Criando uma Interface de Usuário In-depth coverage of interface styles, forms, menus, toolbars, and more.

A interface de usuário é talvez a parte mais importante de uma aplicação; é certamente o mais visível. Para os usuários, a interface é a aplicação; eles provavelmente não estão atentos do código que está executando atrás das cenas. Não importa quanto tempo e esforço que você investiu em escrever e aperfeiçoar seu código, o usabilidade de sua aplicação depende da interface.

Quando você projeta uma aplicação, várias decisões precisam de ser feitas relativo à interface. Você deveria usar um único-documento ou estilo de múltiplo-documento? De quantas formas diferentes você precisará? Que comandos incluirão seus menus, e você usará barras de ferramentas para duplicar funções de menu? Como as caixas de diálogo vão interagir com o usuário? De quanta ajuda você precisa prover?

Antes de você começar a projetar a interface de usuário, você precisa pensar no propósito da aplicação. O desígnio para uma aplicação primária que estará em uso constante deveria ser diferente de um que é ocasionalmente só usado para períodos pequenos de tempo. Uma aplicação com o propósito primário de exibir informação tem exigências diferentes que a pessoa obtenha informação.

A assitência intencional também deveria influenciar seu projeto. Uma aplicação apontada a um usuário de começo exige simplicidade em seu projeto, enquanto um para usuários experientes pode ser mais complexo. Outras aplicações usadas por sua assistência designada podem influenciar as expectativas deles/delas para o comportamento de uma aplicação. Se você planeja distribuir internacionalmente, idioma e cultura devem ser consideradas parte de seu projeto.

Projetando uma interface de usuário é aproximado melhor como um processo interativo - você raramente proporá um projeto perfeito na primeira passagem. Este capítulo o apresenta ao processo de projetar uma interface em Visual Básico e provê uma introdução para as ferramentas que você precisa para criar uma grande aplicação para seus usuários.

Interface Styles A discussion of MDI, SDI, and other interface styles.

If you've been using Windows-based applications for a while, you've probably noticed that not all user interfaces look or behave the same. There are two main styles of user interface: the *single-document interface* (*SDI*) and the *multiple-document interface* (*MDI*). An example of the SDI interface is the WordPad application included with Microsoft Windows (Figure 6.1). In WordPad, only a single document may be open; you must close one document in order to open another.

Figure 6.1 WordPad, a single-document interface (SDI) application

Applications such as Microsoft Excel and Microsoft Word for Windows are MDI interfaces; they allow you to display multiple documents at the same time, with each document displayed in its own window (Figure 6.2). You can recognize a MDI application by the inclusion of a Window menu item with submenus for switching between windows or documents.

Figure 6.2 Microsoft Excel, a multiple-document interface (MDI) application

In determining which interface style is best, you need to look at the purpose of the application. An application for processing insurance claims might lend itself to the MDI style — a clerk is likely to be working on more than one claim at a time or might need to compare two claims. On the other hand, a calendar application would be best suited to the SDI style — it's not likely that you would need more than one calendar open at a time; in the rare event that you did, you could open a second instance of the SDI application.

The SDI style is the more common; most of the examples in the *Programmer's Guide* assume an SDI application. There are a number of considerations and techniques unique to creating MDI applications, which are addressed in "Multiple-Document Interface (MDI) Applications" later in this chapter.

In addition to the two most common interface styles, SDI and MDI, a third interface style is becoming more popular: the *explorer-style* interface (Figure 6.3). The explorer-style interface is a single window containing two *panes* or regions, usually consisting of a tree or hierarchical view on the left and a display area on the right, as in the Microsoft Windows Explorer. This type of interface lends itself to navigating or browsing large numbers of documents, pictures, or files.

Figure 6.3   The Windows Explorer, an explorer-style interface

In addition to the MDI and SDI application examples that accompany this chapter, the Application Wizard provides a good way to compare the different interface styles. You can use the Wizard to generate a framework for each style and view the forms and code that it generates.

For More Information   To learn more about MDI applications, see "Multiple-Document Interface (MDI) Applications." The basics of working with forms are covered in "Forms, Controls, and Menus." For information on accessing the Application Wizard, see "Using Wizards and Add-Ins" in "Managing Projects."

Multiple-Document Interface (MDI) Applications Techniques for designing MDI applications.

The multiple-document interface (MDI) allows you to create an application that maintains multiple forms within a single container form. Applications such as Microsoft Excel and Microsoft Word for Windows have multiple-document interfaces.

An MDI application allows the user to display multiple documents at the same time, with each document displayed in its own window. Documents or *child windows* are contained in a *parent window*, which provides a workspace for all the child windows in the application. For example, Microsoft Excel allows you to create and display multiple-document windows of different types. Each individual window is confined to the area of the Excel parent window. When you minimize Excel, all of the document windows are minimized as well; only the parent window's icon appears in the task bar.

A child form is an ordinary form that has its MDIChild property set to True. Your application can include many MDI child forms of similar or different types.

At run time, child forms are displayed within the *workspace* of the MDI parent form (the area inside the form's borders and below the title and menu bars). When a child form is minimized, its icon appears within the workspace of the MDI form instead of on the taskbar, as shown in Figure 6.4.

Figure 6.4   Child forms displayed within the workspace of the MDI form

Note   Your application can also include standard, non-MDI forms that are not contained in the MDI form. A typical use of a standard form in an MDI application is to display a modal dialog box.

A MDI form is similar to an ordinary form with one restriction. You can't place a control directly on a MDI form unless that control has an Align property (such as a picture box control) or has no visible interface (such as a timer control).

Creating an MDI Application

Use the following procedure to create an MDI form and its child forms.

To create an MDI application

1. Create an MDI form.

From the **Project** menu, choose **Add MDI Form**.

Note   An application can have only one MDI form. If a project already has an MDI form, the Add MDI Form command on the Project menu is unavailable.

2. Create the application's child forms.
To create an MDI child form, create a new form (or open an existing one) and set its MDIChild property to True.

To learn more about MDI applications, see the following topics:

- **Working with MDI Child Forms at Design Time** A discussion of the design time behavior of child forms.

At design time, child forms are not restricted to the area inside the MDI form. You can add controls, set properties, write code, and design the features of child forms just as you would with any other Visual Basic form.

You can determine whether a form is an MDI child by looking at its MDIChild property, or by examining the Project Explorer. If the form's MDIChild property is set to True, it is a child form. Visual Basic displays special icons in the Project Explorer for the MDI and MDI child forms, as shown in Figure 6.5.

Figure 6.5 Icons in the Project Explorer identify MDI child, standard, and MDI forms

- **Run-Time Features of MDI Forms** A discussion of the run time behavior of child forms.

At run time, an MDI form and all of its child forms take on special characteristics:

- All child forms are displayed within the MDI form's workspace. The user can move and size child forms like any other form; however, they are restricted to this workspace.
- When a child form is minimized, its icon appears on the MDI form instead of the taskbar. When the MDI form is minimized, the MDI form and all of its child forms are represented by a single icon. When the MDI form is restored, the MDI form and all the child forms are displayed in the same state they were in before being minimized.
- When a child form is maximized, its caption is combined with the caption of the MDI form and is displayed in the MDI form's title bar (see Figure 6.6).
- By setting the AutoShowChildren property, you can display child forms automatically when forms are loaded (True), or load child forms as hidden (False).
- The active child form's menus (if any) are displayed on the MDI form's menu bar, not on the child form.

Figure 6.6 A child form caption combined with the caption of an MDI form

- **The MDI NotePad Application** Details of the MDI Notepad sample application.

The MDI NotePad sample application is a simple text editor similar to the NotePad application included with Microsoft Windows. The MDI NotePad application, however, uses a multiple-document interface (MDI). At run time, when the user requests a new document (implemented with the New command on the application's File menu), the application creates a new instance of the child form. This allows the user to create as many child forms, or documents, as necessary.

To create a document-centered application in Visual Basic, you need at least two forms — an MDI form and a child form. At design time, you create an MDI form to contain the application and a single child form to serve as a template for the application's document.

To create your own MDI NotePad application

1. From the **File** menu, choose **New Project**.
2. From the **Project** menu, choose **Add MDI Form** to create the container form.
The project should now contain an MDI form (MDIForm1) and a standard form (Form1).
3. Create a text box (Text1) on Form1.
4. Set properties for the two forms and the text box as follows.

Object	Property	Setting

MDIForm1		Caption	MDI NotePad
Form1	Caption		
MDIChild		Untitled	
True			
Text1	MultiLine		
Text			
Left			
Top	True		
(Empty)			
0			
0			

5. Using the **Menu Editor** (from the **Tools** menu), create a File menu for MDIForm1.

Caption	**Name**	**Indented**
&File	mnuFile	No
&New	mnuFileNew	Yes

6. Add the following code to the mnuFileNew_Click procedure:

```
Private Sub mnuFileNew_Click ()
      ' Create a new instance of Form1, called NewDoc.
      Dim NewDoc As New Form1
      ' Display the new form.
      NewDoc.Show
End Sub
```

This procedure creates and then displays a new instance (or copy) of Form1, called NewDoc. Each time the user chooses New from the File menu, an exact duplicate (instance) of Form1 is created, including all the controls and code that it contains.

7. Add the following code to the Form_Resize procedure for Form1:

```
Private Sub Form_Resize ()
      ' Expand text box to fill the current child form.
      Text1.Height = ScaleHeight
      Text1.Width = ScaleWidth
End Sub
```

The code for the Form_Resize event procedure, like all the code in Form1, is shared by each instance of Form1. When several copies of a form are displayed, each form recognizes its own events. When an event occurs, the code for that event procedure is called. Because the same code is shared by each instance, you might wonder how to reference the form that has called the code — especially since each instance has the same name (Form1). This is discussed in "Working with MDI Forms and Child Forms," later in this chapter.

8. Press F5 to run the application.

Tip The Mdinote.vbp sample application contains examples of many MDI techniques besides those mentioned in this chapter. Take some time to step through the example code to discover these techniques. The Sdinote.vbp sample application is an implementation of the same application converted to the SDI style; compare the two samples to learn the differences between MDI and SDI techniques.

- **Working with MDI Forms and Child Forms** A discussion of the interaction between MDI parent and child forms.

When users of your MDI application open, save, and close several child forms in one session, they should be able to refer to the active form and maintain state information on child forms.

This topic describes coding techniques you can use to specify the active child form or control, load and unload MDI and child forms, and maintain state information for a child form.

Specifying the Active Child Form or Control

Sometimes you want to provide a command that operates on the control with the focus on the currently active child form. For example, suppose you want to copy selected text from the child form's text box onto the Clipboard. In the Mdinote.vbp sample application, the Click event of the Copy item on the Edit menu calls EditCopyProc, a procedure that copies selected text onto the Clipboard.

Because the application can have many instances of the same child form, EditCopyProc needs to know which form to use. To specify this, use the MDI form's ActiveForm property, which returns the child form that has the focus or that was most recently active.

Note At least one MDI child form must be loaded and visible when you access the ActiveForm property, or an error is returned.

When you have several controls on a form, you also need to specify which control is active. Like the ActiveForm property, the ActiveControl property returns the control with the focus on the active child form. Here's an example of a copy routine that can be called from a child form menu, a menu on the MDI form, or a toolbar button:

```
Private Sub EditCopyProc ()
      ' Copy selected text onto Clipboard.
      ClipBoard.SetText _
            frmMDI.ActiveForm.ActiveControl.SelText
End Sub
```

If you're writing code that will be called by multiple instances of a form, it's a good idea to *not* use a form identifier when accessing the form's controls or properties. For example, refer to the height of the text box on Form1 as `Text1.Height` instead of `Form1.Text1.Height`. This way, the code always affects the current form.

Another way to specify the current form in code is to use the Me keyword. You use Me to reference the form whose code is currently running. This keyword is useful when you need to pass a reference to the current form instance as an argument to a procedure.

For More Information For information on creating multiple instances of a form using the New keyword with the Dim statement, see "Introduction to Variables, Constants and Data Types" in "Programming Fundamentals" and "Dim Statement" in the *Language Reference* in Books Online.

Loading MDI Forms and Child Forms

When you load a child form, its parent form (the MDI form) is automatically loaded and displayed. When you load the MDI form, however, its children are not automatically loaded.

In the MDI NotePad example, the child form is the default startup form, so both the child and MDI forms are loaded when the application is run. If you change the startup form in the MDI NotePad application to frmMDI (on the General tab of Project Properties) and then run the application, only the MDI form is loaded. The first child form is loaded when you choose New from the File menu.

You can use the AutoShowChildren property to load MDI child windows as hidden, and leave them hidden until you display them using the Show method. This allows you to update various details such as captions, position, and menus before a child form becomes visible.

You can't show an MDI child form or the MDI form modally (using the Show method with an argument of vbModal). If you want to use a modal dialog box in an MDI application, use a form with its MDIChild property set to False.

Setting Child Form Size and Position

When an MDI child form has a sizable border (BorderStyle = 2), Microsoft Windows determines its initial height, width, and position when it is loaded. The initial size and position of a child form with a sizable border depends on the size of the MDI form, not on the size of the child form at design time. When an MDI child form's border is not sizable (BorderStyle = 0, 1, or 3), it is loaded using its design-time Height and Width properties.

If you set AutoShowChildren to False, you can change the position of the MDI child after you load it, but before you make it visible.
For More Information See "AutoShowChildren Property and "Show Method" in the *Language Reference* on Books Online.

Maintaining State Information for a Child Form

A user deciding to quit the MDI application must have the opportunity to save work. To make this possible, the application needs to be able to determine, at all times, whether the data in the child form has changed since the last time it was saved.
You can do this by declaring a public variable on each child form. For example, you can declare a variable in the Declarations section of a child form:

```
Public boolDirty As Boolean
```

Each time the text changes in Text1, the child form's text box Change event sets `boolDirty` to True. You can add this code to indicate that the contents of Text1 have changed since the last time it was saved:

```
Private Sub Text1_Change ()
      boolDirty = True
End Sub
```

Conversely, for each time the user saves the contents of the child form, the text box's Change event sets `boolDirty` to False to indicate that the contents of Text1 no longer need to be saved. In the following code, it is assumed that there is a menu command called Save (mnuFileSave) and a procedure called FileSave that saves the contents of the text box:

```
Sub mnuFileSave_Click ()
      ' Save the contents of Text1.
      FileSave
      ' Set the state variable.
      boolDirty = False
End Sub
```

Unloading MDI Forms with QueryUnload

The `boolDirty` flag becomes useful when the user decides to exit the application. This can occur when the user chooses Close from the MDI form's Control menu, or through a menu item you provide, such as Exit on the File menu. If the user closes the application using the MDI form's Control menu, Visual Basic will attempt to unload the MDI form.
When an MDI form is unloaded, the QueryUnload event is invoked first for the MDI form and then for every child form that is open. If none of the code in these QueryUnload event procedures cancels the Unload event, then each child is unloaded and finally, the MDI form is unloaded.
Because the QueryUnload event is invoked before a form is unloaded, you can give the user the opportunity to save a form before unloading it. The following code uses the `boolDirty` flag to determine if the user should be prompted to save the child before it is unloaded. Notice that you can access the value of a public form-level variable anywhere in the project. This code assumes that there is a procedure, FileSave, that saves the contents of Text1 in a file.

```
Private Sub mnuFExit_Click()
      ' When the user chooses File Exit in an MDI
      ' application, unload the MDI form, invoke
      ' the QueryUnload event for each open child.
      Unload frmMDI
      End
End Sub

Private Sub Form_QueryUnload(Cancel As Integer, _
      UnloadMode As Integer)
      If boolDirty Then
```

```
            ' Call routine to query the user and save
            ' file if necessary.
            FileSave
      End If
End Sub
```

For More Information See "QueryUnload Event" in the *Language Reference* on Books Online.

More About Forms Information about the advanced features of forms.

In addition to the basics of form design, you need to think about the beginning and end of your application. There are several techniques available for determining what the user will see when your application starts. It's also important to be aware of the processes that occur when an application is unloaded.

Setting the Startup Form

By default, the first form in your application is designated as the *startup form*. When your application starts running, this form is displayed (so the first code to execute is the code in the Form_Initialize event for that form). If you want a different form to display when your application starts, you must change the startup form.

To change the startup form

1. From the **Project** menu, choose **Project Properties**.
2. Choose the **General** tab.
3. In the **Startup Object** list box, select the form you want as the new startup form.
4. Choose **OK**.

Starting Without a Startup Form

Sometimes you might want your application to start without any form initially loaded. For example, you might want to execute code that loads a data file and then displays one of several different forms depending on what is in the data file. You can do this by creating a Sub procedure called Main in a standard module, as in the following example:

```
Sub Main()
      Dim intStatus As Integer
      ' Call a function procedure to check user status.
      intStatus = GetUserStatus
      ' Show a startup form based on status.
      If intStatus = 1 Then
            frmMain.Show
      Else
            frmPassword.Show
      End If
```

This procedure must be a Sub procedure, and it cannot be in a form module. To set the Sub Main procedure as the startup object, from the Project menu, choose Project Properties, select the General tab, and select Sub Main from the Startup Object box.

Displaying a Splash Screen on Startup

If you need to execute a lengthy procedure on startup, such as loading a large amount of data from a database or loading several large bitmaps, you might want to display a splash screen on startup. A splash screen is a form, usually displaying information such as the name of the application, copyright information, and a simple bitmap. The screen that is displayed when you start Visual Basic is a splash screen.

To display a splash screen, use a Sub Main procedure as your startup object and use the Show method to display the form:

```
Private Sub Main()
      ' Show the splash screen.
      frmSplash.Show
      ' Add your startup procedures here.
      …
      ' Show the main form and unload the splash screen.
      frmMain.Show
      Unload frmSplash
End Sub
```

The splash screen occupies the user's attention while your startup routines are executing, giving the illusion that the application is loading faster. When the startup routines are completed, you can load your first form and unload the splash screen.
In designing a splash screen, it's a good idea to keep it simple. If you use large bitmaps or a lot of controls, the splash screen itself may be slow to load.

Ending an Application

An event-driven application stops running when all its forms are closed and no code is executing. If a hidden form still exists when the last visible form is closed, your application will appear to have ended (because no forms are visible), but will in fact continue to run until all the hidden forms are closed. This situation can arise because any access to an unloaded form's properties or controls implicitly loads that form without displaying it.
The best way to avoid this problem when closing your application is to make sure all your forms are unloaded. If you have more than one form, you can use the Forms collection and the Unload statement. For example, on your main form you could have a command button named cmdQuit that lets a user exit the program. If your application has only one form, the Click event procedure could be as simple as this:

```
Private Sub cmdQuit_Click ()
      Unload Me
End Sub
```

If your application uses multiple forms, you can unload the forms by putting code in the Unload event procedure of your main form. You can use the Forms collection to make sure you find and close all your forms. The following code uses the forms collection to unload all forms:

```
Private Sub Form_Unload
      Dim i as integer
      ' Loop through the forms collection and unload
      ' each form.
      For i = 0 to Forms.Count - 1
            Unload Forms(i)
      Next
End Sub
```

There may be cases where you need to end your application without regard for the state of any existing forms or objects. Visual Basic provides the End statement for this purpose.
The End statement ends an application immediately: no code after the End statement is executed, and no further events occur. In particular, Visual Basic will not execute the QueryUnload, Unload or Terminate event procedures for any forms. Object references will be freed, but if you have defined your own classes, Visual Basic will not execute the Terminate events of objects created from your classes.
In addition to the End statement, the Stop statement halts an application. However, you should use the Stop statement only while debugging, because it does not free references to objects.
For More Information   For information on the Stop statement, see "Using Break Mode" in "Error Handling and Debugging," and "Stop Statement" in the *Language Reference* on Books Online. For information on the forms collection or freeing references to objects, see "Programming with Objects."

Using Menus in Your Application In-depth coverage of menus, including pop-up menus.

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

- **Creating Menus with the Menu Editor** How to use the Menu Editor.

You can use the Menu Editor to create new menus and menu bars, add new commands to existing menus, replace existing menu commands with your own commands, and change and delete existing menus and menu bars.

To display the Menu Editor

- From the **Tools** menu, choose **Menu Editor**.

– or –

Click the **Menu Editor** button on the toolbar.

This opens the Menu Editor, shown in Figure 6.7.

Figure 6.7 The Menu Editor

While most menu control properties can be set using the Menu Editor, all menu properties are available in the Properties 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 Editor, including Index, Checked, and NegotiatePosition, are described later in this chapter.

Using the List Box in the Menu Editor

The menu control list box (the lower portion of the Menu Editor) 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 6.7 shows the menu controls for a File menu in a typical application. 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 when the user clicks the preceding menu title.
- An indented 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.

Note A menu control cannot be a separator bar if it is a menu title, has submenu items, is checked or disabled, or has a shortcut key.

To create menu controls in the Menu Editor

1. Select the form.

2. From the **Tools** menu, choose **Menu Editor**.

– or –

Click the **Menu Editor** button on the toolbar.

3. In the **Caption** text box, type the text for the first menu title that you want to appear on the menu bar. Also, place an ampersand (&) before the letter you want to be the access key for that menu item. This letter will automatically be underlined in the menu.

The menu title text is displayed in the menu control list box.
4. In the **Name** text box, type the name that you will use to refer to the menu control in code. See "Menu Title and Naming Guidelines" later in this chapter.
5. Click the left arrow or right arrow buttons to change the indentation level of the control.
6. Set other properties for the control, if you choose. You can do this in the Menu Editor or later, in the Properties window.
7. Choose **Next** to create another menu control.
– or –
Click **Insert** to add a menu control between existing controls.
You can also click the up arrow and down arrow buttons to move the control among the existing menu controls.
8. Choose **OK** to close the Menu Editor when you have created all the menu controls for that form.
The menu titles you create are displayed on the form. At design time, click a menu title to drop down its corresponding menu items.

Separating Menu Items

A separator bar is displayed as a horizontal line between items on a menu. On a menu with many items, you can use a separator bar to divide items into logical groups. For example, the Help menu in Visual Basic uses separator bars to divide its menu items into three groups, as shown in Figure 6.8.
Figure 6.8 Separator bars

To create a separator bar in the Menu Editor

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

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

Assigning Access Keys and Shortcut Keys

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

Access Keys

Access keys allow the user to open a menu by pressing the ALT key and typing a designated letter. Once a menu is open, the user can choose a control by pressing the letter (the access key) assigned to it. For example, ALT+E might open the Edit menu, and P might select the Paste menu item. An access-key assignment appears as an underlined letter in the menu control's caption, as shown in Figure 6.9.
Figure 6.9 Access keys

To assign an access key to a menu control in the Menu Editor

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

For example, if the Edit menu shown in Figure 6.9 is open, the following Caption property settings respond to the corresponding keys.

Menu control caption	Caption property	Access keys
Cut	Cu&t	t
Copy	C&opy	o
PasteDeleteSelect AllTime/Date	&PasteDe&leteSelect &AllTime/&Date	plad

Note Do not use duplicate access keys on menus. If you use the same access key for more than one menu item, the key will not work. For example, if C is the access key for both Cut and Copy, when you select the Edit menu and press C, the Copy command will be selected, but the application will not carry out the command until the user presses ENTER. The Cut command will not be selected at all.

Shortcut Keys

Shortcut keys run a menu item immediately when pressed. Frequently used menu items may be assigned a keyboard shortcut, which provides a single-step method of keyboard access, rather than a three-step method of pressing ALT, a menu title access character, and then a menu item access character. Shortcut key assignments include function key and control key combinations, such as CTRL+F1 or CTRL+A. They appear on the menu to the right of the corresponding menu item, as shown in Figure 6.10.

Figure 6.10 Shortcut keys

To assign a shortcut key to a menu item

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

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

Note Shortcut keys appear automatically on the menu; therefore, you do not have to enter CTRL+*key* in the Caption box of the Menu Editor.

- **Menu Title and Naming Guidelines** A discussion of naming conventions.

To maintain consistency with other applications, it's a good idea to follow established naming guidelines when creating menus.

Setting the Caption Property

When assigning captions for menu items, you should try to follow these guidelines:

- Item names should be unique within a menu, but may be repeated in different menus to represent similar actions.
- Item names may be single, compound, or multiple words.
- Each item name should have a unique mnemonic access character for users who choose commands with keyboards. The access character should be the first letter of the menu title, unless another letter offers a stronger mnemonic link; no two menu titles should use the same access character. For more information about assigning access and shortcut keys, see "Creating Menus with the Menu Editor" earlier in this chapter.
- An ellipsis (…) should follow names of commands that require more information before they can be completed, such as commands that display a dialog (Save As…, Preferences…).
- Keep the item names short. If you are localizing your application, the length of words tends to increase approximately thirty percent in foreign versions, and you may not have enough space to adequately list all of your menu items. For more details on localizing your application, see "International Issues."

Menu Naming Conventions

To make your code more readable and easier to maintain, it's a good idea to follow established naming conventions when setting the Name property in the Menu Editor. Most naming convention

guidelines suggest a prefix to identify the object (that is, mnu for a menu control) followed by the name of the top-level menu (for example, File). For submenus, this would be followed by the caption of the submenu (for example, mnuFileOpen).
For More Information For an example of suggested naming conventions, see "Visual Basic Coding Conventions."

- **Creating Submenus** A discussion of submenus.

Each menu you create can include up to five 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.)

In the Menu Editor, 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.

To create a submenu

1. Create the menu item that you want to be the submenu title.
2. Create the items that will appear on the new submenu, and indent them by clicking the right arrow button.
Each indent level is preceded by four dots (....) in the Menu Editor. To remove one level of indentation, click the left arrow button.

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

- **Creating a Menu Control Array** Using arrays to eliminate redundant menu code.

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

- Create a new menu item at run time when it must be a member of a control array. The MDI Notepad sample, for example, uses a menu control array to store a list of recently opened files.
- Simplify code, because common blocks of code can be used for all menu items.

Each menu control array element is identified by a unique index value, indicated in the Index property box on the Menu Editor. When a member of a control array recognizes an event, Visual Basic passes its Index property value to the event procedure as an additional argument. Your event procedure must include code to check the value of the Index property, so you can determine which control you're using.

For More Information For more information on control arrays, see "Working with Control Arrays" in "Using Visual Basic's Standard Controls."

To create a menu control array in the Menu Editor

1. Select the form.
2. From the **Tools** menu, choose **Menu Editor**.
– or –
Click the **Menu Editor** button on the toolbar.

3. In the **Caption** text box, type the text for the first menu title that you want to appear on the menu bar.
The menu title text is displayed in the menu control list box.
4. In the **Name** text box, type the name that you will use to refer to the menu control in code. Leave the **Index** box empty.
5. At the next indentation level, create the menu item that will become the first element in the array by setting its **Caption** and **Name**.
6. Set the **Index** for the first element in the array to 0.
7. Create a second menu item at the same level of indentation as the first.
8. Set the **Name** of the second element to the same as the first element and set its **Index** to 1.
9. Repeat steps 5 – 8 for subsequent elements of the array.

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

- **Creating and Modifying Menus at Run Time** Techniques for working with menus.

The menus you create at design time can also respond dynamically to run-time conditions. For example, if a menu item action becomes inappropriate at some point, you can prevent users from selecting that menu item by *disabling* it. In the MDI NotePad application, for example, if the clipboard doesn't contain any text, the Paste menu item is dimmed on the Edit menu, and users cannot select it.

You can also dynamically add menu items, if you have a menu control array. This is described in "Adding Menu Controls at Run Time," later in this topic.

You can also program your application to use a check mark to indicate which of several commands was last selected. For example, the Options, Toolbar menu item from the MDI NotePad application displays a check mark if the toolbar is displayed. Other menu control features described in this section include code that makes a menu item visible or invisible and that adds or deletes menu items.

Enabling and Disabling Menu Commands

All menu controls have an Enabled property, and when this property is set to False, the menu is disabled and does not respond to user actions. Shortcut key access is also disabled when Enabled is set to False. A disabled menu control appears dimmed, like the Paste menu item in Figure 6.11.
Figure 6.11 A disabled menu item

For example, this statement disables the Paste menu item on the Edit menu of the MDI NotePad application:
```
mnuEditPaste.Enabled = False
```

Disabling a menu title in effect disables the entire menu, because the user cannot access any menu item without first clicking the menu title. For example, the following code would disable the Edit menu of the MDI Notepad application:
```
mnuEdit.Enabled = False
```

Displaying a Check Mark on a Menu Control

Using the Checked property, you can place a check mark on a menu to:
- Tell the user the status of an on/off condition. Choosing the menu command alternately adds and removes the check mark.
- Indicate which of several modes is in effect. The Options menu of the MDI Notepad application uses a check mark to indicate the state of the toolbar, as shown in Figure 6.12.

Figure 6.12 A checked menu item

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

```
Private Sub mnuOptions_Click ()
      ' Set the state of the check mark based on
      ' the Visible property.
      mnuOptionsToolbar.Checked = picToolbar.Visible
End Sub
```

Making Menu Controls Invisible

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

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

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

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

Note Making a menu control invisible effectively disables it, because the control is inaccessible from the menu, access or shortcut keys. If the menu title is invisible, all the controls on that menu are unavailable.

Adding Menu Controls at Run Time

A menu can grow at run time. In Figure 6.13, for example, as files are opened in the SDI NotePad application, menu items are dynamically created to display the path names of the most recently opened files.

Figure 6.13 Menu control array elements created and displayed at run time

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

When you create mnuRecentFile(0), you actually create a separator bar that is invisible at run time. The first time a user saves a file at run time, the separator bar becomes visible, and the first file name is added to the menu. Each time you save a file at run time, additional menu controls are loaded into the array, making the menu grow.

Controls created at run time can be hidden by using the Hide method or by setting the control's Visible property to False. If you want to remove a control in a control array from memory, use the Unload statement.

- **Writing Code for Menu Controls** A discussion of coding techniques for menus.

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.

The code that you write in a menu event procedure is no different than that which you would write in any other control's event procedure. For example, the code in a File, Close menu's Click event might look like this:

```
 Sub mnuFileClose_Click()
       Unload Me
End Sub
```

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 Editor. Choosing a menu item on the form displays the Click event procedure for that menu control.

- **Displaying Pop-up Menus** Techniques for working with pop-up menus.

A *pop-up menu* is a floating menu that is displayed over a form, independent of the menu bar. The items displayed on the pop-up menu depend on where the pointer was located when the right mouse button was pressed; therefore, pop-up menus are also called *context menus*. In Microsoft Windows 95, you activate context menus by clicking the right mouse button.

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

[*object*.]**PopupMenu** *menuname* [*, flags* [*,x* [, *y* [, *boldcommand*]]]]

For example, the following code displays a menu named mnuFile when the user clicks a form with the right mouse button. You can use the MouseUp or MouseDown event to detect when the user clicks the right mouse button, although the standard is to use the MouseUp event:

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

Any code following a call to the PopupMenu method is not run until the user selects an item in the menu or cancels the menu.
Note Only one pop-up menu can be displayed at a time. While a pop-up menu is displayed, calls to the PopupMenu method are ignored. Calls to the PopupMenu method are also ignored whenever a menu control is active.

Often you want a pop-up menu to access options that are not usually available on the menu bar. To create a menu that will not display on the menu bar, make the top-level menu item invisible at design time (make sure the Visible check box in the Menu Editor is not checked). When Visual Basic displays a pop-up menu, the Visible property of the specified top-level menu is ignored.

The Flags Argument

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

Location constants	**Description**
vbPopupMenuLeftAlign	Default. The specified *x* location defines the left edge of the pop-up menu.
vbPopupMenuCenterAlign	The pop-up menu is centered around the specified *x* location.
vbPopupMenuRightAlign	The specified *x* location defines the right edge of the pop-up menu.

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

Behavior constants	**Description**
vbPopupMenuLeftButton	Default. The pop-up menu is displayed when the user clicks a menu item with the left mouse button only.
vbPopupMenuRightButton	The pop-up menu is displayed when the user clicks a menu item with either the right or left mouse button.

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

```
Private Sub Command1_Click ()
      ' Dimension X and Y variables.
      Dim xloc, yloc

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

      ' Display the pop-up menu.
      PopupMenu mnuEdit, vbPopupMenuCenterAlign Or _
      vbPopupMenuRightButton, xloc, yloc
End Sub
```

The Boldcommand Argument

You use the *boldcommand* argument to specify the name of a menu control in the displayed pop-up menu that you want to appear in bold. Only one menu control in the pop-up menu can be bold.

- **Menus in MDI Applications** A discussion of MDI menu techniques.

In an MDI application, the menus for each child are displayed on the MDI form, rather than on the child forms themselves. When a child form has the focus, that child's menu (if any) replaces the MDI form's menu on the menu bar. If there are no child forms visible, or if the child with the focus does not have a menu, the MDI form's menu is displayed (see Figures 6.14 and 6.15).

It is common for MDI applications to use several sets of menus. When the user opens a document, the application displays the menu associated with that type of document. Usually, a different menu is displayed when no child forms are visible. For example, when there are no files open, Microsoft Excel displays only the File and Help menus. When the user opens a file, other menus are displayed (File, Edit, View, Insert, Format, Tools, Data, Window, and so on).

Creating Menus for MDI Applications

You can create menus for your Visual Basic application by adding menu controls to the MDI form and to the child forms. One way to manage the menus in your MDI application is to place the menu controls you want displayed all of the time, even when no child forms are visible, on the MDI form. When you run the application, the MDI form's menu is automatically displayed when there are no child forms visible, as shown in Figure 6.14.

Figure 6.14 The MDI form menu is displayed when no child forms are loaded

Place the menu controls that apply to a child form on the child form. At run time, as long as there is at least one child form visible, these menu titles are displayed in the menu bar of the MDI form.
Some applications support more than one type of document. For example, in Microsoft Access, you can open tables, queries, forms, and other document types. To create an application such as this in Visual Basic, use two child forms. Design one child with menus that perform spreadsheet tasks and the other with menus that perform charting tasks.

At run time, when an instance of a spreadsheet form has the focus, the spreadsheet menu is displayed, and when the user selects a chart, that form's menu is displayed. If all the spreadsheets and charts are closed, the MDI form's menu is displayed. For more information on creating menus, see "Using Menus in Your Application" earlier in this chapter.

Creating a Window Menu

Most MDI applications (for example, Microsoft Word for Windows and Microsoft Excel) incorporate a Window menu. This is a special menu that displays the captions of all open child forms, as shown in Figure 6.15. In addition, some applications place commands on this menu that manipulate the child windows, such as Cascade, Tile, and Arrange Icons.

Figure 6.15 The Window menu displays the name of each open child form

Any menu control on an MDI form or MDI child form can be used to display the list of open child forms by setting the WindowList property for that menu control to True. At run time, Visual Basic automatically manages and displays the list of captions and displays a check mark next to the one that had the focus most recently. In addition, a separator bar is automatically placed above the list of windows.

To set the WindowList property

1. Select the form where you want the menu to appear, and from the **Tools** menu, choose **Menu Editor**.

Note The WindowList property applies only to MDI forms and MDI child forms. It has no effect on standard (non-MDI) forms.

2. In the **Menu Editor** list box, select the menu where you want the list of open child forms to display.

3. Select the **WindowList** check box.

At run time, this menu displays the list of open child forms. In addition, the WindowList property for this menu control returns as True.

For More Information See "WindowList Property" in the *Language Reference* on Books Online.

Arranging Child Forms

As was mentioned earlier, some applications list actions such as Tile, Cascade, and Arrange Icons on a menu, along with the list of open child forms. Use the Arrange method to rearrange child forms in the MDI form. You can display child forms as cascading, as horizontally tiled, or as child form icons arranged along the lower portion of the MDI form. The following example shows the Click event procedures for the Cascade, Tile, and Arrange Icons menu controls.

```
Private Sub mnuWCascade_Click ()
      ' Cascade child forms.
      frmMDI.Arrange vbCascade
End Sub

Private Sub mnuWTile_Click ()
      ' Tile child forms (horizontal).
      frmMDI.Arrange vbTileHorizontal
End Sub

Private Sub mnuWArrange_Click ()
      ' Arrange all child form icons.
      frmMDI.Arrange vbArrangeIcons
End Sub
```

Note The intrinsic constants vbCascade, vbTileHorizontal, and vbArrangeIcons are listed in the Visual Basic (VB) object library of the Object Browser.

When you tile or cascade child forms that have a fixed border style, each child form is positioned as if it had a sizable border. This can cause child forms to overlap.

Toolbars How to add toolbars to your application.

You can further enhance your application's menu interface with toolbars. Toolbars contain toolbar buttons, which provide quick access to the most frequently used commands in an application. For example, the Visual Basic toolbar contains toolbar buttons to perform commonly used commands, such as opening existing projects or saving the current project.

- **Creating a Toolbar** Techniques for creating your own toolbars.

The *toolbar* (also called a ribbon or control bar) has become a standard feature in many Windows-based applications. A toolbar provides quick access to the most frequently used menu commands in an application. Creating a toolbar is easy and convenient using the toolbar control, which is available with the Professional and Enterprise editions of Visual Basic. If you are using the Learning Edition of Visual Basic, you can create toolbars manually as described in "Negotiating Menu and Toolbar Appearance" later in this chapter.

The following example demonstrates creating a toolbar for an MDI application; the procedure for creating a toolbar on a standard form is basically the same.

To manually create a toolbar

1. Place a picture box on the MDI form.

The width of the picture box automatically stretches to fill the width of the MDI form's workspace. The workspace is the area inside a form's borders, not including the title bar, menu bar, or any toolbars, status bars, or scroll bars that may be on the form.

Note You can place only those controls that support the Align property directly on an MDI form (the picture box is the only standard control that supports this property).

2. Inside the picture box, place any controls you want to display on the toolbar.

Typically, you create buttons for the toolbar using command buttons or image controls. Figure 6.16 shows a toolbar containing image controls.

To add a control inside a picture box, click the control button in the toolbox, and then draw it inside the picture box.

Note When an MDI form contains a picture box, the internal area of the MDI form does not include the area of the picture box. For example, the ScaleHeight property of the MDI form returns the internal height of the MDI form, which does not include the height of the picture box.

Figure 6.16 You can create buttons for the toolbar using image controls

3. Set design-time properties.

One advantage of using a toolbar is that you can present the user with a graphical representation of a command. The image control is a good choice as a toolbar button because you can use it to display a bitmap. Set its Picture property at design time to display a bitmap; this provides the user with a visual cue of the command performed when the button is clicked. You can also use *ToolTips*, which display the name of the toolbar button when a user rests the mouse pointer over a button, by setting the ToolTipText property for the button.

4. Write code.

Because toolbar buttons are frequently used to provide easy access to other commands, most of the time you call other procedures, such as a corresponding menu command, from within each button's Click event.

Tip You can use controls that are invisible at run time (such as the timer control) with an MDI form without displaying a toolbar. To do this, place a picture box on the MDI form, place the control in the picture box, and set the picture box's Visible property to False.

Writing Code for Toolbars

Toolbars are used to provide the user with a quick way to access some of the application's commands. For example, the first button on the toolbar in Figure 6.16 is a shortcut for the File New command. There are now three places in the MDI NotePad sample application where the user can request a new file:

- On the MDI form (New on the MDI form File menu)
- On the child form (New on the child form File menu)
- On the toolbar (File New button)

Rather than duplicate this code three times, you can take the original code from the child form's mnuFileNew_Click event and place it in a public procedure in the child form. You can call this procedure from any of the preceding event procedures. Here's an example:

```
' This routine is in a public procedure.
Public Sub FileNew ()
      Dim frmNewPad As New frmNotePad
      frmNewPad.Show
End Sub

' The user chooses New on the child form File menu.
Private Sub mnuchildFileNew_Click ()
      FileNew
End Sub

' The user chooses New on the MDI form File menu.
Private Sub mnumdiFileNew_Click ()
      frmNotePad.FileNew
End Sub

' The user clicks the File New button on the toolbar.
Private Sub btnFileNew_Click ()
      frmNotePad.FileNew
End Sub
```

- **Negotiating Menu and Toolbar Appearance** Working with toolbars and menus for insertable objects.

When an object supplied by another application is activated on a form, there are a number of ways that object's menus and toolbars may appear on the container form; therefore, you need to specify how they will be displayed. This process is called *user-interface negotiation* because Visual Basic and the object you have linked or embedded must negotiate for space in the container form.

Controlling Menu Appearance

You can determine whether a linked or embedded object's menu will appear in the container form by setting a form's NegotiateMenus property. If the child form's NegotiateMenus property is set to True (default) and the container has a menu bar defined, the object's menus are placed on the container's menu bar when the object is activated. If the container has no menu bar, or the NegotiateMenus property is set to False, the object's menus will not appear when it is activated.

Note The NegotiateMenus property does *not* apply to MDI Forms.

Controlling Toolbar Appearance

The MDI form's NegotiateToolbars property determines whether the linked or embedded object's toolbars will be floating palettes or placed on the parent form. This behavior does not require toolbars to be present on the MDI parent form. If the MDI form's NegotiateToolbars property is True, the object's toolbar appears on the MDI parent form. If NegotiateToolbars is False, the object's toolbar will be a floating palette.

Note The NegotiateToolbars property applies *only* to MDI forms.

If an MDI form includes a toolbar, it is usually contained in a picture box control on the parent form. The picture box's Negotiate property determines whether the container's toolbar is still displayed or is replaced by the object's toolbar when activated. If Negotiate is True, the object's toolbar is displayed in addition to the container's toolbar. If Negotiate is False, the object's toolbar replaces the container's toolbar.
Note Menu and toolbar negotiation will occur only for insertable objects that support in-place activation. For more information on in-place activation, see "Programming with ActiveX Components."

You can see how these three properties interact by using the following procedure.

To perform menu and toolbar negotiation

1. Add a toolbar to an MDI form. This is described in "Creating a Toolbar" earlier in this chapter.
2. Place an insertable object on a child form.
3. Set the NegotiateMenus, NegotiateToolbars, and Negotiate properties.
4. Run the application, and double-click the object.

Dialog Boxes How to use dialog and message boxes to interact with users.

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

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

In Visual Basic, for example, you use the File Open dialog box to display existing projects. The About dialog box in Visual Basic is also an example of how you can use a dialog box to display information. When the user clicks the Help, About Visual Basic menu item on the menu bar, the About dialog box is displayed.

The following topics discuss dialog boxes:

- **Modal and Modeless Dialog Boxes** A discussion of dialog box types.

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

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

Modeless dialog boxes let you shift the focus between the dialog box and another form without having to close the dialog box. You can continue to work elsewhere in the current application while the dialog box is displayed. Modeless dialog boxes are rare. From the Edit menu, the Find dialog box in Visual Basic is an example of a modeless dialog box. Use modeless dialog boxes to display frequently used commands or information.

To display a form as a modal dialog box

- Use the Show method with a *style* argument of vbModal (a constant for the value 1).

For example:
```
' Display frmAbout as a modal dialog.
frmAbout.Show vbModal
```

To display a form as a modeless dialog box

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

For example:
```
' Display frmAbout as a modeless dialog.
```

```
frmAbout.Show
```

Note If a form is displayed as modal, the code following the Show method is not executed until the dialog box is closed. However, when a form is shown as modeless, the code following the Show method is executed immediately after the form is displayed.

The Show method has another optional argument, *owner*, that can be used to specify a parent-child relationship for a form. You can pass the name of a form to this argument to make that form the owner of the new form.

To display a form as a child of another form

- Use the Show method with both *style* and *owner* arguments.

For example:

```
' Display frmAbout as a modeless child of frmMain.
frmAbout.Show vbModeless, frmMain
```

Using the *owner* argument with the Show method ensures that the dialog box will be minimized when it’s parent is minimized, or unloaded should the parent form be closed.

- **Using Predefined Dialog Boxes** Techniques for using message and input boxes.

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

The following table lists the functions you can use to add predefined dialog boxes to your Visual Basic application.

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

Prompting for Input with InputBox

Use the InputBox function to solicit data from the user. This function displays a modal dialog box that asks the user to enter some data. The text input box shown in Figure 6.17 prompts the user for the name of the file to open.

Figure 6.17 A dialog box using the InputBox function

The following code displays the input box shown in Figure 6.17:

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

Note Remember that when you use the InputBox function, you have little control over the components of the dialog box. You can change only the text in the title bar, the command prompt displayed to the user, the position of the dialog box on the screen, and whether or not it displays a Help button.

For More Information See "InputBox Function" in the *Language Reference* in Books Online.

Displaying Information with MsgBox

Use the MsgBox function to get yes or no responses from users, and to display brief messages, such as errors, warnings, or alerts in a dialog box. After reading the message, the user chooses a button to close the dialog box.

An application named Text Editor might display the message dialog box shown in Figure 6.18 if a file cannot be opened.

Figure 6.18 An error message dialog box created using the MsgBox function

The following code displays the message box shown in Figure 6.18:

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

Note Modality can either be limited to the application or the system. If a message box's modality is limited to the application (default), then users cannot switch to another part of the application until the dialog box is dismissed, but they can switch to another application. A system modal message box does not allow the user to switch to another application until the message box is dismissed.

For More Information See "MsgBox Function" in the *Language Reference* in Books Online.

- **Using Forms as Custom Dialog Boxes** Creating your own dialog boxes.

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

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

To customize an existing dialog box

1. From the **Project** menu, choose **Add Form** to add an existing form to your project.
2. From the **File** menu, choose **Save** *filename* **As** and enter a new file name. (This prevents you from making changes to the existing version of the form).
3. Customize the appearance of the form as needed.
4. Customize event procedures in the Code window.

To create a new dialog box

1. From the **Project** menu, choose **Add Form**.
– or –
Click the **Form** button on the toolbar to create a new form.
2. Customize the appearance of the form as needed.
3. Customize event procedures in the Code window.

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

Adding a Title

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

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

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

Setting Standard Dialog Box Properties

Generally, the user responds to a dialog box by providing information and then closing the dialog box with an OK or Cancel command button. Because a dialog box is temporary, users usually don't need to move, size, maximize, or minimize it. As a result, the sizable border style, Control menu

box, Maximize button, and Minimize button that come with a new form are unnecessary on most dialog boxes.
You can remove these items by setting the BorderStyle, ControlBox, MaxButton, and MinButton properties. For example, an About dialog box might use the following property settings.

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

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

Adding and Placing Command Buttons

Modal dialog boxes must contain at least one command button to exit the dialog box. Usually, two command buttons are used: one button to let the user start an action, and one button to close the dialog box without making any changes. Typically, the Caption property settings for these buttons are OK and Cancel. In this scenario, the OK command button has its Default property set to True, and the Cancel button has its Cancel property set to True. Although OK and Cancel are the most commonly used buttons, other button caption combinations work as well.
Dialog boxes that display messages usually use a label control to display the error message or command prompt, and one or two command buttons to perform an action. For example, you might assign the error message or command prompt to the Caption property of the label, and Yes and No to the Caption property of two command button controls. When users choose Yes, one action takes place; when they choose No, another action occurs.
Command buttons on this type of dialog are usually placed on the bottom or right side of the dialog box, with the top or left button being the default button, as shown in Figure 6.19.

Figure 6.19 Command button placement on dialog boxes

Setting Default, Cancel, and Focus

Command button controls provide the following properties:

- Default
- Cancel
- TabIndex
- TabStop

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

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

You can also specify the button that will have the focus when the dialog is displayed. The control with the lowest TabIndex setting receives the focus when the form is displayed. Pressing the ENTER key invokes the Click event for the default command button or for the command button that has the focus. To give a command button the focus when the form is displayed, set the command button's TabIndex to 0 and its TabStop property to True. You can also use the SetFocus method to give a specific control the focus when a form is displayed.
For More Information   See "TabIndex Property" and "TabStop Property" in the *Language Reference* on Books Online.

Disabling Controls on a Dialog Box
Sometimes controls need to be disabled because their actions would be inappropriate in the current context. For example, when the Visual Basic Find dialog box is first displayed, the Find Next button is disabled, as shown in Figure 6.20. You can disable a control on a dialog by setting its Enabled property to False.
Figure 6.20   Disabled controls on a dialog box

To disable a control on a dialog box

- Set each control's Enabled property to False. For example:

```
cmdFindNext.Enabled = False
cmdReplace.Enabled = False
```

Displaying a Custom Dialog Box
You display a dialog box in the same way you display any other form in an application. The startup form loads automatically when the application is run. When you want a second form or dialog box to appear in the application, you write code to load and display it. Similarly, when you want the form or dialog box to disappear, you write code to unload or hide it.
The following code displays the About dialog box when the user selects the Help, About menu item:

```
Private Sub mnuHelpAbout_Click ()
	' The Show method with style = vbModal is used here
	' to display the dialog as modal.
	frmAbout.Show vbModal
End Sub
```

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

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

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

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

```
Unload frmAbout
–       or      –
frmAbout.Hide
```

The Unload statement removes the dialog box from memory, while the Hide method merely removes the dialog box from view by setting its Visible property to False. When you unload a form, the form itself and its controls are unloaded from memory (including any controls that were loaded at run time). When you hide a form, the form and its controls remain in memory.
When you need to save space in memory, it's better to unload a form, because unloading a form frees memory. If you use the dialog box often, you can choose to hide the form. Hiding a form retains any data attached to it, including property values, print output, and dynamically created controls. By hiding a form, you can continue to refer to the properties and controls of a hidden form in code.

Designing for Different Display Types A discussion of display considerations.

Microsoft Windows is device-independent — a windows-based application can be run on many different computers with different display resolutions and color depths. The applications that you write in Visual Basic are likely to be run on different display types as well; you need to be aware of this when designing an application.

Designing Resolution-independent Forms

By default, Microsoft Visual Basic doesn't change your form and control sizes as you change screen resolutions. What this means is that a form that you design at 1024 by 768 resolution may extend past the edges of the screen when run at 640 by 480 resolution. If you want to create forms and controls that have the same proportions no matter what screen resolution you use, you must either design your forms at the lowest resolution, or add code to your program that changes the forms.
The easiest way to avoid sizing problems is to design your forms at 640 by 480 resolution. If you prefer to work at a higher resolution, you still need to be aware of how your form will appear at a lower resolution. One way to do this is to create a 640 by 480 pixel solid color bitmap and assign it to the Picture property of your form. You can then place your controls within the boundaries of the bitmap at design time. Don't forget to remove the bitmap once you're done with the design.
Visual Basic also places your form at run time based on its location at design time. If you are running at 1024 by 768 resolution at design time and you place a form in the lower right-hand corner of the screen, it may not be visible when run at a lower resolution. You can avoid this by positioning your form with code in the Form Load event:

```
Private Sub Form_Load()
      Me.Move 0, 0
End Sub
```

This has the same effect as setting both the Left and Top properties of the form to 0, but the Move method accomplishes it in a single step.
Visual Basic uses a device-independent unit of measurement, a *twip*, for calculating size and position. Two properties of the Screen object, TwipsPerPixelX and TwipsPerPixelY, can be used to determine the size of the display at run time. Using these properties, you can write code to adjust the size and position of your forms and controls:

```
Private Sub SetControls()
      Dim X As Integer
      Dim Y As Integer

      X = Screen.TwipsPerPixelX
      Y = Screen.TwipsPerPixelY
      Select Case X, Y
            Case 15, 15
```

```
                  ' Resize and move controls.
                  txtName.Height = 200
                  txtName.Width = 500
                  txtName.Move 200, 200
            ' Add code for other resolutions.
            …
End Sub
```

You also need to be aware of the position of Visual Basic's own windows at design time. If you position the Project window to the right side of the screen at high resolution, you may find that it is no longer accessible when you open your project at a lower resolution.

Designing for Different Color Depths

In designing an application, you also need to consider the color display capabilities of the computers that may be running your application. Some computers can display 256 or more colors, others are limited to 16. If you design a form using a 256-color palette, *dithering* (a process used to simulate colors that are not available) may cause some of the elements on the form to disappear when displayed at 16 colors.

To avoid this situation, it's best to limit the colors used in your application to the 16 standard Windows colors. These are represented by the Visual Basic color constants (vbBlack, vbBlue, vbCyan, and so on). If it's necessary to use more than 16 colors in your application, you should still stick with the standard colors for text, buttons, and other interface elements.

Designing with the User in Mind A discussion of user-focused design and design techniques.

Unless you're creating Visual Basic applications strictly for your own use, the value of your creations is going to be judged by others. The user interface of your application has the greatest impact on the user's opinion — no matter how technically brilliant or well optimized your code may be, if the user finds your application difficult to use, it won't be well received.

As a programmer, you are undoubtedly familiar with the technological aspects of computers. It's easy to forget that most users don't understand (and probably don't care) about the technology behind an application. They see an application as a means to an end: a way to accomplish a task, ideally more easily than they would without the aid of a computer. A well-designed user interface insulates the user from the underlying technology, making it easy to perform the intended task.

In designing the user interface for your application, you need to keep the user in mind. How easily can a user discover the various features of your application without instruction? How will your application respond when errors occur? What will you provide in terms of help or user assistance? Is the design aesthetically pleasing to the user? The answers to these and other questions relating to user-focused design are covered in this section.

- **The Basics of Interface Design** A discussion of graphic design principles.

You don't need to be an artist to create a great user interface — most of the principles of user interface design are the same as the basic design principles taught in any elementary art class. The elementary design principles of composition, color, and so forth apply equally well to a computer screen as they do to a sheet of paper or a canvas.

Although Visual Basic makes it easy to create a user interface by simply dragging controls onto a form, a little planning up front can make a big difference in the usability of your application. You might consider drawing your form designs on paper first, determining which controls are needed, the relative importance of the different elements, and the relationships between controls.

Composition: The Look and Feel of an Application

The composition or layout of your form not only influences its aesthetic appeal, it also has a tremendous impact on the usability of your application. Composition includes such factors as positioning of controls, consistency of elements, affordances, use of white space, and simplicity of design.

Positioning of Controls

In most interface designs, not all elements are of equal importance. Careful design is necessary to ensure that the more important elements are readily apparent to the user. Important or frequently accessed elements should be given a position of prominence; less important elements should be relegated to less prominent locations.

In most languages, we are taught to read from left to right, top to bottom of a page. The same holds true for a computer screen — most user's eyes will be drawn to the upper left portion of the screen first, so the most important element should go there. For example, if the information on a form is related to a customer, the name field should be displayed where it will be seen first. Buttons, such as OK or Next, should be placed in the lower right portion of the screen; the user normally won't access these until they have finished working with the form.

Grouping of elements and controls is also important. Try to group information logically according to function or relationship. Because their functions are related, buttons for navigating a database should be grouped together visually rather than scattered throughout a form. The same applies to information; fields for name and address are generally grouped together, as they are closely related. In many cases, you can use frame controls to help reinforce the relationships between controls.

Consistency of Interface Elements

Consistency is a virtue in user interface design. A consistent look and feel creates harmony in an application — everything seems to fit together. A lack of consistency in your interface can be confusing, and can make an application seem chaotic, disorganized, and cheap, possibly even causing the user to doubt the reliability of an application.

For visual consistency, establish a design strategy and style conventions before you begin development. Design elements such as the types of controls, standards for size and grouping of controls, and font choices should be established in advance. You can create prototypes of possible designs to help you make design decisions.

The wide variety of controls available for use in Visual Basic make it tempting to use them all. Avoid this temptation; choose a subset of controls that best fit your particular application. While list box, combo box, grid, and tree controls can all be used to present lists of information, it's best to stick with a single style where possible.

Also, try to use controls appropriately; while a text box control can be set to read-only and used to display text, a label control is usually more appropriate for that purpose. Remain consistent in the setting of properties for your controls — if you use a white back color for editable text in one place, don't use grey in another unless there's a good reason.

Consistency between different forms in your application is important to usability. If you use a grey background and three-dimensional effects on one form and a white background on another, the forms will appear to be unrelated. Pick a style and stick with it throughout your application, even if it means redesigning some features.

Affordances: Form Follows Function

Affordances are visual clues to the function of an object. Although the term may be unfamiliar, examples of affordances are all around you. A handgrip on a bicycle has depressions where you place your fingers, an affordance that makes it obvious that it is meant to be gripped. Push buttons, knobs, and light switches are all affordances — just by looking at them you can discern their purpose.

A user interface also makes use of affordances. For instances, the three-dimensional effects used on command buttons make them look like they are meant to be pushed. If you were to design a command button with a flat border, you would lose this affordance and it wouldn't be clear to the user that it is a command button. There are cases where flat buttons might be appropriate, such as games or multimedia applications; this is okay as long as you remain consistent throughout your application.

Text boxes also provide a sort of affordance — users expect that a box with a border and a white background will contain editable text. While it's possible to display a text box with no border (BorderStyle = 0), this will make it look like a label and it won't be obvious to the user that it is editable.

Use of White Space

The use of *white space* in your user interface can help to emphasize elements and improve usability. White space doesn't necessarily have to be white — it refers to the use of blank space between and around controls a form. Too many controls on a form can lead to a cluttered interface, making it difficult to find an individual field or control. You need to incorporate white space in your design in order to emphasize your design elements.
Consistent spacing between controls and alignment of vertical and horizontal elements can make your design more usable as well. Just as text in a magazine is arranged in orderly columns with even spacing between lines, an orderly interface makes your interface easy to read.
Visual Basic provides several tools that make it easy to adjust the spacing, alignment, and size of controls. Align, Make Same Size, Horizontal Spacing, Vertical Spacing, and Center in Form commands can all be found under the Format menu.

Keep It Simple

Perhaps the most important principle of interface design is one of simplicity. When it comes to applications, if the interface looks difficult, it probably is. A little forethought can help you to create an interface that looks (and is) simple to use. Also, from an aesthetic standpoint, a clean, simple design is always preferable.
A common pitfall in interface design is to try and model your interface after real-world objects. Imagine, for instance, that you were asked to create an application for completing insurance forms. A natural reaction would be to design an interface that exactly duplicates the paper form on screen. This creates several problems: the shape and dimensions of a paper form are different than those of the screen, duplicating a form pretty much limits you to text boxes and check boxes, and there's no real benefit to the user.
It's far better to design your own interface, perhaps providing a printed duplicate (with print preview) of the original paper form. By creating logical groupings of fields from the original form and using a tabbed interface or several linked forms, you can present all of the information without requiring the user to scroll. You can also use additional controls, such as a list box preloaded with choices, which reduce the amount of typing required of the user.
You can also simplify many applications by taking infrequently used functions and moving them to their own forms. Providing defaults can sometimes simplify an application; if nine out of ten users select bold text, make the text bold by default rather than forcing the user to make a choice each time (don't forget to provide an option to override the default). Wizards can also help to simplify complex or infrequent tasks.
The best test of simplicity is to observe your application in use. If a typical user can't immediately accomplish a desired task without assistance, a redesign may be in order.

Using Color and Images

The use of color in your interface can add visual appeal, but it's easy to overuse it. With many displays capable of displaying millions of colors, it's tempting to use them all. Color, like the other basic design principles, can be problematic if not carefully considered in your initial design.
Preference for colors varies widely; the user's taste may not be the same as your own. Color can evoke strong emotions, and if you're designing for international audiences, certain colors may have cultural significance. It's usually best to stay conservative, using softer, more neutral colors.
Of course, your choice of colors may also be influenced by the intended audience and the tone or mood you are trying to convey. Bright reds, greens, and yellows may be appropriate for a children's application, but would hardly evoke an impression of fiscal responsibility in a banking application.
Small amounts of bright color can be used effectively to emphasize or draw attention to an important area. As a rule of thumb, you should try and limit the number of colors in an application, and your color scheme should remain consistent. It's best to stick to a standard 16-color palette if possible; dithering can cause some other colors to disappear when viewed on a 16-color display.
Another consideration in the use of color is that of colorblindness. Many people are unable to tell the difference between different combinations of primary colors such as red and green. To someone with this condition, red text on a green background would be invisible.

Images and Icons

The use of pictures and icons can also add visual interest to your application, but again, careful design is essential. Images can convey information compactly without the need for text, but images are often perceived differently by different people.

Toolbars with icons to represent various functions are a useful interface device, but if the user can't readily identify the function represented by the icon, they can be counterproductive. In designing toolbar icons, look at other applications to see what standards are already established. For example, many applications use a sheet of paper with a folded corner to represent a New File icon. There may be a better metaphor for this function, but representing it differently could confuse the user.

It's also important to consider the cultural significance of images. Many programs use a picture of a rural-style mailbox with a flag (Figure 6.21) to represent mail functions. This is primarily an American icon; users in other countries or cultures probably won't recognize it as a mailbox.

Figure 6.21 An icon representing a mailbox

In designing your own icons and images, try to keep them simple. Complex pictures with a lot of colors don't degrade well when displayed as a 16-by-16 pixel toolbar icon, or when displayed at high screen resolutions.

Choosing Fonts

Fonts are also an important part of your user interface, because they often communicate important information to the user. You need to select fonts that will be easily readable at different resolutions and on different types of displays. It's best to stick with simple sans serif or serif fonts where possible. Script and other decorative fonts generally look better in print than on screen, and can be difficult to read at smaller point sizes.

Unless you plan on distributing fonts along with your application, you should stick to standard Windows fonts such as Arial, New Times Roman, or System. If the user's system doesn't include a specified font, the system will make a substitution, resulting in a completely different appearance than what you intended. If you're designing for an international audience, you'll need to investigate what fonts are available in the intended languages. Also, you'll need to consider text expansion when designing for other languages — text strings can take up to 50% more space in some languages.

Again, design consistency is important in choosing fonts. In most cases, you shouldn't use more than two fonts at two or three different point sizes in a single application. Too many fonts can leave your application looking like a ransom note.

- **Designing for Usability** Tips for creating a user-focused design.

The usability of any application is ultimately determined by the user. Interface design is an iterative process; rarely is the first pass at designing an interface for your application going to yield a perfect interface. By getting users involved early in the design process, you can create a better, more usable interface with less effort.

What is a Good Interface?

The best place to start when designing a user interface is to look at some of the best-selling applications from Microsoft or other companies; after all, they probably didn't get to be best-sellers because of poor interfaces. You'll find many things in common, such as toolbars, status bars, ToolTips, context-sensitive menus, and tabbed dialogs. It's no coincidence that Visual Basic provides the capabilities for adding all of these to your own applications.

You can also borrow from your own experience as a user of software. Think about some of the applications that you have used; what works, what doesn't, and how you would fix it. Remember, however, that your personal likes and dislikes may not match those of your users; you'll need to validate your ideas with them.

You may have also noticed that most successful applications provide choices to accommodate varying user preferences. For instance, the Microsoft Windows Explorer allows users to copy files with menus, keyboard commands, or by drag and drop. Providing options will broaden the appeal of your application; as a minimum you should make all functions accessible by both mouse and keyboard.

Windows Interface Guidelines

One of the main advantages of the Windows operating system is that it presents a common interface across all applications. A user that knows how to use one Windows-based application should be able to easily learn any other. Unfortunately, applications that stray too far from the established interface guidelines aren't as easily learned.

Menus are a good example of this — most Windows-based applications follow the standard of a File menu on the left, then optional menus such as Edit and Tools, followed by Help on the right. It could be argued that Documents would be a better name than File, or that the Help menu should come first. There's nothing to prevent you from doing this, but by doing so you will confuse your users and decrease the usability of your application. Users will have to stop and think every time they switch between your application and another.

The placement of submenus is also important. Users expect to find Copy, Cut and Paste beneath the Edit menu; moving them to the File menu would be confusing at best. Don't deviate from the established guidelines unless you have a good reason to do so.

Testing for Usability

The best way to test the usability of your interface is to involve users throughout the design phase. Whether you're designing a major shrink-wrap application or a small application for limited use, the design process should be pretty much the same. Using your established design guidelines, you'll want to start by designing the interface on paper.

The next step is to create one or more prototypes, designing your forms in Visual Basic. You'll need to add just enough code to make the prototype functional: displaying forms, filling list boxes with sample data, and so forth. Then you're ready to start usability testing.

Usability testing can be an informal process, reviewing your design with a few users, or a formal process in an established usability lab. Either way, the purpose is the same — learning first-hand from the users where your design works and where it needs improvement. Rather than questioning the user, it's more effective to simply turn the user loose with the application and observe them. Have the user verbalize their thought process as they attempt to perform a set of tasks: "I want to open a new document, so I will look under the File menu." Make note of where the interface design doesn't respond to their thought processes. Test with multiple users; if you see several users having difficulty with a particular task, that task probably needs more attention.

Next, you'll want to review your notes and consider how you can change the interface to make it more usable. Make the changes to your interface and test it again. Once you are satisfied that your application is usable, you're ready to start coding. You'll also want to test occasionally during the development process to make sure that the assumptions for the prototype were valid.

Discoverability of Features

One of the key concepts in usability testing is that of discoverability. If a user can't discover how to use a feature (or even that a feature exists), that feature is of little use. For example, the majority of Windows 3.1 users were never aware that the ALT, TAB key combination could be used to switch between open applications. There was no clue anywhere in the interface to help users discover this feature.

To test the discoverability of a feature, ask the user to perform a task without explaining how to do it (for example, "Create a new document using a Form Letter Template"). If they can't accomplish the task, or if it takes several attempts, the discoverability of that feature needs work.

- **When Things Go Wrong: Interacting with Users** Techniques for gracefully handling user and system errors.

In an ideal world, software and hardware would always work flawlessly, and users would never make mistakes. Reality dictates that mistakes can and will happen. A part of user interface design involves deciding how the application will respond when things go wrong.

A common response is to display a dialog box, asking for user input as to how the application should deal with the problem. A less common (but preferable) response would be to simply resolve the problem without bothering the user. After all, the user is primarily concerned with performing a task, not with technical details. In designing your user interface, think about the potential errors and determine which ones require user interaction and which ones can be resolved programmatically.

Creating Intelligent Dialog Boxes

Occasionally an error occurs in your application and it's necessary to make a decision in order to resolve the situation. This usually occurs as a branch in your code — an If…Then statement or a Case statement. If the decision requires user interaction, the question is usually posed to the user

with a dialog box. Dialog boxes are a part of your user interface, and like the other parts of the interface, their design plays a role in the usability of your application.
Sometimes it seems as if many dialog boxes were designed by programmers who have never had an intelligent conversation with another human being. A message such as "A sector of fixed disk C: is corrupted or inaccessible. Abort, Retry, Ignore?" (see Figure 6.22) has little meaning to the average user. It's kind of like a waitress asking you "We're out of soup or the kitchen is on fire. Abort, Retry, Ignore?" How would you answer? It's important to phrase questions (and choices) in a manner that the user can understand. In the prior example, a better message might be "There is a problem saving your file on drive C. Save file on drive A, Don't save the file?"
Figure 6.22 Which dialog box presents the clearest message?

When creating dialog boxes for your application, keep the user in mind. Does the message convey useful information to the user? Is it easily understandable? Do the command buttons present clear choices? Are the choices appropriate for the given situation? Keep in mind that it only takes one annoying message box to give a user a bad impression of your application.
If you're designing your own custom dialog forms, try to stick to a standard style. If you vary too far from the standard message box layout, users may not recognize it as a dialog box.
For More Information To learn more about dialogs, see "Dialog Boxes" earlier in this chapter.

Handling Errors Without Dialog Boxes

It isn't always necessary to interrupt the user when an error occurs. Sometimes it's preferable to handle the error in code without notifying the user, or to warn the user in a way that doesn't stop their work flow. A good example of this technique is the AutoCorrect feature in Microsoft Word: if a common word is mistyped, Word fixes it automatically; if a less common word is misspelled, it is underlined in red so the user can correct it later.
There are a number of techniques that you can use; it's up to you to decide which techniques are appropriate for your own application. Here are a few suggestions:

- Add an Undo function to the Edit menu. Rather than interrupting the user with a confirmation dialog for deletions and so forth, trust that they are making the right decision and provide a Undo function in case they change their mind later.
- Display a message on a status bar or icon. If the error doesn't affect the user's current task, don't stop the application. Use a status bar or a brightly colored warning icon to warn the user — they can handle the problem when they are ready.
- Correct the problem. Sometimes the solution to an error is obvious. For instance, if a disk is full when the user tries to save a file, check the system for space on other drives. If space is available, save the file; put a message on the status bar to let the user know what you did.
- Save the message until later. Not all errors are critical or demand immediate attention; consider logging these to a file and displaying them to the user when they exit the application or at another convenient time. If the user makes a possible entry error (for example, Mian St. instead of Main St.), log it. Add a Review Entries button and a function to display the discrepancies so the user can correct them.
- Don't do anything. Sometimes an error isn't important enough to warrant a warning. For instance, the fact that a printer on LPT1 is out of paper doesn't mean much until you're ready to print. Wait until the message is appropriate to the current task.

For More Information To learn more about error handling techniques, see "Debugging Your Code and Handling Errors."

- **Designing a User Assistance Model** Providing help for learning and using your application.

No matter how great your user interface, there will be times that a user needs assistance. The user assistance model for your application includes such things as online Help and printed documentation; it may also contain user assistance devices such as ToolTips, Status Bars, What's This help, and Wizards.

The user assistance model should be designed just like any other part of your application: before you start developing. The contents of your model will vary depending on the complexity of the application and the intended audience.

Help and Documentation

Online Help is an important part of any application — it's usually the first place a user will look when they have a question. Even a simple application should provide Help; failing to provide it is like assuming that your users will never have questions.

In designing your Help system, keep in mind that its primary purpose is to answer questions. Try to think in terms of the user when creating topic names and index entries; for example, "How do I format a page?" rather than "Edit, Page Format menu" will make your topic easier to locate. Don't forget about context sensitivity; it's frustrating to most users if they press the F1 key for help on a specific field and find themselves at the Contents topic.

Conceptual documentation, whether printed and/or provided on compact disc, is helpful for all but the simplest applications. It can provide information that may be difficult to convey in the shorter Help topic. At the very least, you should provide documentation in the form of a ReadMe file that the user can print if desired.

User Assistance Devices

Within the user interface, there are several techniques for providing assistance to the user. Visual Basic makes it easy to add ToolTips, What's This help, Status displays, and Wizards to your applications. It's up to you to decide which of these devices are appropriate for your application.

ToolTips

ToolTips (Figure 6.23) are a great way to display information to the user as they navigate the user interface. A ToolTip is a small label that is displayed when the mouse pointer is held over a control for a set length of time, usually containing a description of the control's function. Normally used in conjunction with toolbars, ToolTips also work well in most any part of the interface.

Figure 6.23 A ToolTip for the Visual Basic toolbar

Most Visual Basic controls contain a single property for displaying ToolTips: ToolTipText. The following code would implement a ToolTip for a command button named cmdPrint:

```
cmdPrint.ToolTipText = "Prints the current document"
```

As with other parts of the interface, make sure that the text clearly conveys the intended message to the user.

For More Information To learn more about ToolTips, see "ToolTipText Property" in the *Language Reference* in Books Online.

What's This Help

What's this Help provides a link to a pop-up Help topic (see Figure 6.24) when the user selects What's This Help and clicks the What's This cursor on a control. What's This Help can be initiated from a toolbar button, a menu item, or a button on the title bar of a dialog box.

Figure 6.24 A What's This Help pop-up window

To enable What's This Help from a menu or toolbar

1. Select the control for which you wish to provide help .
2. In the Properties window, select the **WhatsThisHelpID** property.
3. Enter a context ID number for the associated pop-up Help topic.
4. Repeat steps 1 through 3 for any additional controls.
5. Select the form.
6. In the Properties window, set the Form's WhatsThisHelp property to True.
7. In the Click event of the menu or toolbar button, enter the following:

```
formname.WhatsThisHelp
```

When the user clicks the button or menu, the mouse pointer will change to the What's This pointer.

To enable What's This Help on the title bar of a custom dialog form, set the form's WhatsThisButton and WhatsThisHelp properties to True.
For More Information To learn more about What's This Help, see "WhatsThisHelp Property" and WhatsThisButton Property" in the *Language Reference* in Books Online.

Status Displays

A status display can also be used to provide user assistance in much the same way as a ToolTip. Status displays are a good way to provide instructions or messages that may not fit easily into a ToolTip. The status bar control included in the Professional and Enterprise editions of Visual Basic works well for displaying messages; a label control can also be used as a status display.
The text displayed in a status display can be updated in one of two ways: in the GotFocus event of a control or form, or in the MouseMove event. If you want to use the display as a learning device, add an item to the Help menu to toggle its Visible property on and off.

To add a status display

1. Add a label control to your form.
2. Select the control for which you wish to display a message.
3. Add the following code to the control's MouseMove (or GotFocus) event:

```
Labelname.Caption = "Enter the customer's ID number in this field"
```

When the user moves the mouse over the control, the message will be displayed in the label control.
4. Repeat steps 2 and 3 for any additional controls.

Wizards

A wizard is a user assistance device that takes the user step by step through a procedure, working with the user's actual data. Wizards are usually used to provide task-specific assistance. They help a user accomplish a task that would otherwise require a considerable (and undesirable) learning curve; they provide expert information to a user that has not yet become an expert.
The Professional and Enterprise editions of Visual Basic include the Wizard Manager, a tool for creating wizards.
For More Information To learn more about wizards, see "Using Wizards and Add-Ins" in "Managing Projects."

ooO   Ooo