print Something command button.

   The word "Something" will appear on the form in bold.

3. Click the Bold check box.

   This removes the X from the check box.

4. Click the Print Something command button.

   The word Something will appear on the form with the bold format removed.

5. Close **IfElse**.

Σ **To create the If...Then...Else example**

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Start a new project.

3. Select FontBold on the Properties list box.

   View the Form property FontBold. Note that the default value is **True**. This means when the chkBold check box is created, its value should be set to Checked.

4. Double-click the check box tool in the Toolbox.

   Create a check box on the form.

5. Set the following properties for the check box:

Name	chkBold
Caption	Bold Font
Value	1 - Checked

6. Double-click the command button tool.

   Create a command button.

7. Set the following properties for the command button:

Name	cmdPrintSomething
Caption	Print Something

   Now that you have created the interface, you need to add the code to make it function.

8. From the Project window, choose View Code.
9. In the Object list box, select chkBold_Click.
10. Add the following code:

```
Const CHECKED = 1
If chkBold.Value = CHECKED Then
    Form1.FontBold = TRUE
Else
    Form1.FontBold = FALSE
End If
```

   This will cause the font to be set to bold or not, depending on how the user has set the check box.

11. In the Object list box, select cmdPrintSomething_Click.
12. Add the following:

```
Print "Something"
```

   This will print the word Something to the form. Depending on the value of the check box, it will be in bold format or not.

13. From the Run menu, choose Start.
14. From the Run menu, choose End.
15. From the File menu, choose Save As.
16. Save the form as IFELSE.FRM in \WALKTHRU\LOGIC.
17. Save the project as IFELSE.MAK in \WALKTHRU\LOGIC.

Select Case Statements

```
Select Case testexpression
    Case expressionlist1
        statementblock1
    Case expressionlist2
        statementblock2
    Case Else
        statementblockn
End Select
```

Select Case statements look a lot like **If...Then...Else** blocks, and there is a good reason for this: **Select Case** statements offer the same kind of functionality but in a much more efficient—for both the code and the coder manner. As you can see from the example, **Select Case** statements work particularly well as a means for handling structured choices offered to the user.

Walk Through—Coding a Message Box with a Select Case Statement

In another module, you created a message box that prompted users to save their data prior to closing the application. This message box allowed users three choices: Yes, No, and Cancel.

This module has provided the tools needed to write code to detect which button the user clicked: the **Select Case** statement.

∑ **To code a message box with a Select Case statement:**

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Open MSGBOX2.MAK.

   This file is located in \WALKTHRU\LOGIC and should already contain the following:

```
Sub Command1_Click ()
     ' The values for MsgBox constant declarations
       come from \VB\CONSTANT.TXT
       MsgBox parameters
    Const YESNOCANCEL = 3                      ' Yes, No, & Cancel btns
    Const ICONQUESTION = 32                    ' Warning query
    MsgS = "Have you saved all your work?"
    MsgBox MsgS, YESNOCANCEL + ICONQUESTION, "MsgBox WalkThru"
End Sub
```

   In order to get the buttons to respond appropriately, you need to add a number of statements to this. Do that by declaring a variable and several constants.

4. In design mode, double-click the command button on the form.

   This opens the Code window with the event procedure code in it.

5. Add the following code to the Code window:

```
Const CANCEL = 2
Const YES = 6
Const NO = 7
Dim MBResponse As Integer
```

   The Cancel, Yes, and No declarations define more meaningful names for the values that are returned for each of the buttons. For example, if the user clicks the No button on the message box, the value 7 is returned by the system. For code readability and maintainability, you should place **Const** declarations in your code and use the constants in the **Select Case** statement. Remember, you can find the correct values to declare in \VB\CONSTANT.TXT or by searching online Help for the topic **MsgBox** function and checking the table of return values.

In the earlier message box example, you created a message box by calling the **MsgBox** statement. Even though you created a message box with three buttons, you had no way to tell which button the user clicked. Another way to create a message box is to call the **MsgBox** function. In this case, you need to declare an **Integer** variable to receive the value the **MsgBox** function returns.

Next, you need to alter the **MsgBox** statement, making it into a function call Remember, when you call a function you must assign its return value to the appropriate type of variable. If you pass the function any arguments, the argument list must be enclosed in parentheses.

6. Edit the original **MsgBox** statement so that it reads:

```
MB_Response = MsgBox(Msg$, YESNOCANCEL+ICONQUESTION,
^"MsgBox WalkThru")
```

Now you need to add the Select Case statement. You may want to add only the actual code and leave the comments out.

7. Add the following code:

```
' When an integer value is returned by MsgBox function.
Select Case MB_Response
     ' if it matches the value of the Const YES (6).
     Case YES
          ' indicate that the user clicked the Yes button.
          Print "User clicked Yes"
     ' if it matches the value of the Const NO (7).
     Case NO
          ' indicate that the user clicked the No button
          Print "User clicked No"
     ' if it matches the value of the Const CANCEL (2)
     Case CANCEL
          ' indicate that the user clicked the Cancel button
          Print "User clicked Cancel"
End Select
```

8. From the Run menu, choose Start.

   .Test the code of your application.

9. From the Run menu, choose End.

   Close the application.

   End the walk through.

Examples in the Employee Qatabase Code

Form	Procedure/Routine	Control structure type
EMPDB.FRM	FillFields	If...Then If...Then...Else
	cmdDelete_Click FillFields	Select Case statement Select Case statement
ADDPHOTO.FRM	cmdOK_Click	If...Then...Else

∑ Loops

- Do While
- Do Until
- For...Next

Do...Loop

```
Do While condition
    statements
Loop
```

– Or –

```
Do
    statements
Loop While condition
```

Use a **Do...Loop** to execute a block of statements an indefinite number of times. (By contrast, **For** loops let you specify how many times a set of statements are executed.) In **Do... Loop While** loops the number of times the loop is executed is controlled by a **True/False** condition. When some other event switches the value of the condition from **True** to **False**, the looping stops.

What's the Difference?

The location of the **While** condition is the key difference between the two examples of syntax above, but what does that mean? If the **While** condition is at the top of the block, the statements within the loop will not be executed if the condition is false to begin with. The block of statements will always be executed at least once if the **While** condition is placed at the bottom of the loop.

Walk Through—Coding a Do While Loop

∑ **To code a Do While loop**

1. Start Visual Basic.

   From the File menu, choose New Project.

2. Double-click the command button tool in the Toolbox.

   A command button appears on the form.

3. Double-click the command button on the form.

   This will open the command button Code window.

4. Add the following code to Form1.frm:

```
Sub Command1_Click ()
    Dim I As Integer
    I = 1
    Do While I <= 5
        Print I
        I = I + 1
    Loop
End Sub
```

5. From the Run menu, choose Start.

   Run the application.

6. From the Run menu, choose End.

   End the walk through.

Do...Loop

Do...Loop Until loops are almost identical to **Do...Loop While**, but they test to see if condition is false rather than true.

Further Examples

For further examples of ...Loop While, see the following.

Application	Module/Form	Procedure/Routine
IconWorks	ICONWRKS.BAS	Help File_In_Path

Walk Through—Do...Loop While with Lists

∑ **To use Do...Loop While with lists**

1. From the Walk Throughs program group, start Do While with Lists.

   When the application starts, a form with a list box and a command button labeled Add To List appears.

   The purpose of the application is to show an example of using a **Do...Loop While** structure.

2. Choose the Add To List button.

   An input box will appear prompting the user to add . item to the list box.

3. Type some text and choose OK.

   The text just typed in the text box will be added as an item to the list on the loop example form.

4. Repeat the above step several times.

   Note that if you add more items to the list than can be displayed at once in the list box, a vertical scroll bar will automatically be added to allow you to view all items. In the case of this example, you must add at least seven items to the list to make a scroll bar appear.

5. Quit the application.

∑ **To use Do While loops in list processing**

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Start a new project.

3. Set the following properties:

Caption	Do Loop While Example
Height	4035 (approximately)
Width	3870 (approximately)
Left	5565 (approximately)
Top	1290 (approximately)

4. Double-click the list box tool in the Toolbox.

   Create a list box on the form.

5. Set the following properties:

Height	1455 (approximately)
Width	1215 (approximately)
Left	960 (approximately)
Top	360 (approximately)

6. Double-click the command button tool in the Toolbox.

   Create a command button.

7. Set the following properties:

Name	cmdAddToList
Caption	Add to List
Height	375 (approximately)
Width	1215 (approximately)
Left	1080 (approximately)
Top	1920 (approximately)

   Now that you have created the interface, you need to add the code to make it function.

8. From the Project window, choose View Code.
9. In the Object list box, select cmdAddToList_Click.
10. Add the following code:

```
Dim Response As String
Do
    Response = InputBox$("Item to add:", "Add Item", "", 0, 1000)
    If Response <> "" Then List1.AddItem Response
Loop While Response <> ""
```

   By creating a **While** loop with the test at the bottom, you ensure going through the loop once to prompt for input, before any conditions are tested.

   An input box is similar to a message box, except it contains a text field for user input. The arguments for the input box are:

   Arg1 = Prompt

   Arg2 = Title bar caption

   Arg3 = Default value for text box

   Arg4 = X-position for input box

   Arg5 = Y-position for input box

   An **If** test is added so that the empty string that indicates the user wants to exit the loop will not be added as an item in the list.

   Keep this project open, because in just a moment you are going to add a **Do Until** loop.

11. Test your code.

   End the walk through.

Do Until

```
Do Until condition
    statements
Loop

-Or-

Do
    statements
Loop Until condition
```

Do Until is the opposite of **Do While**. The statements in a **Do Until** loop are executed only while the condition is False. A **Do Until** condition is the equivalent of a **Do While Not** condition.

Walk Through—Do Until Example

Σ **To use Do Until**

1. From the Walk Throughs program group, start Do Until.

   This is an enhanced version of the previous Do While example. When the application starts, a form with a list box, a command button labeled Add To List, and another command button labeled Clear List appear.

   The purpose of the application is to add an example of using a **Do Until** structure.

2. Choose the Add to List button.

   An input box will appear prompting the user to add an item to the list box.

3. Type some text and choose OK.

   The text just typed into the text box will be added as an item to the list on the Loop Example form.

4. Repeat the above step several times.

   Add several items to the list.

5. Choose the Clear List button.

   All items will be cleared from the list box.

6. Quit the application.

Σ To create the Do Until example

1. Start Visual Basic.

   Make sure that you are still working on the **Do While** walk through.

2. Set the following properties for the form:

Caption	Loop Example

3. Double-click the command button tool in the Toolbox.

   Create another command button.

4. Set the following properties of the command button:

Name	cmdClearList
Caption	Clear List
Height	375 (approximately)
Width	1215 (approximately)
Left	1080 (approximately)
Top	2520 (approximately)

   Now that you have created the interface, you need to add the code to make it function.

5. From the Project window, choose View Code.

6. In the Object list box, select cmdClearList_Click.

7. Add the following code:

   ```
   Do Until List1.ListCount = 0
      List1.RemoveItem 0
   Loop
   ```

   A **Do Until** loop can be used to keep removing items from List1 until the list box is empty. The ListCount property keeps track of the number of items currently in the list.

   Inside the loop is a statement to call the **RemoveItem** method. When you pass a zero to **RemoveItem**, you are telling it to remove the top item in the list.

   Rather than setting up a loop, you can clear an entire list box with the **clear** method.

   ```
   List1.clear
   ```

8. From the File menu, choose Save File As.

   Save this file as DOUNTIL.FRM and the project as DOUNTIL.MAK in the \WALKTHRU\LOGIC subdirectory. You will need these files in the For Loop lab.

   End the walk through.

For...Next

For *counter* = *start* To *end* [Step *increment*]

[*statements*]

[Exit For]

[*statements*]

Next [*counter*]

For...Next loops are used to execute a block of statements a fixed number of times. The key difference between a **For...Next** loop and a **Do...Loop** is that a **For...Next** loop includes a counter that increases or decreases with each repetition of the loop.

Complete the demonstration for an example of **For...Next** loops.

Walk Through—Using Visual Basic For...Next Loops

∑ **To use For...Next loops**

1. Start Visual Basic.

. Open Visual Basic. Open a new form.

2. From the File menu, choose New Project.
3. Double-click the command button tool in the Toolbox.

This will open the command button Code window.

4. Add the fc'lowing code in the Code window:
5. Add the appropriate code to the Command1_Click event:

```
Sub Commandl_Click ()
    Dim I As Integer
    For I = 1 to 6
        Print I
    Next I
End Sub
```

6. From the Run menu, choose Start.

Test the application you have created. The default value for Step is 1. The application prints the numbers 1 through 6 on the form.

7. From the Run menu, choose End.
8. In the Code window, change the Step value to 2.
9. From the Run menu, choose Start.

   Test the application again.

   How has the output you are getting changed?

   How would you change the **For...Next** loop to display only even numbers?
10. From the Run menu, choose End.

    End the walk through.

A more sophisticated use of the **For...Next** loop can be found in Visual Basic Help.

Example

```
'For...Next Statement Example
Sub ForNextDemo ()
    NL$ = Chr$(13) + Chr$(10)            ' Define newline.
    For Rep% = 5 To 1 Step -1            ' Set up five repetitions.
            ' Equate alphabet to numbers.
        For Indx% = Asc("A") To Asc("Z")
                ' Append each letter to string.
            Msg$ = Msg$ + Chr$(Indx%)
        Next Indx%
        Msg$ = Msg$ + NL$                ' Add newline for each rep.
    Next Rep%
    MsgBox Msg$                          ' Display results.
End Sub
```

Notice that one **For.. Next** loop is "nested" inside the other in this example? This is a powerful tool for processing multiple rows or columns in arrays.

If you want to see this code running, follow the directions detailed above and make the appropriate modifications to the code so that it executes as the result of a command button being clicked.

Walk Through—Another For...Next Loop Example

∑ To use another For...Next loop

1. From the Walk Throughs program group, start ForLoop to Clear List.

   This is an enhanced version of the previous **Do While** example. When the application starts, a form with a list box, a command button labeled Add To List, and another command button labeled Clear List appear.

   The purpose of the application is to change the example to use a **For...Next** loop to clear the list.

2. Choose the Add to List button.

   An input box will appear prompting the user to add an item to the list box.

3. Type some text and choose OK.

   The text just typed into the text box will be added as an item to the list on the loop example form.

4. Repeat the above step several times.

   Add several items to the list.

5. Choose the Clear List button.

   All items will be cleared from the list box.

6. Quit the application.

∑ To create the For...Next loop example

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Open DOUNTIL.MAK.

   Open the project from the **Do Until** Walk Through. It is located in the \WALKTHRU\LOGIC subdirectory.

4. From the Project window, choose View Code.
5. In the Object list box, select cmdClearList_Click.
6. Replace the existing code with:

```
Dim I As Integer
For I = 0 To List1.ListCount-1
   List1.RemoveItem 0
Next I
```

   You can use a **For...Next** loop to keep removing items from List1 until the list box is empty. In this case, the range for the loop will be from 0 (the top item in the list) to ListCount–1 (the last item in the list).

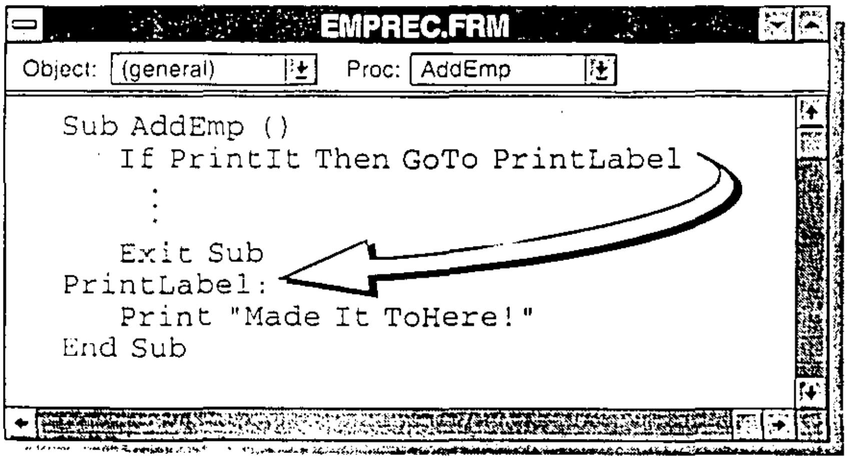
   Inside the loop is a statement to call the **RemoveItem** method. When you pass a zero to **RemoveItem**, you are telling it to remove the top item in the list.

Further Examples

For further examples of a **For...Next** loop see:

Application	Module/Form	Procedure/Routine
Employee Database	EMPDB.FRM	Form_Load

The GoTo Statement



Object: (general) Proc: AddEmp

```
Sub AddEmp ()
   If PrintIt Then GoTo PrintLabel
     :
     :
   Exit Sub
PrintLabel:
   Print "Made It ToHere!"
End Sub
```

GoTo causes execution to jump from the **GoTo** statement to another location marked by a label or line number.

Even though the GoTo statement doesn't enjoy much favor these days, Visual Basic supports it, and there are several places where there is an obvious use for it. Error handling is a particularly good example.

Note GoTos are not like procedures; there is no standard return from a GoTo as there is from a procedure.

Style Guidelines

- Each line label must begin with an alphabetic character.
- Each line label must end with a colon.
- Each line label must be unique within its own module.
- Each line label can have no more than 40 characters.
- Do not use Visual Basic keywords in line labels.
- Line labels are not case sensitive.
- A line label must start with the first nonblank character on a line, but it need not be in the first column. Visual Basic forces it to the leftmost column.

Examples

Additional sample code is located in Visual Basic Help.

For further examples of GoTo statements see the following.

Further Examples

Application	Module/Form	Procedure/Routine
IconWorks	ICONWRKS.BAS	Validate_FileSpec
Employee Database	EMPREC.FRM	AddEmp

Summary

- **Control Structures**

  If...Then Blocks

  Select Case Statements

  Do While Loops

  Do Until Loops

  For...Next Loops

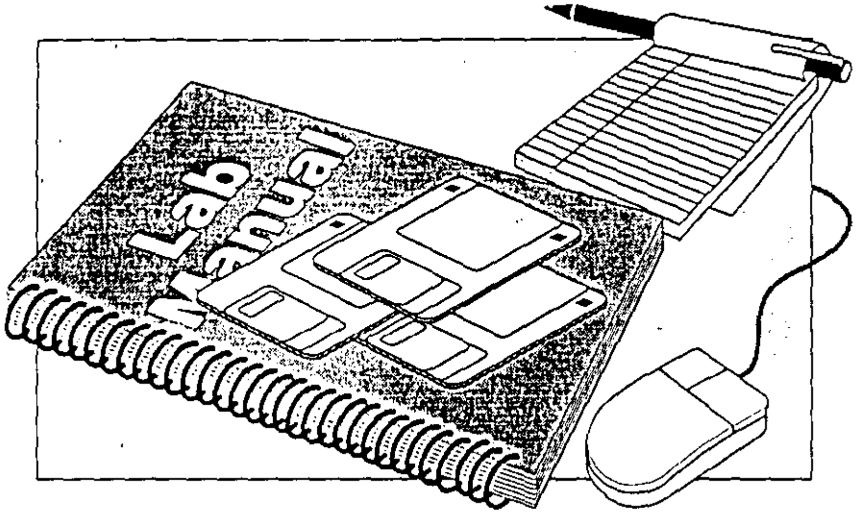
  GoTo Statements

Objectives

In this module you learned to use:

- **If...Then...Else** blocks
- **Select Case** statements
- **Do While** loops
- **Do Until** loops
- **For...Next** loops
- **GoTo** statements

Lab Time



Go to the Conditional Logic and Loops portion of your lab manual.

Module 11: Debugging Code in Visual Basic

Σ Overview

- Debugging Terms
- Debugging Code in Visual Basic

  Using the Call Tree

  Using Watch Variables to Monitor Program Execution

Overview

Visual Basic offers programmers of all skill levels a set of robust tools to use during application development.

Prerequisites

Prior to starting this module, you should already be familiar with:

- Controls, forms, and properties
- Function and Sub procedures
- General and event procedures

Overall Objective

The overall objective of this module is to introduce you to some of the most useful debugging tools available in Visual Basic. This is not intended to be comprehensive treatment of debugging techniques in general.

Learning Objectives

At the end of this module, you will be able to:

- Distinguish among run, design, and debug modes in Visual Basic.
- Use the Watch window to display the current values of variables within a program.
- Set breakpoints within code.
- Single step through application procedures.

Debugging Terms

- Watch Expressions
- Watch Point Variables

  Watch point—break when true

  Watch point—break when changed
- The Debug Window

  The Immediate pane (?, =)

  Watch pane

Watch Expressions

A watch expression is a variable whose value is displayed in the Debug window whenever a program enters break mode.

You can set a watch expression by opening the Debug menu and choosing the Add Watch or Edit Watch command. This can be done in either design mode or break mode.

Watch Point Variables

A watch point variable will cause a program to enter break mode whenever the watch point condition is satisfied.

You can set a watch point variable by opening the Debug menu and choosing either the Add Watch or Edit Watch command. This can be done in either design mode or break mode.

Watch Point—Break When True

This watch point variable will cause a program to enter break mode whenever the variable's value becomes true.

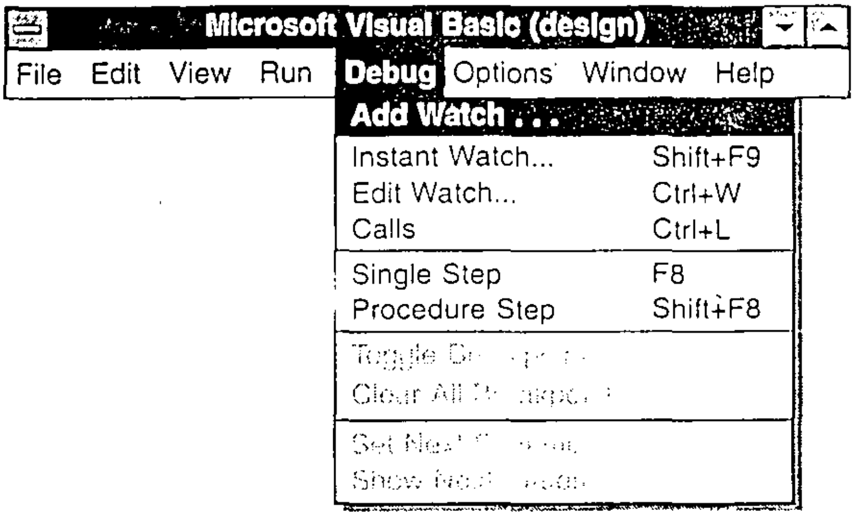
Watch Point—Break When Changed

This watch point variable will cause a program to enter break mode whenever the variable's value changes.

The Debug Window

The Debug window is broken into two parts: the Immediate pane (the lower half of the window) and the Watch pane (the upper half). The Immediate pane is where you can have an interactive conversation with the debugger using either the question mark (?) or the equal sign (=). The Watch pane is where the watch variables are displayed during break mode.

Debugging Code in Visual Basic



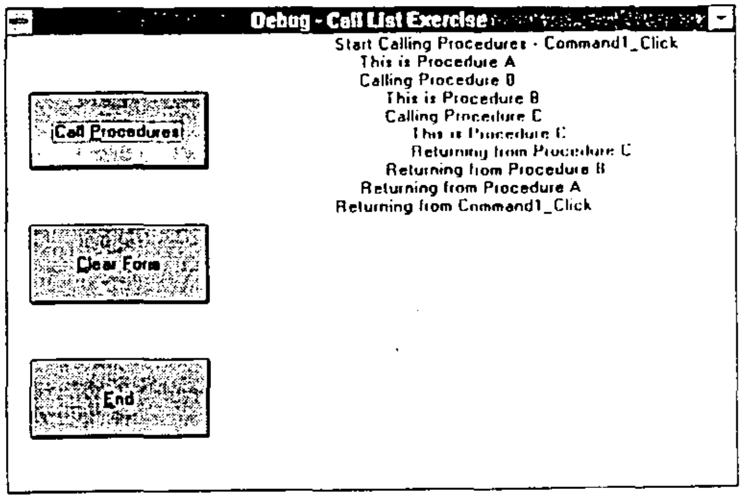
Using the Visual Basic Debugger

This module is actually made up of a number of walk throughs that introduce you to the various tools available within the Visual Basic debugger.

Walk Through—Using the Call Tree

∑ **To see the first sample application work**

1. From the Walk Through program group, start Debug1.
2. Click the Call Procedures command button.
3. Observe the output generated on the form.



```
Start Calling Procedures - Command1_Click
    This is Procedure A
    Calling Procedure B
        This is Procedure B
        Calling Procedure C
            This is Procedure C
            Returning from Procedure C
        Returning from Procedure B
    Returning from Procedure A
Returning from Command1_Click
```

Note that Command1_Click calls ProcedureA, ProcedureA calls ProcedureB, and then ProcedureB calls ProcedureC.

4. Choose End.

∑ To examine the code for the walk through

The purpose of this walk through is to introduce you to two tools that you will find very useful: the Call window and single stepping through code.

1. Start Visual Basic.
2. Open the DEBUG1.MAK file located in \WALKTHRU\DEBUG.
3. Select MOD1.BAS in the Project window and click View Code.
4. From the Procedures drop-down list box, select ProcC.
5. Examine the code for ProcC.

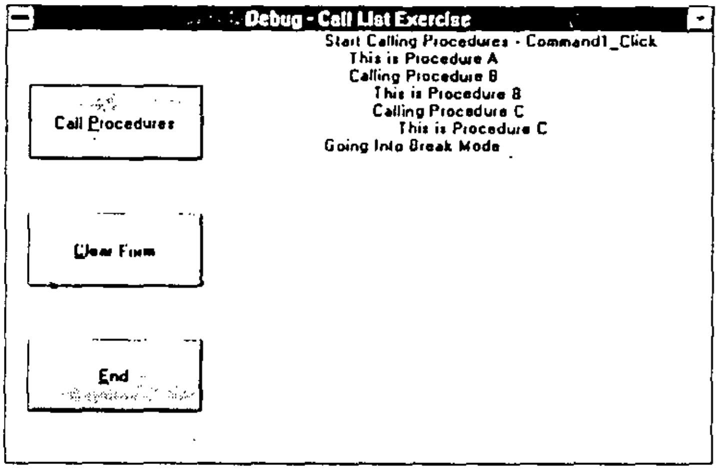
   As it stands, this procedure prints two statements at column 49 on your form. The other three lines are "commented out."

6. Remove the three comment mark· in front of the **Stop** and **Print** statements.

   What will happen when you do this? You have added three new statements to the application. The first one prints a message to the form at column 40 that tells you the application is going into break mode. The second statement actually stops that application using **Stop**. The final statement prints a message to the form that tells you that you have entered single-step mode after pressing F8 twice.

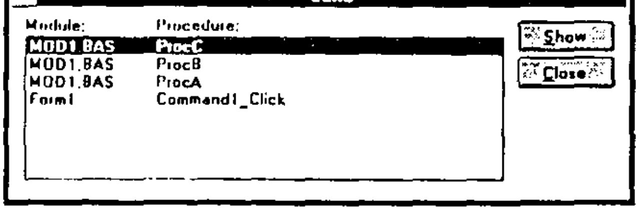
7. From the Run menu, choose Start.
8. Click the Call Procedures command button.

   If you move the Code window from on top of Form1, you will see that the output is slightly changed. Now it should look like this.



```
Start Calling Procedures - Command1_Click
    This is Procedure A
    Calling Procedure B
        This is Procedure B
        Calling Procedure C
            This is Procedure C
Going Into Break Mode
```

9. From the Debug menu, choose Calls. Now you should see a form like this.



Module:	Procedure:
MOD1.BAS	ProcC
MOD1.BAS	ProcB
MOD1.BAS	ProcA
Form1	Command1_Click

Notice that the calls are listed with the most recently called at the top.

10. Click the Show command button.

    This returns you to MOD1.BAS.

    You should see the code for ProcC with a box around the code line containing the **Stop** statement.

11. Single step by pressing F8 once.

    This will start you single stepping through the rest of the procedure. This makes `Print "Going into Single Step Mode"` the next statement to execute.

    If you position both windows so that you can see almost all of them, you will see Visual Basic work as you single step through the application.

12. Press F8 one more time.

    This executes the current **Print** statement and makes `Print "Returning from Procedure C"` the next statement to execute.

13. Press F8 a third time.

    This executes the current **Print** statement (check the output on the form) and makes the **End Sub** statement the next statement to execute.

14. Press F8 a fourth time.

    This executes the **End Sub** of ProcC and takes you to the next statement to execute in ProcB, which is the `Print "Returning from Procedure B"` statement.

15. From the Debug menu, choose Calls or press CTRL+L.

    Note the most recent procedure is now Procedure B.

    Doing this verifies that the call list has changed since the return from ProcC.

16. Click the Show command button to return to ProcB.

17. Press F8 a fifth time and then a sixth time.

    This takes you through the remaining two Procedure B statements and takes you to the next statement in ProcA—`Print "Returning from Procedur A"`.

18. Press F8 twice more.

    This single steps you through the remaining Procedure A statements and takes you to the next statement in Command1_Click—`Print "Returning from Command1_Click"`.

19. Press F8 twice more.

    This takes you through the remaining the Command1_Click event procedure.

20. From the Run menu, choose Break.

21. From the Debug menu, choose Calls.

    Notice that *no calls* are listed. That is because no **Sub** or **Function** procedure is currently "open" and your application is in idle waiting for the user to do something.

22. Choose the Close command button on the Calls dialog box.

23. From the File menu of Visual Basic, choose Exit.

    If you want to try this exercise a second time, do not save the changes to the files.

Walk Through—Using Watch Variables to Monitor Program Execution

Debugging Applications

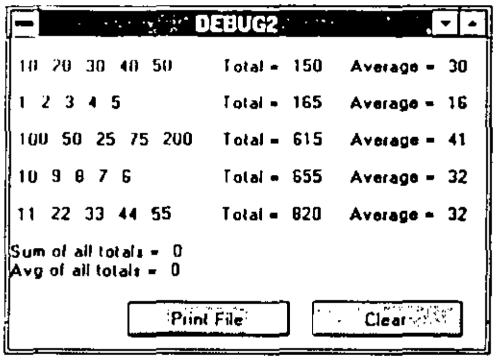
Debugging applications is somewhere between a science and an art. By careful use of your debugging tools, such as breakpoints, watch variables, single stepping, and procedure stepping, you can zero in on the logical bugs existing in your code.

This walk through applies breakpoints and watch expressions to locate a number of logical errors contained within the code.

Σ To see the second debugging application run

1. From the Walk Through program group, start Debug2.
2. Click the Print File command button.

   You get output that looks like this.

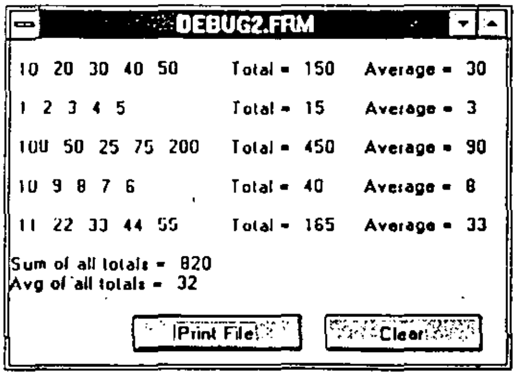


```
10  20  30  40  50        Total =  150   Average =  30
1  2  3  4  5             Total =  165   Average =  16
100  50  25  75  200      Total =  615   Average =  41
10  9  8  7  6            Total =  655   Average =  32
11  22  33  44  55        Total =  820   Average =  32
Sum of all totals =  0
Avg of all totals =  0
```

   This program lists five numbers, their total, and their average. It does this for five sets of data. At the end, it prints the total of all the numbers and the average for all the numbers.

3. But examine the output carefully. It isn t giving you what you want: The totals and averages aren't correct.

   If the application were coded correctly, final output should look something like this.



```
10  20  30  40  50        Total =  150   Average =  30
1  2  3  4  5             Total =  15    Average =  3
100  50  25  75  200      Total =  450   Average =  90
10  9  8  7  6            Total =  40    Average =  8
11  22  33  44  55        Total =  165   Average =  33
Sum of all totals =  820
Avg of all totals =  32
```

   You need to correct the logic, but where do you begin?

4. Double-click the Control menu on the Debug2 form to close the application.

∑ To analyze the variables used in the sample application

1. If Visual Basic is not running already, start it.
2. From the File menu, choose Open Project to locate and start DEBUG2.MAK, located in \WALKTHRU\DEBUG.
3. Note that there are two files to this project: DEBUG2.FRM and MOD2.BAS.
4. What are the *global-level* variables, and where they are located?

   ____

   If you inspect the General Declarations section of MOD2.BAS, you will see x(5), an **Integer** array, and sumcounter declared as an **Integer**.

   There is also a **String** variable called Filename$ in MOD2.BAS.

5. What are the *form-level* variables, and where they are located?

   ____

   If you inspect the General Declarations section of DEBUG2.FRM, you will find Sum declared as an **Integer**. But, be careful here. Visually inspecting the General Declarations section doesn't tell you all you need to know, and this will come back to haunt you a little later on.

   Why do you need to know this? Awareness of the types and scope of the variables within your application will help in the analysis of the code.

∑ To analyze the procedures used in the sample application

1. In DEBUG2.FRM, examine the cmdPrintFile_Click event procedure.
2. Which general procedure is called in this procedure?

   ____

   If you said the Readfile procedure from MOD2.BAS, you were correct.

3. Examine the ReadFile general procedure in MOD2.BAS.

   To display the source for the ReadFile general procedure, place the cursor in the word ReadFile in the cmdPrintFile_Click procedure and press SHIFT + F2

   Note the outer **For...Next** loop is controlled by J, whereas the inner **For...Next** loop is controlled by I. These two loops control part of the calculations in your application.

∑ To use breakpoints and watch expressions to observe program behavior

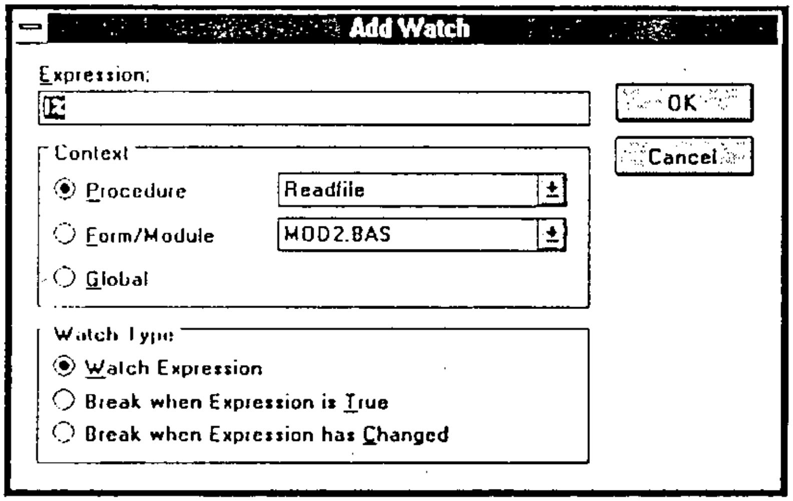
In order to observe the logic flow and I and J's values, we will use two watch expressions and a breakpoint.

1. Set a breakpoint on the `Next I%` line.

   Place the cursor anywhere on the `Next I%` line and press `F9`.

2. Highlight the I% variable on the `For I%` line.
3. From the Debug Menu, choose Add Watch.

   A form will appear on screen that looks like this.



Expression:

Context

● Procedure Readfile

○ Form/Module MOD2.BAS

○ Global

Watch Type

● Watch Expression

○ Break when Expression is True

○ Break when Expression has Changed

4. Make sure that the values for the Add Watch window are as follows.

Control	Setting
Expression	I%
Procedure	Readfile
Form/Module	MOD2.BAS
Watch Type	Watch Expression

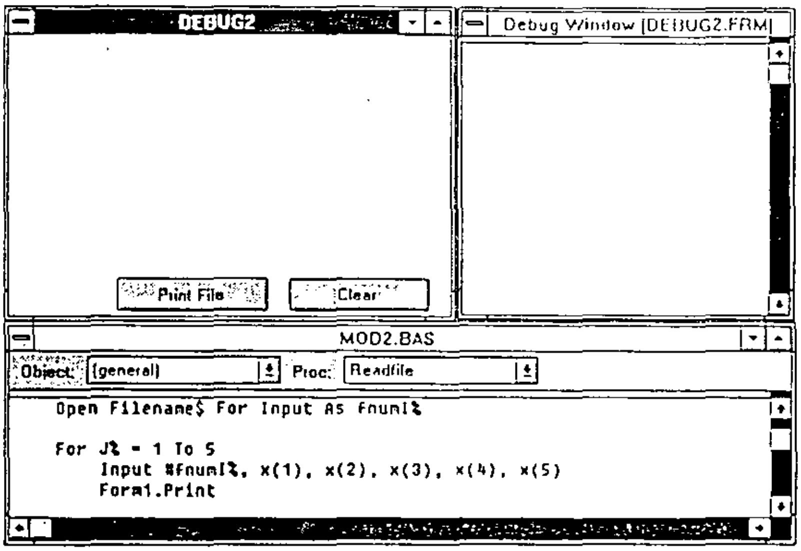
5. Choose OK.
6. Repeat the above steps (1 thorough 5) for J% to create a watch expression for J%.

   You now have two watch expressions. Each time your program enters into break mode, the current values of these two variables will be displayed at the top of the Debug window.

∑ To observe your watch expressions in action

1. Start your program by pressing F5.
2. Arrange the windows so that you can see DEBUG2.FRM, the Debug window, and MOD2.BAS code window.

   You might want to arrange them so that they look like this.



```
    Open Filename$ For Input As Fnum1%

    For J% = 1 To 5
        Input #Fnum1%, x(1), x(2), x(3), x(4), x(5)
        Form1.Print
```

3. Choose the Print File button.

   Because you have set a breakpoint, your application will stop execution on the line `Next I%`.

4. Note the values of I% and J% in the Debug window.

   Right now they should each be at 1 because you are in the first iteration of both loops. If you look at the output of DEBUG2.FRM, you will see that the value :: has been printed there.

5. Continue execution by repeatedly pressing F5 until J% equals 2 and I% equals 5.

   Each time you press F5, your application enters break mode and the current values of I% and J% are displayed in the Watch pane.

   At this point you know that I% and J% are behaving correctly. It is time to pursue analysis in a different direction.

6. From the Run menu, choose End to stop execution of your program.
7. Press F9 to clear your breakpoint on the `Next I%` line.

   The `Next I%` line in the Readfile procedure of MOD2.BAS should already be selected; so press F9.

To this point, you have seen what we wanted to demonstrate about watch points and watch expressions. If that is all you need from this exercise, then quit here. If, however, you want to sharpen your debugging skills, continue on with the exercise.

The question here is this: What is going wrong with the application? First, it doesn't seem to be calculating the individual totals properly. You are ending up with a value of 165 for the second set of numbers when it should be 15. Why don't you next set a watch point on Total%?

∑ To use watch points—break when expression has changed (Total%)

1. In the Code window for MOD2.BAS, locate the procedure Readfile and highlight the variable Total% within the For I% loop.
2. From the Debug menu, choose Add Watch.
3. Make sure that all the controls for the watch point are as follows.

Control	Setting
Expression	Total%
Procedure	Readfile
Form/Module	MOD2.BAS
Watch Type	Break when Expression has Changed

4. Choose OK to close the Add window.

∑ To observe your watch variable in action

1. From the Run menu, choose Start to run your program.

   Note the entry added to the top of the Debug window. This will cause the program to enter into break mode whenever Total% changes.

   You may need to expand the Watch pane display area in order to see the additional watch variable.

2. Click the Print File command button to enter the Readfile procedure.
3. What is the first value for Total%?

   ____

   If you said 10, you were correct.

4. Press F5 six more times (so that I% equals 1 and J% equals 2).
5. Now, what is the value of Total%?

   ____

   If you said 151, you were correct; but what should the value of Total% be at this point, and what does that tell you?

   ____

   The value for Total% at this point should be 1, because that is the first value for the second pass through the J% loop. What this tells you is that you need to reset the value of Total% before each entry into the I% loop.

6. From the Run menu, choose End.
7. Add the following line just before the `For I% = 1 to 5` line:

```
Total% = 0
```

8. Test your change by running the code again. Notice that the total is correct.

   You will notice that the averages for the last four sets of numbers have the values 1, 30, 2, and 6. That is, they are still wrong. To fix this problem, what other variable must you zero out before reentering the I% loop?

   If you said counter%, you were correct.

9. From the Run menu, choose End.
10. Add the following line just before the `For I% = 1 to 5` line:

    ```
    Counter% = 0
    ```

11. Test by running the program again.
12. Inspect the Total and Average for each of the five sets of data.

    Total and Average for each set of numbers should now be correct. Compare them with the correct output displayed earlier in this walk through.

13. Inspect the values displayed for the overall sum and average values.

    They are both 0. That can't be right.

14. From the Run menu, choose End.

Σ To clear all watch variables and all breakpoints

1. From the Debug menu, choose Edit Watch.
2. Choose Delete All to delete all watch points you may have set.
3. Choose Close to close the Edit Watch dialog box.
4. From the Debug menu, choose Clear All to clear all breakpoints that are set.

Σ To use watch points—break when expression becomes True (Sum)

At this point you have fixed the calculation of individual totals and averages, but the calculation of the overall total and average is still wrong. For this reason, closer examination of the Sum variable is warranted.

1. From the Debug menu, choose Add Watch.
2. Fill in the following entries:

Expression	Sum > 0
Form/Mod	DEBUG2.FRM
Procedure	cmdClear_Click

   Break when Expression is True

   Choose OK

3. From the Run menu, choose Start.
4. Note the entry added to the top of the Debug window.

   This will cause the program to enter into break mode whenever Sum > 0 becomes True.

Σ To observe your watch variable in action

1. Press the Print File command button to run the program.
2. Did you ever enter Break mode?

____

Your program should not have entered break mode because the variable Sum was never greater than 0.

3. From the Run menu, choose Break. Notice in the debug window it specifies Sum <Not in Context>. This is because Sum is a local variable and you are not currently running the procedure that contains Sum.
4. From the Run menu, choose End.

Σ To fix the code—using debugging and logical analysis to correct the code

1. First of all, how many variables named Sum are in this application?

____

Remember at the start of this walk through that we asked that question and said that there was only one. Well, there are really two. If you said two, you were correct.

2. Look in the General Declarations section of DEBUG2.FRM. Notice that it contains an explicitly declared *form-level* integer variable called Sum. This variable is not available outside this form.
3. Now look at the code in MOD2.BAS. Is the variable called Sum there, the same one as in DEBUG2.FRM? No, this is an implicit variable local to the Sub procedure Readfile. This is the source of your problem.

   How can you make sure that the variable called Sum in cmdPrintFile in DEBUG2.FRM and the one referred to in Readfile are the same?

____

If you said delete the local variable declaration and make it Global, you were right.

4. Delete the declaration in DEBUG2.FRM.
5. Add the following declaration to MODULE2.BAS:

```
Global Sum As Integer
```

Σ To test your scoping changes

1. Run the program again.

   After making these changes, you'll discover that the sum of all totals is 2025 and average of all totals is 81. They're still not right.
2. Choose End from the Run menu.

3. Set the breakpoint on Next I% and the watch expressions on I% and J% in the Readfile procedure. (Follow the guidelines outlined earlier in the walk through, if you need to.)

Σ To single step through your program

1. From the Run menu, choose Start.
2. Click the Print File command button.

   The next line of code to be executed should be `Next I%`.
3. Now, query for the current value of Sum by placing your cursor in the Debug window and typing:

   ?Sum

   Then press the ENTER key.
4. Observe that the value 10 displayed in the Immediate pane of the Debug window. That's the current value of Sum.
5. Step through the I% loop again, stopping at the same point as before, and query the value of Sum.
6. Observe that the value 40 is displayed. However, 10 + 20 does not equal 40.

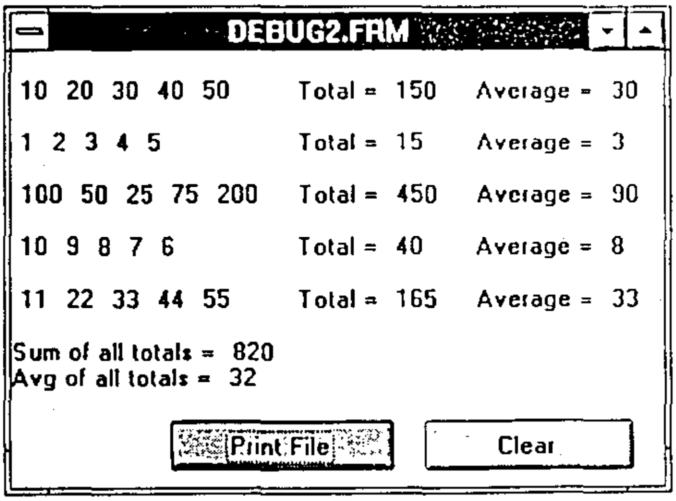
   What's the problem?

   ____

   If you said the line: `Sum% = Total + Sum` needs to be moved below the line: `Next I%`, you're correct.
7. Cut and paste the line `Sum% = Total + Sum` to just below `Next I%`.

Σ To test your logic changes

1. Start and test the application again to see if it's working properly now.
2. Compare your output with the desired final output shown below.



```
10  20  30  40  50       Total =  150     Average =  30
1  2  3  4  5            Total =  15      Average =  3
100  50  25  75  200     Total =  450     Average =  90
10  9  8  7  6           Total =  40      Average =  8
11  22  33  44  55       Total =  165     Average =  33
Sum of all totals =  820
Avg of all totals =  32
```

3. Close the application and give yourself a pat on the back.

Summary

- **Debugging Terms**
- **Debugging Code in Visual Basic**

  Using the Call Tree

  Using Watch Variables to Monitor Program Execution

Objectives

In this module you learned to:

- Distinguish among run, design, and debug modes in Visual Basic.
- Use the Watch window to display the current values of variables within a program.
- Set breakpoints within code.
- Single step through application procedures.

Module 12: Printing to Forms and Printers

∑ Overview

- Scenario
- Methods for Printing
- Print-Related Functions
- Using PrintForm

Overview

Printing to forms and printers can be as complicated or as simple as you want to make it. This module shows you some of the simple tricks that you can use to prepare text for printing.

Prerequisites

To succeed in the module, you should already be familiar with:

- The general syntax for methods.
- **Select Case** statements and **If...Then...Else** blocks.

Overall Objectives

The purpose of this module is to give you an introduction to printing using Visual Basic. This module is split into two topics: printing to a form and sending that form to a printer.

Learning Objectives

At the end of this module, you will be able to:

- Use the **Print** method to create output directly onto a form.
- Describe how the AutoRedraw property relates to printing directly to a form.
- Use the CurrentX and CurrentY properties of a form and the Spc and Tab functions to control the placement of output printed to a form.
- Use the Cls method to clear a form.
- Use the **PrintForm** method to send a bit-for-bit image of a form to the printer.

Scenario

- Printing a Form

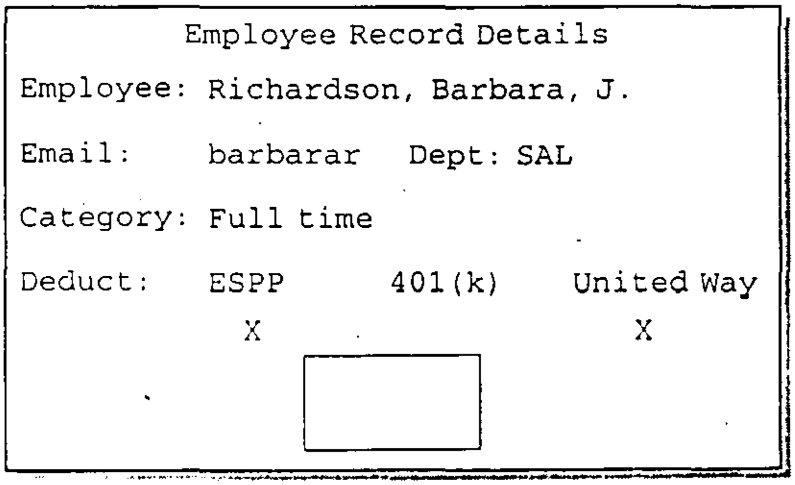
The specification for the Employee Database requires that users be able to print out an individual employee's personnel information.

Individual Reports

You have determined that users want the individual report to include the following information on individual lines:

1. Last name, first name, and middle initial
2. Electronic mail alias and department code
3. Position type
4. Deduction information
5. Employee photo centered at the bottom of the page

After much work, you have determined that a workable individual report form would look like this.



Employee Record Details

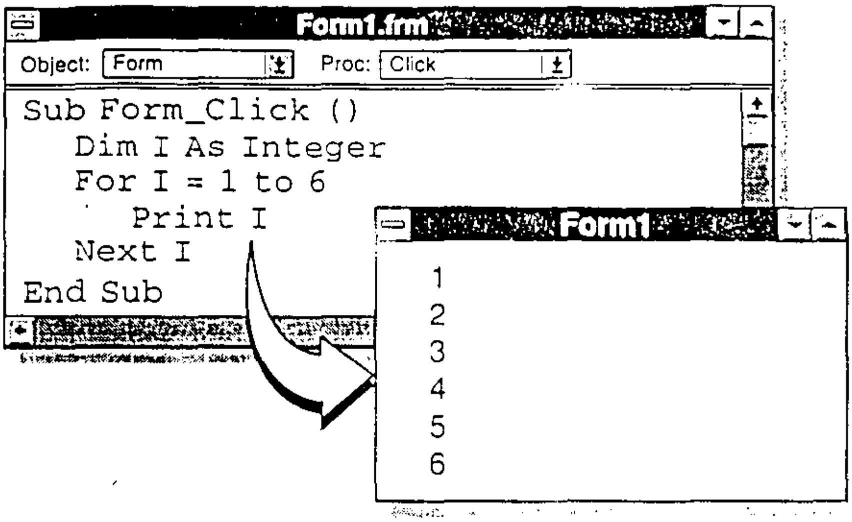
Employee: Richardson, Barbara, J.

Email: barbarar Dept: SAL

Category: Full time

Deduct:	ESPP	401(k)	United Way
	X		X

Methods for Printing



```
Sub Form_Click ()
   Dim I As Integer
   For I = 1 to 6
      Print I
   Next I
End Sub
```

```
1
2
3
4
5
6
```

Overview

There are a number of different methods, functions, and properties you will need to understand in order to print to a form and then send that form to a printer.

Methods	Functions	Properties
Print	Spc	CurrentX
Cls	Tab	CurrentY
PrintForm	Format	AutoRedraw,
		Fontsize, and so on

The Printer Object

The Printer object is a predefined object in Visual Basic. You can send output to the Printer object using the **Print** method. When you are finished placing information on the Printer object, you use the **EndDoc** method to send the output to the printer. The output will print on your default printer.

```
Printer.Print        "Here is some information on Page one."
Printer.NewPage      'This causes a page break
Printer.Print        "Here is some more information"
Printer.EndDoc       'This sends the output to the printer.
```

The Print Method

The **Print** method is used to put a text string on a form, a picture box, or Printer object using the current color and font. This portion of the module will discuss the **Print** method by first talking about it in general and then discussing the implications of printing to a form and printing to a picture box.

Syntax

[*object.*]**Print**[*expressionlist*][{;|,}]

The object can be either a form, a picture, or a printer to which the expression list will be printed.

The expression list is either text or numbers that you want to print. Multiple expressions can be separated with a semicolon, a comma, or a space. If the expression list is omitted, Visual Basic prints a blank line.

The semicolon and the comma are used to specify the location of the text cursor for the next character displayed. A semicolon means the cursor is placed immediately after the last character displayed. The comma means the cursor is placed at the start of the next print zone. Print zones begin every 14 columns.

Note A form does not need to have the Visible property set to **True** to be able to send output to it.

Walk Through—Printing with For...Next Loops

∑ **To print with For...Next loops**

1. From the Walk Throughs program group, start Fancy Print.
2. Click the form.

   This displays the output from the code contained on the next page.
3. From the Control menu, choose Close.

The following example is available in Visual Basic Help.

Example

```
'Print Method Example
Sub PrintDemo ()
   Const BLUE = 1
   Const MAGENTA = 5
   Const BRIGHTWHITE = 15
   Cls                               ' Clear form.
   Width = 7200                      ' Set form width.
   Height = 5000                     ' Set form height.
   BackColor = QBColor(BLUE)         ' Set background color.
   For I% = 1 To 3
      Select Case I%
         Case 1                      ' First time.
            'Set foreground color.
            ForeColor = QBColor(MAGENTA)
            K% = 1: L% = 9: M% = 1
            Msg$ = "Visual"          ' Set message."
            CX = 0                   ' Set position variable.
         Case 2                      ' Second time.
            K% = 1: L% = 9: M% = 1
            Msg$ = "Basic"
            CX = ScaleWidth
         Case 3                      ' Third time.
            ForeColor = QBColor(BRIGHTWHITE)
            K% = 9: L% = 1: M% = -1
            Msg$ = "Visual Basic"
            CX = ScaleWidth / 2
      End Select
      For J% = K% To L% Step M%
         Select Case J%                 ' Change font size.
            Case 1: Fontsize = 8
            Case 2: Fontsize = 10
            Case 3: Fontsize = 12
            Case 4: Fontsize = 14
            Case 5: Fontsize = 18
            Case 6: Fontsize = 20
            Case 7: Fontsize = 24
            Case 8: Fontsize = 36
            Case 9: Fontsize = 48
         End Select
         If I% = 1 Then
            Offset% = 0                 ' Text on left.
         ElseIf I% = 2 Then
            Offset% = TextWidth(Msg$)   ' Text on right.
         Else
            ' Text in center.
            Offset% = TextWidth(Msg$) / 2
         End If
         CurrentX = CX - Offset%
         Print Msg$                   ' Print message.
      Next J%
      CurrentY = 0                    ' Reset to top of form.
   Next I%
End Sub
```

For further examples using the **Print** method, see the following.

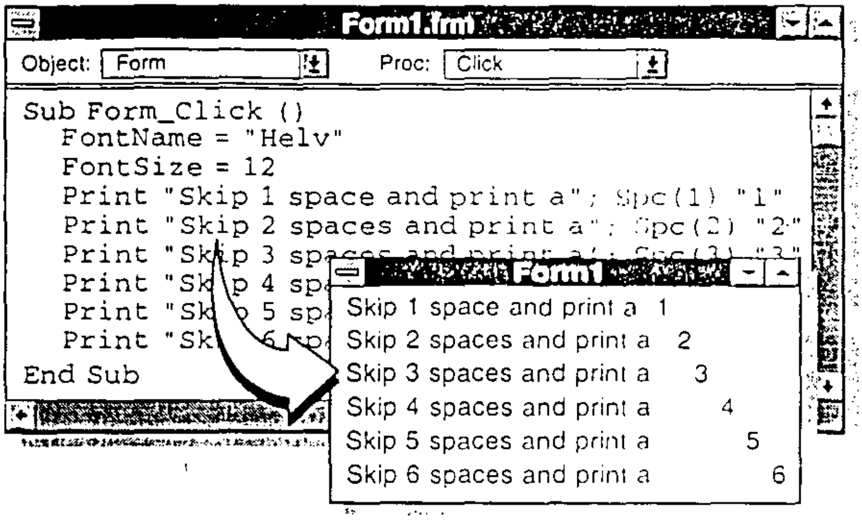
Further Examples

Application	Form	Procedure
IconWorks	ICONEDIT.FRM	DisplayMouseCoordinates
	ICONEDIT.FRM	Pic_StatusArea_Paint
	VIEWICON.FRM	Load_All_Icons
	VIEWICON.FRM	Pic_SelectedIconLabel_Paint
Employee Database	EMPREC.FRM	cmdAddDeleteUpdate_Click

∑ Print-Related Functions

- Spc Function
- Tab Function

Spc Function



```
Sub Form_Click ()
  FontName = "Helv"
  FontSize = 12
  Print "Skip 1 space and print a"; Spc(1) "1"
  Print "Skip 2 spaces and print a"; Spc(2) "2"
  Print "Skip 3 sp
  Print "Sk p 4 sp
  Print "Sk  5 sp
  Print "Sk
End Sub
```

```
Skip 1 space and print a 1
Skip 2 spaces and print a  2
Skip 3 spaces and print a   3
Skip 4 spaces and print a    4
Skip 5 spaces and print a     5
Skip 6 spaces and print a      6
```

This function inserts a specified number of blank characters in a **Print** method, starting at the current print position.

Syntax

Spc(*number%*)

The *number* argument must be an integer between 0 and 32,767 inclusive.

Important When using fixed pitch fonts (where the space allowed for each character is the same size), **Spc** is not complicated to use; but with a proportionally spaced font (such as Times New Roman®), the width of the space is always the average width of all characters in the point size of that font.

Example

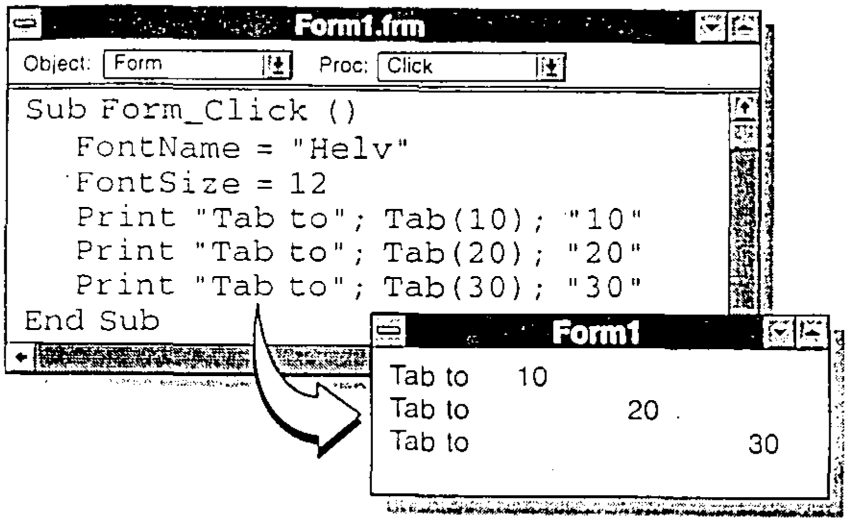
```
'Spc Function Example
Sub Form_Click ()
FontName = "Courier"
Print "         1         2         3         4         5"
Print "12345678901234567890123456789012345678901234567890"
Print "I'll skip some spaces then print an x"; Spc(20); "x"
End Sub
```

For other examples using the Spc function, see the following.

Further Examples

Application	Form	Procedure
Employee Database	EMPREC.FRM	cmdAddPrintUpdate_Click

Tab Function



```
Sub Form_Click ()
   FontName = "Helv"
   FontSize = 12
   Print "Tab to"; Tab(10); "10"
   Print "Tab to"; Tab(20); "20"
   Print "Tab to"; Tab(30); "30"
End Sub
```

```
Tab to    10
Tab to              20
Tab to                        30
```

The Tab function moves the text cursor to a specified print position when used with the **Print** method.

Syntax

Tab(*column%***)**

The parameter *column%* is an **Integer** expression that is the column number of the new print position. The leftmost print position on an output line is always 1.

For forms, the only limit to the rightmost print position is the range of the **Integer** data type.

For a complete description of the **Tab** function, see Visual Basic Help.

Using PrintForm

[*form.*]PrintForm

PrintForm

Once you have finished formatting all the output to a form, you can use the **PrintForm** method to send a bit-by-bit image of the form to a printer.

Syntax

[*form.*]**PrintForm**

This example is available in Visual Basic Help. If your machine is correctly connected to a printer, this code will send the current form to it.

Example

```
'PrintForm Method Example
Sub PrintFormDemo ()
    On Error GoTo ErrorHandler            ' Set up error handler.
    PrintForm                             ' Print form.
    Exit Sub
ErrorHandler:
    Msg$ = "The form could not be printed. "
    MsgBox Msg$                           ' Display message.
    Resume Next
End Sub
```

Further Examples

Application	Form	Procedure
Employee Database	EMPREC.FRM	cmdAddPrintUpdate_Click

Remember, the **Print** method is used to place the data on the form. The **PrintForm** method is used to send a bit-by-bit image of the form to the default printer.

You should set the AutoRedraw property of a form to True if you will print it using the **PrintForm** method. If AutoRedraw is not True, graphics drawn directly on the form using **Print**, **Line**, **Circle** and other graphic statements will not appear on the printer.

Walk Through—Effects of AutoRedraw

∑ **To see the effects of setting the AutoRedraw Property**

1. From the Walk Throughs program group, start PrintForm & AutoRedraw.
2. Click the Draw command button.

   This draws lines on the form.
3. From the AutoRedraw menu notice that AutoRedraw is set to False.
4. Drag the minimized Program Manager icon onto the form and then move it away.

   Notice the form now has a gap where the Program Manager icon erased the lines. The lines are not redrawn. If you had printed this form using the **PrintForm** method, the lines would not appear.
5. From the AutoRedraw menu choose True to set AutoRedraw to True.

   The code in this menu selection invokes the Draw button for you.
6. Drag the minimized Program Manager icon onto the form and then move it away.

   Notice the form automatically redraws the lines if necessary. If you had printed this form using the **PrintForm** method, the lines would appear.

Summary

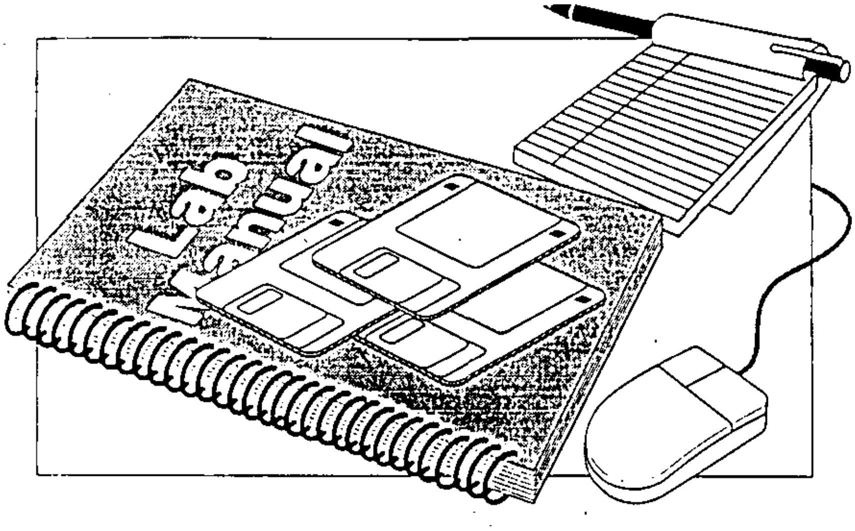
- Scenario
- Methods for Printing
- Print-Related Functions
- Using PrintForm

Objectives

In this module you learned to:

- Use the **Print** method to create output directly onto a form.
- Describe how the AutoRedraw property relates to printing directly to a form.
- Use the CurrentX and CurrentY properties of a form and the **Spc** and **Tab** functions to control the placement of output printed to a form.
- Use the **Cls** method to clear a form.
- Use the **PrintForm** method to send a bit-for-bit image of a form to the printer.

Lab Time



Go to the Getting Output portion of your lab manual.

Module 13: Data Access Using the Data Control

∑ Overview

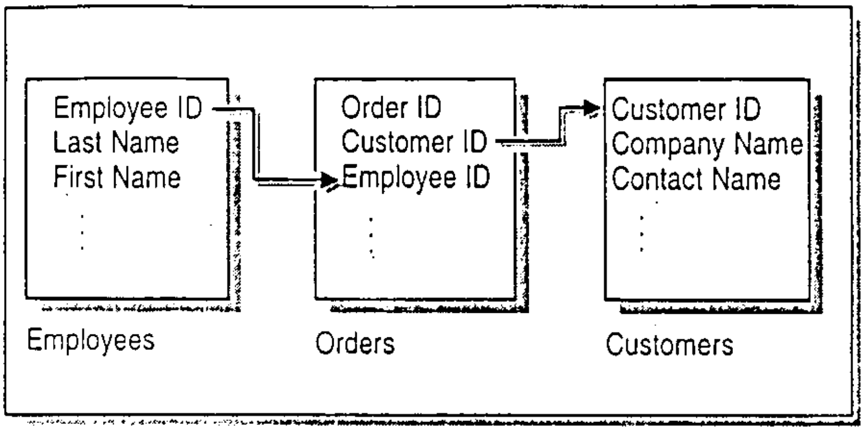
- Overview of a Database
- How Does Visual Basic Access Databases?
- The Data Control
- Binding Controls to the Data Control
- Data Control Walkthru
- Data Control Methods and Properties

Objectives

At the end of this module, you will be able to:

- Describe how Visual Basic accesses databases.
- Use the data control to view the contents of a Microsoft Access® database.

Overview of a Database



NWIND.MDB

Overview of a Database

The following information will help you understand some of the terminology and concepts associated with database structure and design.

Relational Database Objects

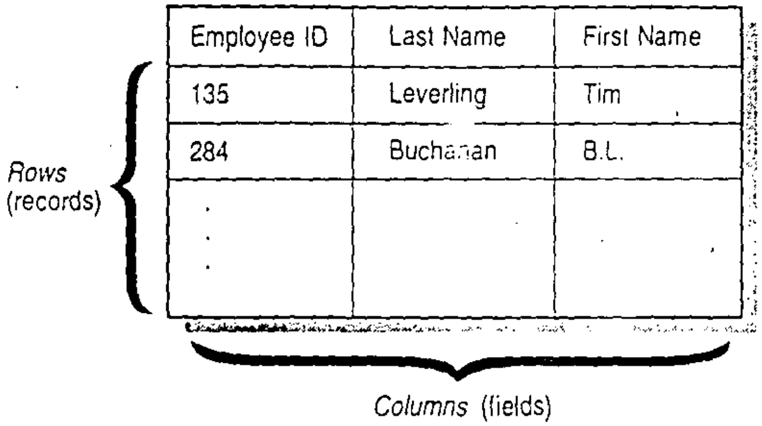
Visual Basic provides a *relational* interface to database files. Basically, a relational database is one that stores data of *tables*, made up of *columns* and *rows*. In Visual Basic, columns are referred to as *fields* and rows are referred to as *records*.

Tables

A *table* is a logical grouping of related information. For example, the Northwind Traders database has a table that lists all the employees and another table that lists all the customers.

Overview of a Table

Employees Table



Employee ID	Last Name	First Name
135	Leverling	Tim
284	Buchanan	B.L.
.		
.		
.		

Fields

Each column or *field* in a table contains a single piece of information. For example, the Employees table has fields for Employee ID, Last Name, and so forth.

Records

A row or *record* in a table contains information about a single entry in a table. For example, a record in the Employees table would have information on a particular employee. Generally, you do not want two records in a table to have the exact same data. You would not want to have two Employees with the same name and the same ID number. Most tables have a field or combination of fields that must be unique.

Indexes

To make access to the database faster, most databases use *indexes*. Database table indexes are sorted lists that are faster to search than the tables.

Structure 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. The user submits a query and the database returns all the rows that match that query.

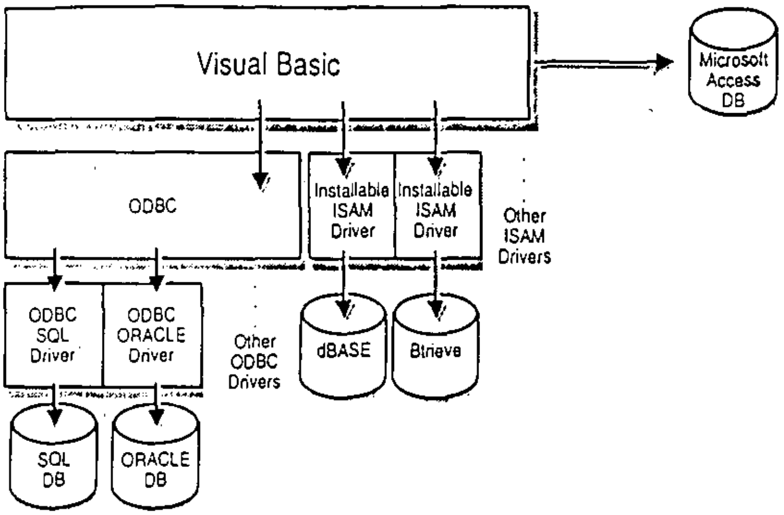
Example

```
Select [Last Name], Title From Employees Where Title = 'Sales Rep'
```

Sample Databases

In this course you will the Northwind Traders sample database. This database is included with Microsoft Access.

How Does Visual Basic Access Databases?



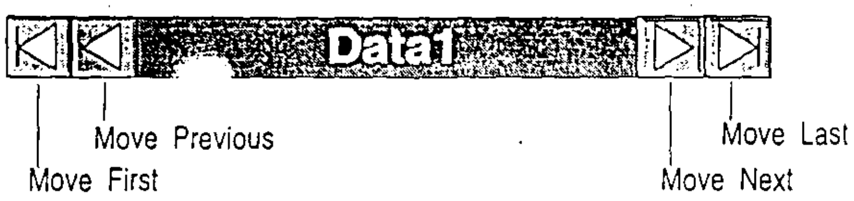
How Does Visual Basic Access Databases?

There are three types of databases that you can access from Visual Basic:

- "Native" Microsoft Access databases. These databases are accessed directly by Visual Basic.
- Indexed sequential access method (ISAM) databases—for example dBASE, Paradox®, and Btrieve® databases. Visual Basic reaches these databases through user-installable drivers that link Visual Basic to the specific databases.
- Open Database Connectivity (ODBC)–accessible databases. These include client-server database management systems (DBMSs), such as Microsoft SQL Server and ORACLE®. Visual Basic reaches these databases through the appropriate ODBC drivers.

There are various gateways that are available to connect to a mainframe database. This is typically implemented through an ODBC driver.

The Data Control



Data Control

The data control allows you to link a Visual Basic form to a database. With the data control, you can create an application that displays and updates data from a database—without writing a single line of code!

Adding the Data Control to the Toolbox

If the data control is not visible in the Toolbox, add the following line at the end of the [Visual Basic] section in the VB.INI file in the C:\WINDOWS directory:

```
DataAccess=1
```

The DatabaseName Property

The data control locates the database through the DatabaseName property. If you are connecting to a dBASE, Paradox, or Btrieve, database, set the DatabaseName property to the directory that contains the database files.

```
Data1.DatabaseName = "c:\walkthru\nwind.mdb"
```

The RecordSource Property

The RecordSource property indicates the name of a table, query or it contains the text of an SQL string.

The following example connects the data control to the Employees table:

```
Data1.RecordSource = "Employees"
```

The following example retrieves a subset of the Employees table.

```
Data1.RecordSource = "Select * from Employees where [Last Name] > 'M'"
```

Joining Two Tables

The following example joins two tables. The brackets are used with fields that have a space in the name.

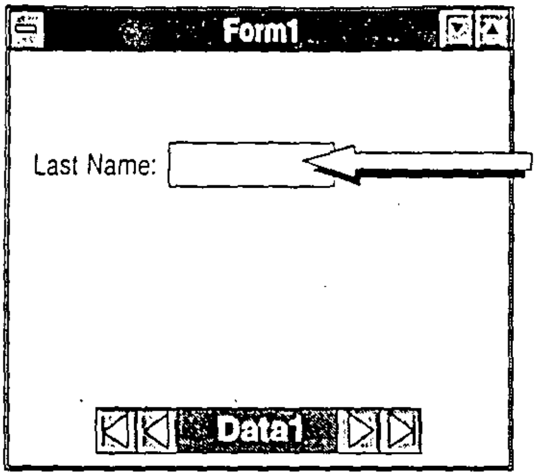
```
Data1.RecordSource =      "Select customers.[Customer Id],
                                 ∧[Contact Name], [Order Id]
                                 ∧From Customers, Orders
                                 ∧Where  Customers.[Customer Id] =
                                 ∧Orders.[Customer Id]"
```

Ordering the Records

The following example selects only some of the fields in the Employees table and orders the records by last name:

```
Data1.RecordSource = "Select [First Name], [Last Name] from
                        ∧Employees Order by [Last Name]"
```

Binding Controls to the Data Control



```
.DataSource = Data1
.DataField = 'Last Name'
```

Binding Controls to the Data Control

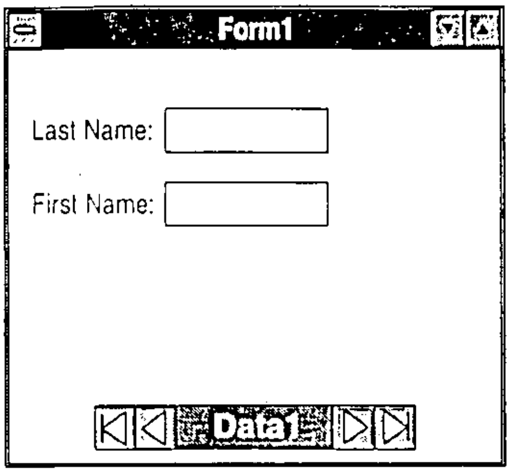
The text, picture, image, check box, masked edit, 3D panel, and 3D check box controls can be *bound* to a data control. When a control is bound to a data control, the data from the database is automatically displayed in the bound control. In addition, if the user changes the data in the bound control, those changes are automatically posted to the database as the user moves to another row.

To bind a control to a data control, set the DataSource property to the data control name and set the DataField property to a field name from the data control's table.

Example

```
'The DataSource property must be set at design time
Text1.DataSource = Data1

'The DataField property can be set at design or run time
Text1.DataField = "[Last Name]"
```

Data Control Walkthrough



Data Control Walkthrough

To get a quick overview of the data control, let's create a simple application that connects to the Microsoft Access Northwind database and browses the Employees table:

1. Start with a new project and add the data control to the form; leave the default name Data1.
2. Set the following properties for Data1:

   DatabaseName = c:\walkthru\nwind.mdb
   RecordSource = Employees

Note When you set the DatabaseName at design time, Visual Basic attempts to connect to the database. If it successfully connects to the database, it will display a list of possible values for the RecordSource property. Notice when you set the RecordSource property, you can type data in directly or choose the DOWN ARROW to display a list of possible selections.

3. Add two text boxes to the form; leave the names Text1 and Text2.
4. Set the following properties for Text1:

   DataSource = Data1
   DataField = Last Name
5. Set the following properties for Text2:

   DataSource = Data1
   DataField = First Name
6. Run your application!

(continued on following page)

7. Click the right arrow button to move forward, left arrow button to move backward.
8. Change a last name, click the right arrow button to move forward, and then click the left arrow button to move back. The record should be updated.
9. Notice you have not written a single line of code!

Data Control Methods and Properties

- Refresh Method
- Connect
- Exclusive
- ReadOnly

Refresh Method

The data control reflects modifications to existing data by other users but does not reflect records deleted or added by other users. The **Refresh** method updates the data control with the latest information from the database.

```
Data1.Refresh
```

Connect Property

The Connect property indicates the type of database that will be opened. The Connect property does not need to be set if you are connecting to a Microsoft Access database.

Database format	Database name	Connect
Microsoft Access	drive:\path\file.mdb	(none)
dBASE	drive:\path\	"dbase III;" or "dbase IV;"
Paradox	drive:\path\	"paradox;"
Btrieve	drive:\path\	"btrieve;"
ODBC	Registered data source name (server)	"odbc;dsn=*datasource*;uid=*user*;pwd=*password*" (Professional Edition only)

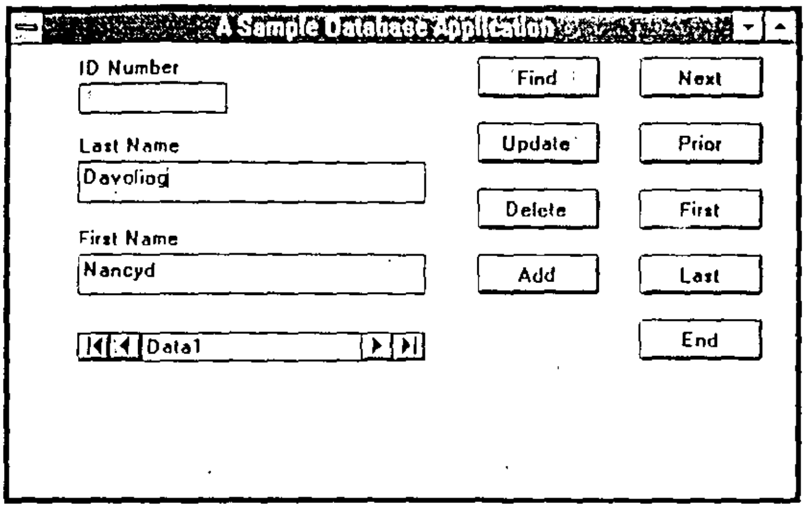
Exclusive Property

If you set the Exclusive property to **True** and then open the database, no other application will be allowed to open the database until you close the database. If another application has the database open and you attempt to open the database with Exclusive set to **True**, your application will receive a run-time error.

ReadOnly Property

you set the ReadOnly property to **True**, your application will not be allowed to write to the database.

A Sample Application



A Sample Application

Here is a sample application that adds, finds, updates and deletes records from a database.

The details on how to code this application are beyond the scope of this class. However, you may find this application useful as an example.

The demonstration program DBSAMPLE.MAK is located in the \WALKTHRU\DBSAMPLE directory.

∑ Summary

- Overview of a Database
- How Does Visual Basic Access Databases?
- The Data Control
- Binding Controls to the Data Control
- Data Control Walkthru
- Data Control Methods and Properties

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

DEPARTAMENTO DE CURSOS INSTITUCIONALES

INSTITUTO MEXICANO DEL PETROLEO

VISUAL BASIC

7- 11 NOVIEMBRE DE 1994

MATERIAL DIDACTICO

Ing. Leonardo Domínguez
Pastrana

Palacio de Minería
México, D.F.

Language Summary

Visual Basic objects include forms, controls, and special objects such as **App**, **Clipboard**, **Debug**, **Printer**, and **Screen**. Each object has an associated set of properties, events and methods. This section includes six tables:

- Table 1.1 lists related functions, statements, and methods, grouped by programming task. Items listed in a given category are often used together.
- Table 1.2 lists properties by object.
- Table 1.3 lists events by object.
- Table 1.4 lists methods by object.
- Table 1.5 lists Recordset objects and the properties that apply to each object.
- Table 1.6 lists Recordset objects and the methods that apply to each object.

Table 1.1 Functions, Statements, and Methods by Programming Task

Category	Action	Functions/Statements/Methods
Arrays	Change default lower limit	**Option Base**
	Declare and initialize	**Dim, Global, ReDim, Static**
	Find the limits	**LBound, UBound**
	Reinitialize	**Erase, ReDim**
Controlling program flow	Branch	**GoSub...Return, GoTo, On Error, On...GoSub, On...GoTo**
	Exit or pause the program	**DoEvents, End, Stop, Unload**
	Loop	**Do...Loop, For...Next, While...Wend**
	Make decisions	**If...Then...Else, Select Case**
Converting	ANSI value to string	**Chr, Chr$**
	Date to serial number	**DateSerial, DateValue**
	Decimal numbers to other	**Hex, Hex$, Oct, Oct$**
	Number to string	**Format, Format$, Str, Str$**
	One numeric data type to another	**CCur, CDbl, CInt, CLng, CSng, CStr, CVar, CVDate, Fix, Int**
	Serial number to date	**Day, Month, Weekday, Year**
	Serial number to time	**Hour, Minute, Second**
	String to ASCII value	**Asc**
	String to number	**Val**
	Time to serial number	**TimeSerial, TimeValue**

Table 1.1 Functions, Statements, and Methods by Programming Task *(continued)*

Category	Action	Functions/Statements/Methods
Copying, cutting, and pasting	Use **Clipboard** object	**Clear**, **GetData**, **GetFormat**, **GetText**, **SetData**, **SetText**
Date/time	Get current date or time	**Date**, **Date$**, **Now**, **Time**, **Time$**
	Set date or time	**Date**, **Date$**, **Time**, **Time$**
	Time a process	**Timer**
Dynamic data exchange (DDE)	Use a Visual Basic application as a DDE client	**LinkExecute**, **LinkPoke**, **LinkRequest**
	Use a Visual Basic application as a DDE server	**LinkSend**
Error trapping	Get error messages	**Error$**
	Get error-status data	**Err**, **Erl**
	Simulate run-time errors	**Error**
	Trap errors while a program is running	**On Error**, **Resume**
File I/O	Access or create a file	**Open**
	Close files	**Close**, **Reset**
	Control output appearance	**Spc**, **Tab**, **Width #**
	Copy one file to another	**FileCopy**
	Get information about a file	**EOF**, **FileAttr**, **FileDate**, **FileLen**, **FreeFile**, **Loc**, **LOF**, **Seek**
	Manage disk drives or directories	**ChDir**, **ChDrive**, **CurDir**, **CurDir$**, **MkDir**, **RmDir**
	Manage files	**Dir**, **Dir$**, **Kill**, **Lock...Unlock**, **Name**
	Read from a file	**Get**, **Input**, **Input #**, **Input$**, **Line Input #**
	Set or get file attributes	**GetAttr**, **SetAttr**
	Set read-write position in a file	**Seek**
	Write to a file	**Print #**, **Put**, **Write #**

Table 1.1 Functions, Statements, and Methods by Programming Task *(continued)*

Category	Action	Functions/Statements/Methods
Graphics	Change coordinate system	**Scale**
	Clear run-time graphics	**Cls**
	Draw shapes	**Circle, Line, PSet**
	Draw text	**Print**
	Find size of text	**TextHeight, TextWidth**
	Load or save a picture file	**LoadPicture, SavePicture**
	Work with colors	**Point, QBColor, RGB**
Manipulating objects	Arrange forms or controls on the screen	**Arrange, ZOrder**
	Direct user input to a control	**SetFocus**
	Display dialog boxes	**InputBox, InputBox$, MsgBox**
	Drag and drop	**Drag**
	Hide or show forms	**Hide, Show**
	Load or unload objects	**Load, Unload**
	Move or resize controls	**Move**
	Print forms	**PrintForm**
	Update the display	**Refresh**
	Work with list boxes and combo boxes	**AddItem, RemoveItem**
Math	General calculations	**Exp, Log, Sqr**
	Generate random numbers	**Randomize, Rnd**
	Get absolute value	**Abs**
	Get the sign of an expression	**Sgn**
	Numeric conversions	**Fix, Int**
	Trigonometry	**Atn, Cos, Sin, Tan**
Printing	Control output appearance	**Scale, Spc, Tab, TextHeight, TextWidth**
	Control printer	**EndDoc, NewPage**
	Print	**Print, PrintForm**

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.

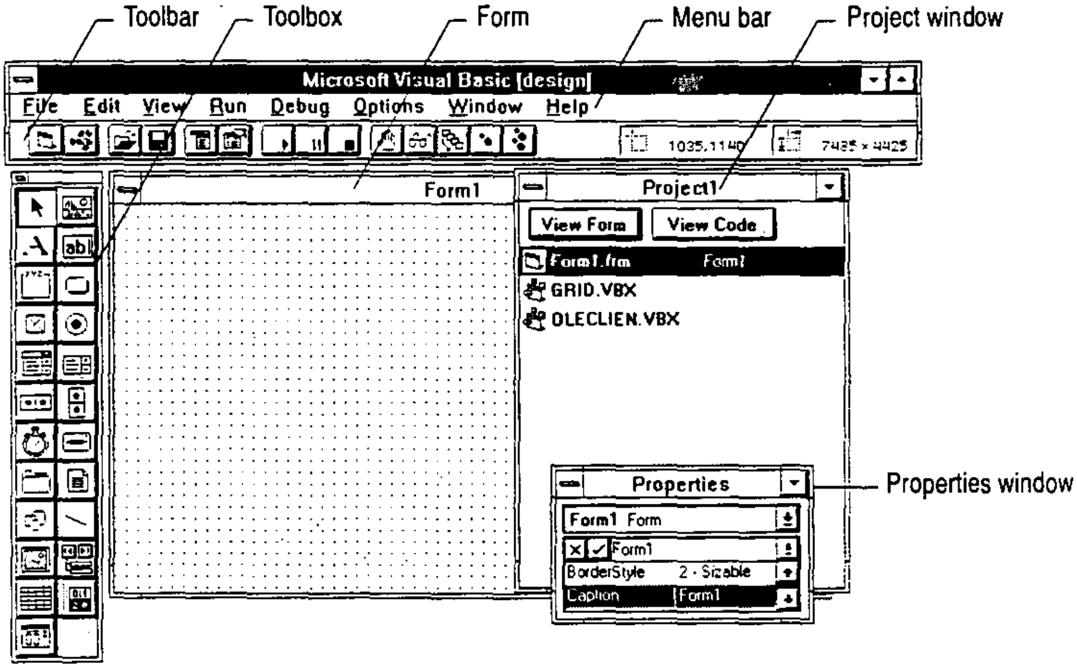


Figure 2.1 The Visual Basic Programming Environment

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.

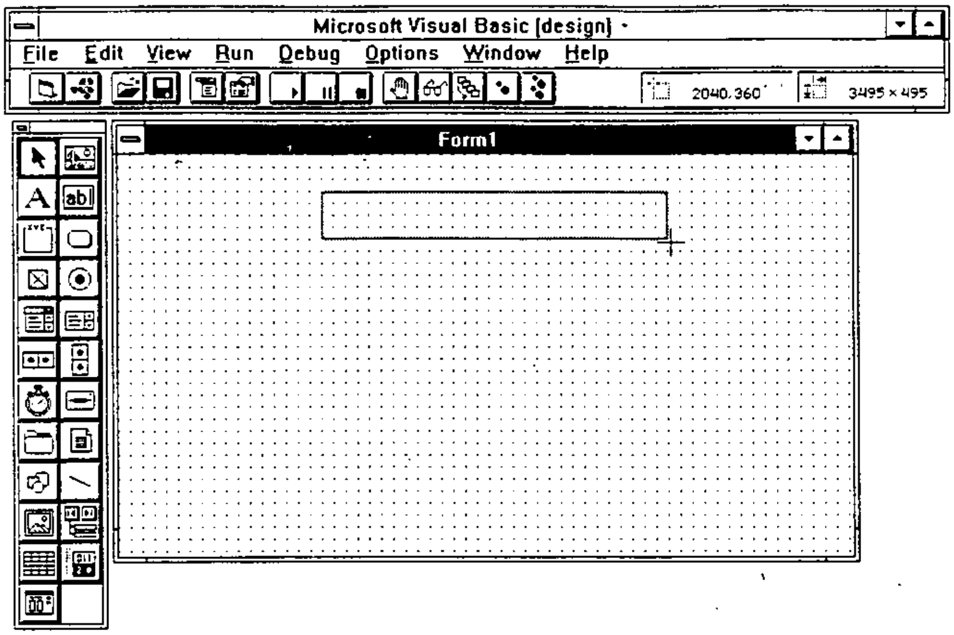

Microsoft Visual Basic [design]
File Edit View Run Debug Options Window Help
2040,360
3495 x 495
Form1

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.

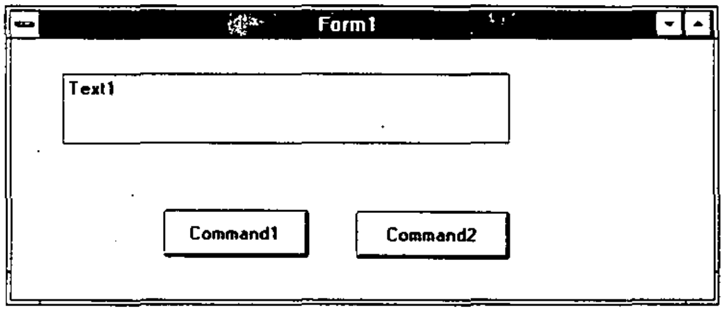
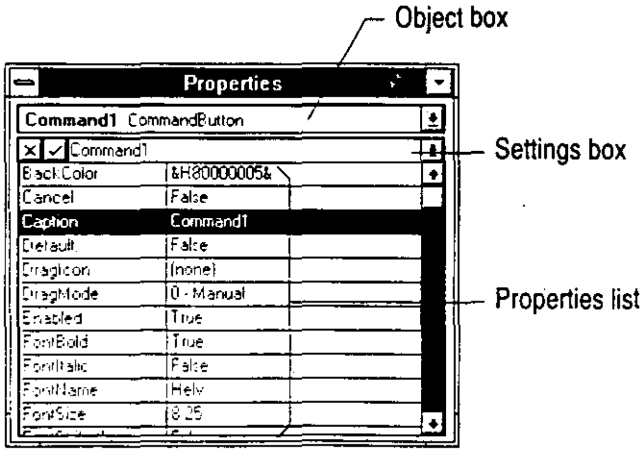


Figure 2.3 The interface for the "Hello, world!" application

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.



Command1 CommandButton

☒ ✓ Command1

BackColor	&H80000005&
Cancel	False
Caption	Command1
Default	False
DragIcon	(none)
DragMode	0 - Manual
Enabled	True
FontBold	True
FontItalic	False
FontName	Helv
FontSize	8.25

Figure 2.4 The Properties Window

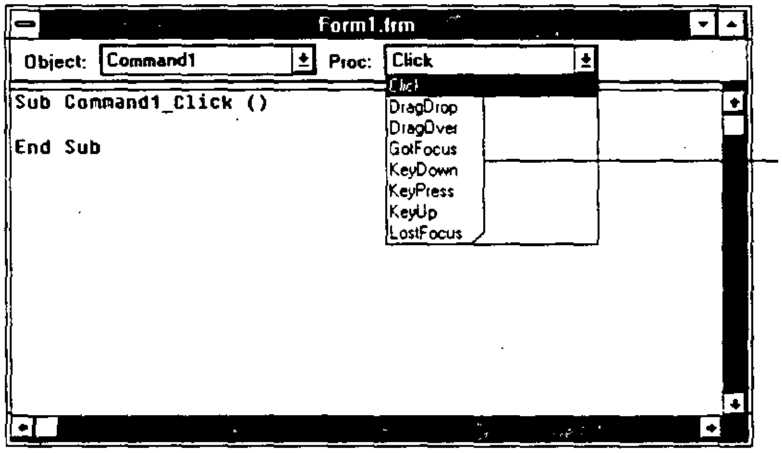
▶ To open the Code window

- 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.



Object: Command1 Proc: Click

```
Sub Command1_Click ()

End Sub
```

Events for Command1

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."

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.

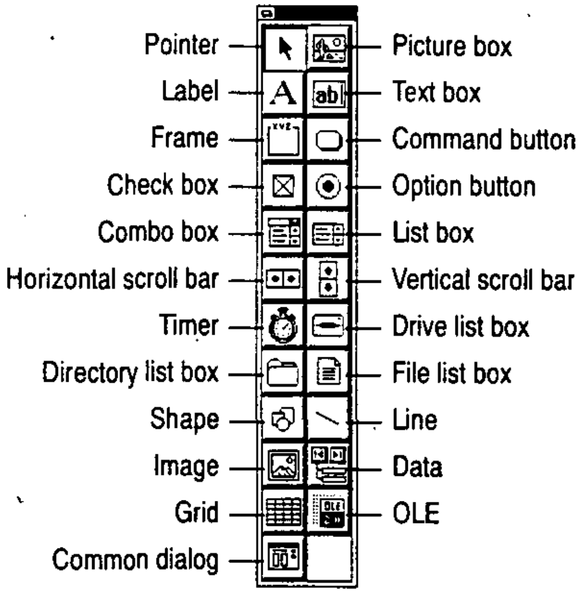


Figure 3.1 The Visual Basic Toolbox

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."
A	Label	Displays text a user cannot interact with or modify.

Icon	Control	Description
ab|	Text box	Provides an area to input or display text.
xyz	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	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.)
▴▾	Vertical scroll bar	
⏱	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

Una *palabra reservada* tiene un significado especial para Visual Basic. Son palabras reservadas las sentencias predefinidas (**For**) y los nombres de funciones (Val), métodos (**Hide**), propiedades (**Caption**) y operadores (**And**).

Tipos de datos

Una variable puede ser de alguno de los seis tipos siguientes:

Tipo	Descripción	Carácter de declaración del tipo	Rango
Integer	Entero (2 bytes)	%	-32768 a 32767
Long	Entero largo (4 bytes)	&	-2147483648 a 2147483647
Single (por defecto)	Real simple precisión (4 bytes)	!	-3.37E+38 a 3.37E+38
Double	Real doble precisión (8 bytes)	#	-1.67D+308 a 1.67D+308
Currency	Número con punto decimal fijo	@	-9.22E+14 a 9.22E+14
String	Cadena de caracteres	$	

Antes de utilizar una variable, hay que declarar su tipo. Una forma de hacer ésto es utilizando la sentencia **Dim** (o una de las palabras **Global** o **Static**). Cualquier declaración de éstas inicializa las variables numéricas con el valor cero y las variables alfanuméricas con el carácter nulo. Por ejemplo,

```
Dim I As Integer
Dim R As Double
Dim Nombre As String
Dim Etiqueta As String * 10
Dim F As Currency
Dim L As Long, X As Currency
```

Las sentencias anteriores declaran *I* como una variable entera, *R* como una variable real de precisión doble, *Nombre* como una variable para contener una cadena de caracteres de longitud variable, *Etiqueta* como una cadena de caracteres de longitud fija (10 caracteres), *F* como una variable fraccionaria, *L* como una variable entera larga, y *X* como una variable fraccionaria. Observe que en una sentencia **Dim** puede realizar más de una declaración.

Otra forma de declarar una variable es utilizando los caracteres de declaración de tipo. Por ejemplo,

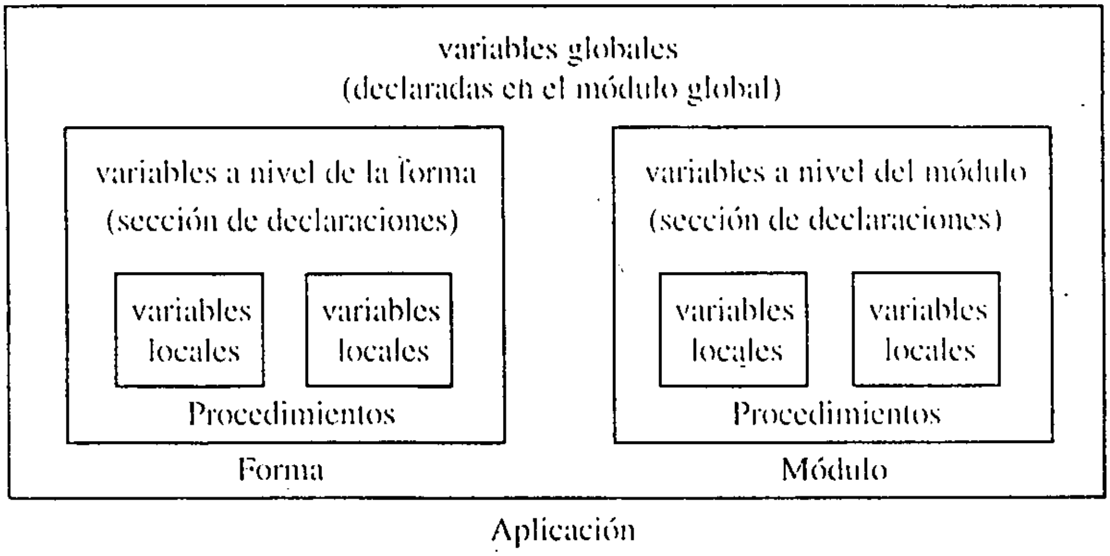
I%	Variable entera
R#	Variable real de precisión doble
Nombre$	Cadena de caracteres
F@	Variable fraccionaria

Si una variable se utiliza y no se declara se asume que es de tipo **Single**.

Si de una variable se sabe que nunca va a contener un valor fraccionario, es mejor declararla como entera, ya que las operaciones con enteros son más rápidas. En caso contrario, si el valor no va a tener más de 4 dígitos decimales y no más de 14 dígitos enteros, es conveniente declararla como fraccionaria (**Currency**). En las variables de tipo **Currency** no tiene lugar el error producido en la conversión entre las bases 2 y 10, que sí tiene lugar cuando la variable es de tipo **Single** o **Double**.

Cuando una variable numérica de un tipo se asigna a otra variable numérica de un tipo diferente, Visual Basic realiza la conversión correspondiente.

Ámbito de las variables


variables globales
(declaradas en el módulo global)
variables a nivel de la forma
(sección de declaraciones)
variables locales
variables locales
Procedimientos
Forma
variables a nivel del módulo
(sección de declaraciones)
variables locales
variables locales
Procedimientos
Módulo
Aplicación

Variables globales

Una *variable global* puede ser accedida desde cualquier parte de la aplicación. Para hacer que una variable sea global, hay que declararla en el *módulo global* de la aplicación. Este módulo no admite código, solo admite declaraciones. El nombre dado a este módulo por defecto es *Global.bas*. Para editarlo, selecciónelo en la ventana *Project* y haga clic en el botón *View Code*.

Para declarar en el módulo *Global.bas* una variable global, utilice la palabra clave **Global** en lugar de **Dim**. Por ejemplo,

```
Global var1_global As Double, var2_global As String
```

Variables con el mismo nombre en diferentes niveles

Una variable local, otra a nivel de la forma o del módulo y otra global pueden tener el mismo nombre, pero no son la misma variable. La regla para estos casos es que el procedimiento siempre utiliza la variable de nivel más cercano (local, forma o módulo y global).

Si una variable aparece en un procedimiento y no está explícitamente declarada es por defecto local. Para asegurarse de que la variable es local, es mejor declararla explícitamente.

OPERADORES

La tabla que se muestra a continuación presenta el conjunto de operadores que soporta Visual Basic colocados de mayor a menor prioridad. Los operadores que aparecen sobre una misma línea tienen igual prioridad. Las operaciones entre paréntesis se evalúan primero, ejecutándose primero los paréntesis más internos.

Tipo	**Operación**	**Operador**
Aritmético	Exponenciación	^
	Cambio de signo	–
	Multiplicación y división	*, /
	División entera	\
	Resto de una división entera	**Mod**
	Suma y resta	+, –
Relacional	Igual, distinto, mayor que, ...	=, <>, >, >=, <, <=

14

Tipo	Operación	Operador
Lógico (manejo de bits)	Negación	**Not**
	And	**And**
	Or inclusiva	**Or**
	Or exclusiva	**Xor**
	Equivalencia (opuesto a Xor)	**Eqv**
	Implicación (verdad si primer operando falso y segundo operando verdadero)	**Imp**

SENTENCIAS

Una sentencia es una línea de texto que indica una o más operaciones a realizar. Una línea puede tener varias sentencias separadas unas de otras por dos puntos.

```
Total = cantidad * precio: suma = suma + total
```

La sentencia más común en Visual Basic es la sentencia de asignación. Su forma general es,

variable = *expresión*

la cual indica que el valor que resulte de evaluar la *expresión* tiene que ser almacenado en la *variable* especificada. Si la expresión es numérica la variable tiene que ser también numérica y si la expresión es alfanumérica la variable tiene que ser también alfanumérica. Por ejemplo,

```
[illegible]
Intereses = Capital * TantoPorCiento / 100
Mensaje = "La operación es correcta"
```

PROPIEDADES

Recordar que un objeto (forma o control) tiene asociadas varias propiedades. Para referirse a una propiedad de un objeto se utiliza la forma,

objeto.propiedad

Por ejemplo, supongamos el objeto *Texto* y su propiedad *Text*. Las siguientes operaciones serían válidas

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.

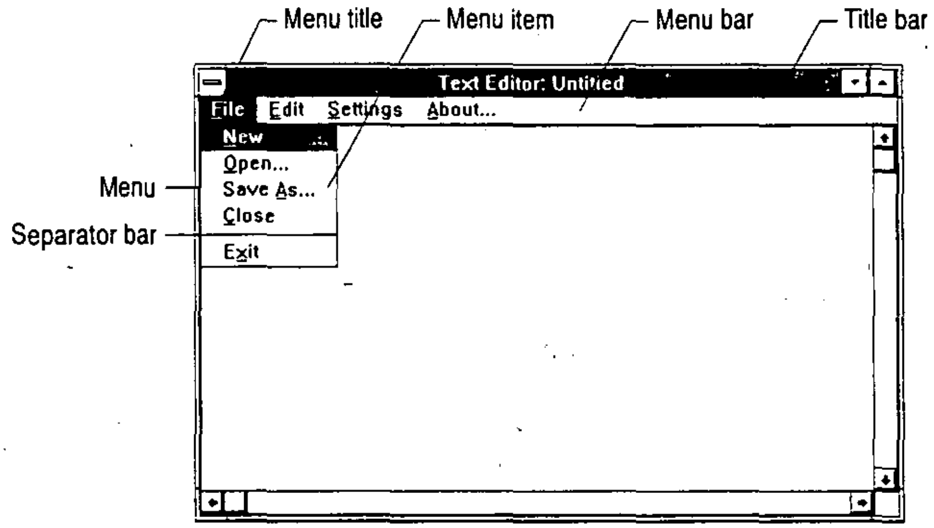


Figure 4.1 The elements of a menu interface on a Visual Basic form

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.

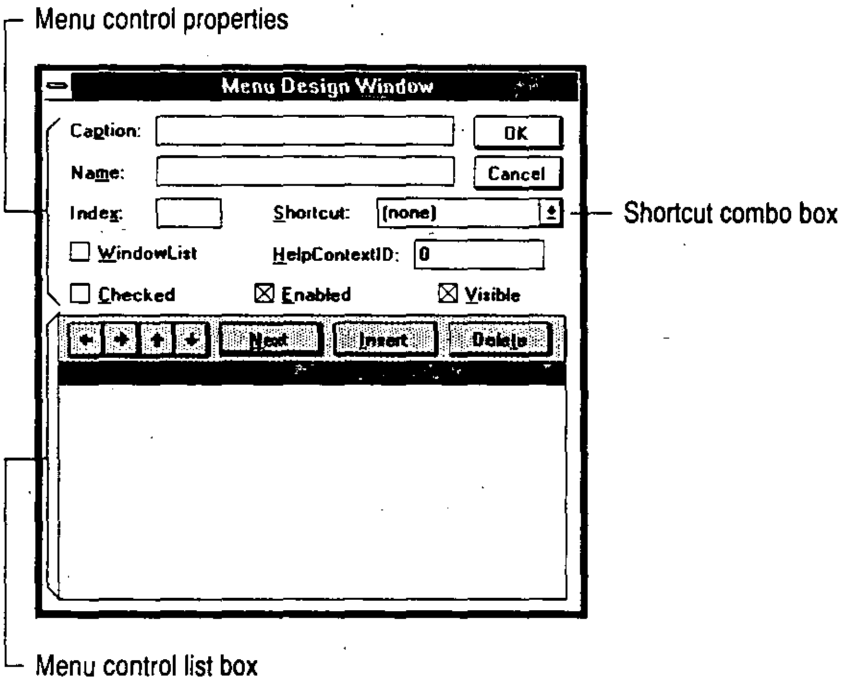
▶ **To display the Menu Design window**

- 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.



Caption:
Name:
Index:
Shortcut: (none)
☐ WindowList
HelpContextID: 0
☐ Checked
☒ Enabled
☒ Visible

Figure 4.2 A Menu Design window

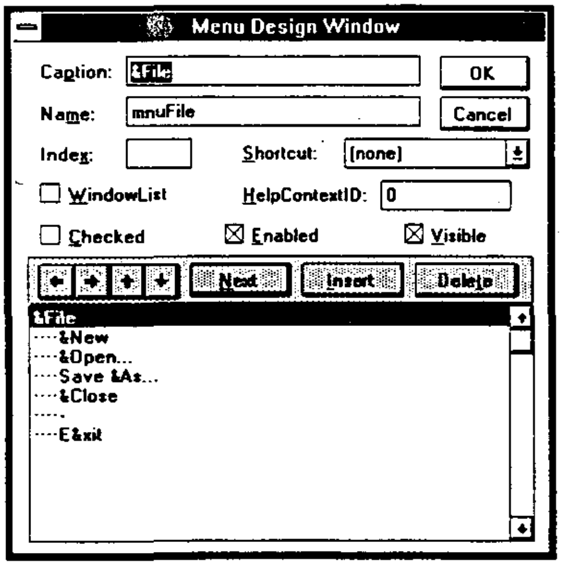
All the menu control design-time properties are shown in the Menu Design window. The two most important properties for menu controls are:

- 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.



Caption: &File

Name: mnuFile

Index:

Shortcut: (none)

□ WindowList

HelpContextID: 0

□ Checked ☒ Enabled ☒ Visible

```
&File
····&New
····&Open...
····Save &As...
····&Close
····-
····E&xit
```

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.

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.

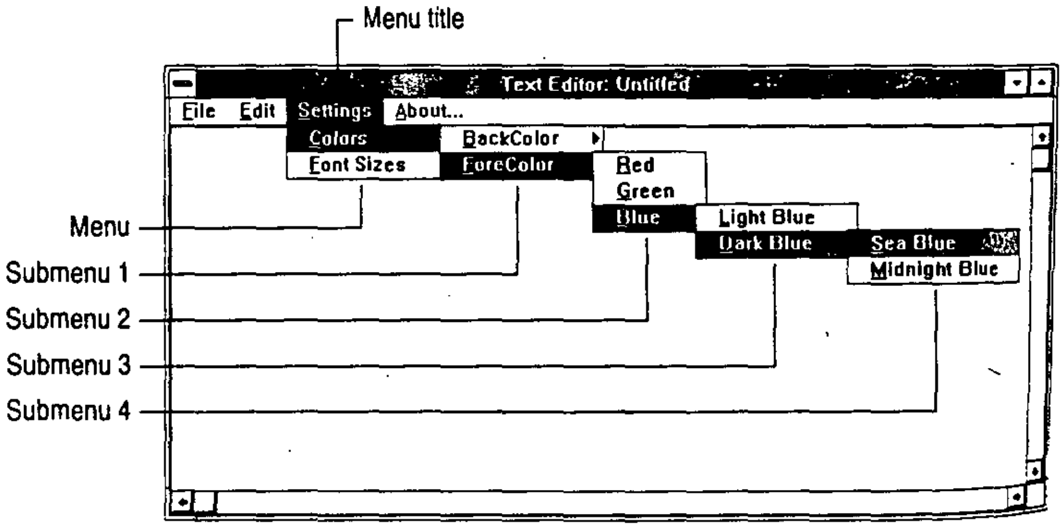


Figure 4.4 A menu interface with four levels of submenus

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.

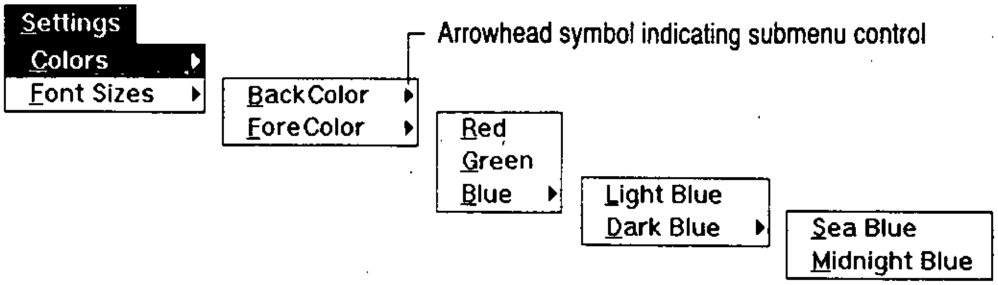


Figure 4.5 Visual cues indicating submenus

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.

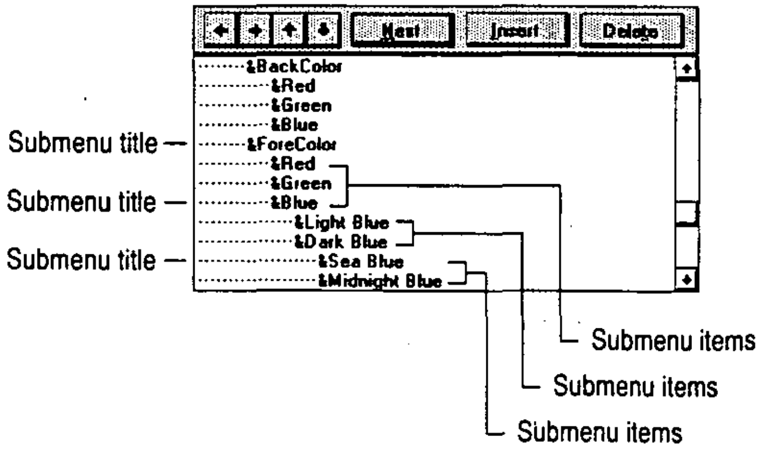


Figure 4.6 Submenu titles and submenu items in the menu control list box

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"
```

22

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
    Timerl.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
```

24

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

26

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
```

28

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

29

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
```

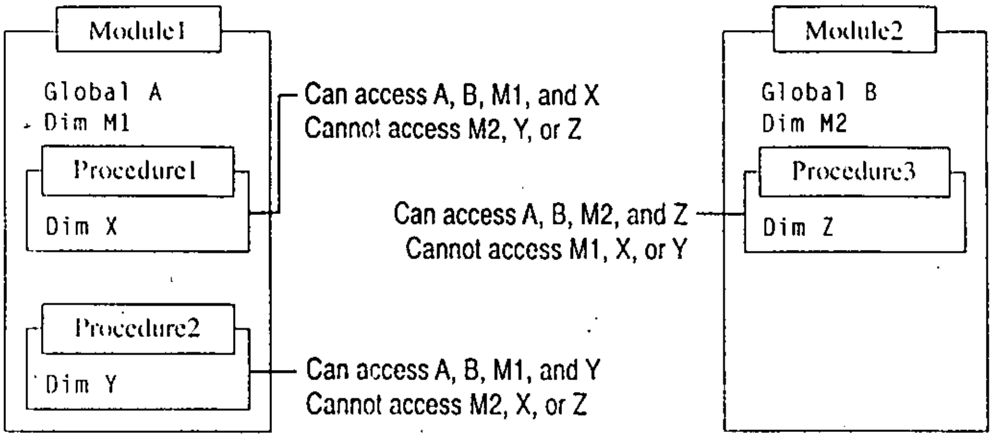

Module1
Global A
Dim M1
Procedure1
Dim X
Procedure2
Dim Y
Can access A, B, M1, and X
Cannot access M2, Y, or Z
Can access A, B, M2, and Z
Cannot access M1, X, or Y
Can access A, B, M1, and Y
Cannot access M2, X, or Z
Module2
Global B
Dim M2
Procedure3
Dim Z

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
```

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

BLOAD

BOF

BSAVE

CALLS

CDECL

CHAIN

CHECKPOINT

COLOR

COM

COMMON

CREATEINDEX

CSRLIN

CVI

CVC

CVD

CVDMBF

CVL

CVS

CVSMBF

DELETEINDEX

DELETETABLE

DRAW

ERDEV

ERDEV$

EVENT

FIELD

FILES

FN

FRE

GETINDEX$

INKEY$

INP

INSERT

IOCTL

IOCTL$ 32

ISAM

KEY

LIST

LOCATE

LPOS

LPRINT

MKC$

MKDMBF$

MKI$

MKL$

MKS$

MKSMBF$

OUT

PAINT

PALETTE

PCOPY

PEEK

PEN

PLAY

PMAP

POKE

POS

PRESET

RUN

SADD

SAVEPOINT

SCREEN

SEEKEQ

SEEKGE

SEEKGT

SEG

SETINDEX

SETMEM

SETUEVENT

SIGNAL

SLEEP

SOUND

SSEG

SSEGADD

STACK

STICK

STRIG

SWAP

SYSTEM

TEXTCOMP

TRON

TROFF

UEVENT

UPDATE

VARPTR

VARPTR$

VARSEG

VIEW

WAIT

WINDOW

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.

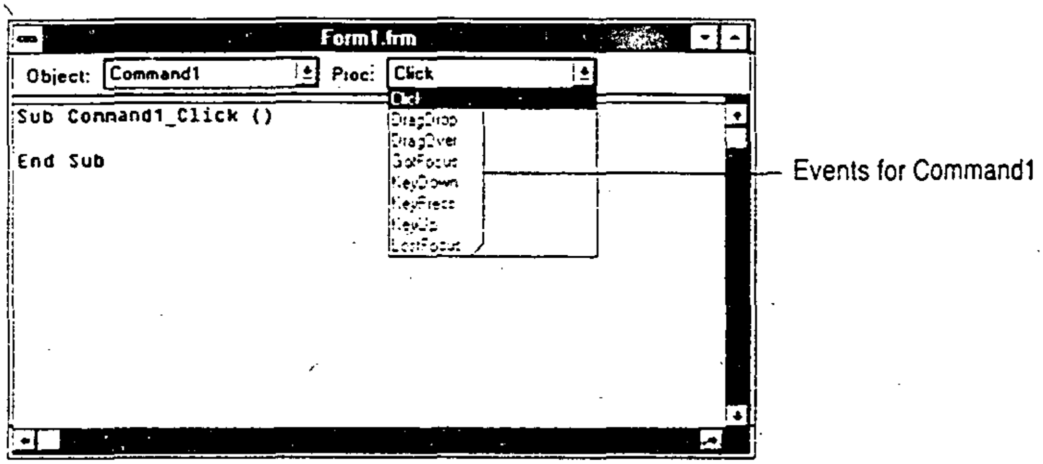
▶ To open the Code window

- 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.



```
Sub Command1_Click ()

End Sub
```

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."

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

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
II	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.