

Visual Basic and Databases

6. Visual Basic Interface Design

Review and Preview

- At this point in the course, we can use Visual Basic (with either DAO or ADO technology) to connect to a database and SQL statements allow us to obtain any view of the database information we desire. But, that's all we can do - view the data. We now want to know how to allow a user to interact with the data - obtain alternate views, modify it, add to it, delete it. To do this, we need a well-designed user interface.
- In this chapter, we look at some design considerations for the Visual Basic front-end. We examine the toolbox controls and Visual Basic coding techniques needed to build a useful interface and application. Several examples illustrate use of the tools and techniques.

Interface Design Philosophy

- The design philosophy for a proper application interface is very basic – keep it as **simple** as possible and as **intuitive** as possible. By doing this, you will save yourself (the programmer) and your users a lot of problems. This may be an obvious statement, but you would be surprised at how many programmers do not follow it.
- A first consideration should be to determine what **processes** and **functions** you want your application to perform. What are the **inputs** and **outputs**? Develop a framework or flow chart of all your application's processes. Possible functions of a database interface include: data entry, searching, deleting information, adding information, editing information, sorting data, and printing capabilities.
- Decide if multiple **forms** are required. Decide what **controls** from the Visual Basic toolbox you need. Do the built-in Visual Basic tools and functions meet your needs? Do you need to develop some tools or functions of your own? Do you need to acquire some third-party controls?
- Minimize the possibility of user errors. This is a very important step. The fewer errors your user can make, the less error checking you have to do. If a particular input calls for numeric data, make sure your user can't type in his name. Choose 'point and click' type tools whenever they can be used to replace tools requiring the user to type in something. For example, let the user point at a month of the year, rather than have the user type in the month. If you can avoid letting your user type anything, do it! Every "typed input" requires some kind of validation that means extra work on your part.
- At all steps in the application, make it **intuitive** to the user what he or she is to do. Don't make or let the user guess. You, as the programmer, control the flow of information from the user to the program and vice versa. Maintain that control at all times. Try to anticipate all possible ways a user can mess up in using your application. It's fairly easy to write an application that works properly when the user does everything correctly. It's difficult to write an application that can handle all the possible wrong things a user can do and still not bomb out. And, although it is difficult, it is straightforward and just a matter of following your common sense.

- Make your interface appealing to the user. Use tolerable colors and don't get carried away with too many font types. Make sure there are no misspellings (a personal pet peeve). Make the interface consistent with other Windows applications. Familiarity is good in program design. It is quite proper to 'borrow' ideas from other applications.
- Although not part of the interface the user sees, you should make your code readable and traceable - future code modifiers will thank you. Choose meaningful variable and control names. Use comments to explain what you are doing. Consider developing reusable code - modules with utility outside your current development. This will save you time in future developments.
- Debug your application completely before distributing it. There's nothing worse than having a user call you to point out flaws in your application. A good way to find all the bugs is to let several people try the code - a mini beta-testing program. Let's illustrate some of these philosophies with an example.

Example 6-1

Mailing List Revisited

Open and run the mailing list example built in Chapter 1 (**Example 1**). It illustrates many of the interface design philosophies just discussed. Notice the program flow - how it directs the user about what to do and minimizes the possibility of errors. In particular, note:

- ⇒ You cannot type **Address Information** unless the timer has started (controlled via the **Enabled** property of **fraMail**).
- ⇒ When the Address Information frame is active, the cursor appears in the first text box, so the user starts typing the Name field first (controlled with the **txtInput** text box **SetFocus** method).
- ⇒ After the user types information in each text box, hitting <Enter> or <Tab> automatically moves them to the next text box (controlled with the **SetFocus** method and the **TabIndex** property).
- ⇒ After the user types in the last text box (**Zip**), the focus moves to the **Accept** command button, so a simple <Enter> accepts the mailing label (using **SetFocus** on the **cmdAccept** button).

Notice how the program flow leads the user through the input process. Regarding the timer portion of the application, notice the **Pause** button is faded (**Enabled** is **False**) initially and is only active (**Enabled** is **True**) when the timer is running. The other timer control buttons toggle accordingly.

There is some validation of inputs in this application also. If there are not five values input, a message box appears informing the user of his error. And, only numbers can be typed when **txtInput (Index =4)** is active (done in the **KeyPress** event). This is the box for the **Zip** that can only be a number. It would probably be more proper to also make sure the entered zip matches either the five or nine digit zip code format. Another validation possible would be to provide a list box control with the 50 states (apologies to our foreign readers for using a provincial example) to choose from instead of asking the user to type in a state name.

Regarding the code in the example, notice the use of comments to explain what is happening in each procedure. This helps others read and understand your code. It also helps you know what you were doing when you look back on the code a year later. Also notice that selection of proper variable and control names aids in understanding what is going on in the code portion of the application. Now, let's look at interface design in more detail.

Visual Basic Standard Controls

- The first step in building a Visual Basic interface is to ‘draw’ the application on a form. We place the required controls on the form, set properties, and write BASIC code for the needed event and general procedures. As the interface designer, you need to decide which controls best meet your needs regarding efficiency, applicability, and minimization of error possibilities.
- In this section, we briefly look at the standard Visual Basic controls (available with both DAO and ADO technologies). We examine how they might be used in a database ‘front-end’ and present some of the important properties, events, and methods associated with these controls. This information is provided as a quick review of what is available in the Visual Basic toolbox - a “one-stop” reference to controls and how they are used with databases. A later look at custom controls will complete the reference.

Form Control

- The **Form** is where the user interface is drawn. It is central to the development of Visual Basic applications, whether for databases or other uses.

- Form Properties:

Appearance	Selects 3-D or flat appearance.
BackColor	Sets the form background color.
BorderStyle	Sets the form border to be fixed or sizeable.
Caption	Sets the form window title.
Enabled	If True, allows the form to respond to mouse and keyboard events; if False, disables form and all controls.
Font	Sets font type, style, size.
ForeColor	Sets color of text or graphics.
Visible	If False, hides the form.

- Form Events:

Activate	Form_Activate event is triggered when form becomes the active window.
Click	Form_Click event is triggered when user clicks on form.
DbClick	Form_DbClick event is triggered when user double-clicks on form.
Load	Form_Load event occurs when form is loaded. This is a good place to initialize variables and set any run-time properties.

Command Button Control



- The **command button** is probably the most widely used control. It is used to begin, interrupt, or end a particular process. With **databases**, it is used to **navigate** among records, **add** records, and **delete** records.
- Command Button Properties:

Appearance	Selects 3-D or flat appearance.
BackColor	Background color of button (applies only if Style is Graphical).
Cancel	Allows selection of button with Esc key (only one button on a form can have this property True).
Caption	String to be displayed on button.
Default	Allows selection of button with Enter key (only one button on a form can have this property True).
Font	Sets font type, style, size.
Picture	Picture appearing on button (applies only if Style is Graphical).
Style	Button can be Standard or Graphical.
- Command Button Event:

Click	Event triggered when button is selected either by clicking on it or by pressing the access key.
--------------	---
- Command Button Method:

SetFocus	Places the focus on the command button.
-----------------	---

Label Control



- A **label** is a control you use to display text. The text in a label can be changed at run-time in response to events. It is widely used in **database** applications for **information display**.

- Label Properties:

Alignment	Aligns caption within border.
Appearance	Selects 3-D or flat appearance.
BackColor	Background color of label.
BorderStyle	Determines type of border.
Caption	String to be displayed in box (property bound to database).
DataField	Field in database table, specified by DataSource (or DataMember), bound to label (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the label is bound to.
Font	Sets font type, style, size.
ForeColor	Color of text in label.

- Label Events:

Click	Event triggered when user clicks on a label.
DbClick	Event triggered when user double-clicks on a label.

Text Box Control



- A **text box** is used to display information entered at design time, by a user at run-time, or assigned within code. The displayed text may be edited. This is the tool used in **database** applications for **editing** fields.

- Text Box Properties:

Appearance	Selects 3-D or flat appearance.
BackColor	Background color of text box.
BorderStyle	Determines type of border.
DataField	Field in database table, specified by DataSource (or DataMember), bound to text box (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the text box is bound to.
Font	Sets font type, style, size.
ForeColor	Color of text in text box.
Locked	When True, the text box contents cannot be edited.
MultiLine	Specifies whether text box displays single line or multiple lines.
ScrollBars	Determines what scroll bars (if any) appear.
Text	Displayed text (property bound to database).

- Text Box Events:

Change	Triggered every time the Text property changes.
LostFocus	Triggered when the user leaves the text box. This is a good place to examine the contents of a text box after editing.
KeyPress	Triggered whenever a key is pressed. Used for key trapping, as seen in Example 6-1.

- Text Box Methods:

SetFocus	Places the cursor in a specified text box.
-----------------	--

Check Box Control



- **Check boxes** provide a way to make choices from a list of potential candidates. Some, all, or none of the choices in a group may be selected. With **databases**, check boxes are used for many kinds of **choices**.

- Check Box Properties:

Caption	Identifying text next to box.
DataField	Field in database table, specified by DataSource (or DataMember), bound to check box (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the check box is bound to.
Font	Sets font type, style, size for Caption.
Value	Indicates if unchecked (0, vbUnchecked), checked (1, vbChecked), or grayed out (2, vbGrayed) (property bound to database).

- Check Box Event:

Click	Triggered when a box is clicked. Value property is automatically changed by Visual Basic.
--------------	---

Option Button Control



- **Option buttons** provide the capability to make a mutually exclusive choice among a group of potential candidate choices. Hence, option buttons work as a group, only one of which can have a True (or selected) value. Option buttons on a form work as an independent group as do groups of options buttons within frames. Option buttons are not data bound controls, yet they can still be used for a variety of **options** in database interfaces.

- Option Button Properties:

Caption	Identifying text next to button.
Font	Sets font type, style, size.
Value	Indicates if selected (True) or not (False). Only one option button in a group can be True. One button in each group of option buttons should always be initialized to True at design time.

- Option Button Event:

Click	Triggered when a button is clicked. Value property is automatically changed by Visual Basic.
--------------	---

Frame Control



- **Frames** provide a way of grouping related controls on a form. Option buttons within a frame act independently of other option buttons in an application.

- Frame Properties:

Caption	Title information at top of frame.
Font	Sets font type, style, size.

Picture Box Control



- The **picture box** allows you to place graphics information on a form. In a **database**, picture boxes are used to store **graphic** data.
- Picture Box Properties:

AutoSize	If True, box adjusts its size to fit the displayed graphic.
DataField	Field in database table, specified by DataSource (or DataMember), bound to picture box (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the picture box is bound to.
Picture	Establishes the graphics file to display in the picture box (property bound to database).

Image Control



- An **image** control is very similar to a picture box in that it allows you to place graphics information on a form. The advantage to an image control is its ability to scale displayed graphics.
- Image Box Properties:

DataField	Field in database table, specified by DataSource (or DataMember), bound to image control (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the image control is bound to.
Picture	Establishes the graphics file to display in the image box (property bound to database).
Stretch	If False, the image box resizes itself to fit the graphic. If True, the graphic resizes to fit the control area.

Example 6-2

Authors Table Input Form

In Chapter 7, we will build a complete database management system for the books database. Each table in the database will require some kind of input form. In this chapter, we build such a form for the **Authors** table. Even though it is a very simple table (only three fields: **Au_ID**, **Author**, **Year Born**), it provides an excellent basis to illustrate many of the steps of proper interface design. We need an input form that allows a user to edit an existing record, delete an existing record or add a new record. The form should also allow navigation from one record to another.

The books database management example (including the Authors input form) will be built using the **DAO data control**. If you prefer to use either the **ADO data control** or **ADO data environment**, we provide needed modification steps. These steps will be in shaded boxes. In the code accompanying this course, we use a special naming convention for all files (projects, forms) to distinguish among examples built using each technology. The convention is:

FileName (no suffix, DAO data control)

FileNameAD (**AD** suffix, ADO data control)

FileNameDE (**DE** suffix, ADO data environment)

We suggest you use this same naming convention.

1. Start a new application. We need three label controls and three text boxes to display the fields. We need two command buttons to move from one record to the next. We need five command buttons to control editing features and one command button to allow us to stop editing. Lastly, a DAO data control is needed. Place these controls on a form. The layout should resemble:

ADO Data Control Modification

Use an ADO data control instead of the DAO data control.

ADO Data Environment Modification

Add an ADO data environment to the project instead of the DAO data control.

2. Set these properties for the form and controls:

Form1:

Name	frmAuthors
BorderStyle	1-Fixed Single
Caption	Authors

Data1:

Name	datAuthors
DatabaseName	BIBLIO.MDB (point to your copy)
RecordSource	SELECT * FROM Authors ORDER BY Author
Visible	False

ADO Data Control Modifications**Adodc1:**

Name	datAuthors
CommandType	1-adCmdText
ConnectionString	Use the Build option to point to the BIBLIO.MDB database
RecordSource	SELECT * FROM Authors ORDER BY Author
Visible	False

ADO Data Environment Modifications

1. **Name** the data environment **denBooks**. **Name** the connection object **conBooks**. Right-click **conBooks** and set **Properties** so it points to your working copy of **BIBLIO.MDB**.
2. Add a **command** object. Assign these properties:

Command1:

Name	comAuthors
ConnectionName	conBooks
CommandType	1-adCmdText
CommandText	SELECT * FROM Authors ORDER BY Author

Label1:

Caption	Author ID
---------	-----------

Text1:

Name	txtAuthorID
DataSource	datAuthors
DataField	Au_ID
Locked	True

ADO Data Environment Modifications**Text1:**

Name	txtAuthorID
DataSource	denBooks
DataMember	comAuthors
DataField	Au_ID
Locked	True

Label2:

Caption	Author Name
---------	-------------

Text2:

Name	txtAuthor
DataSource	datAuthors
DataField	Author
Locked	True

ADO Data Environment Modifications**Text2:**

Name	txtAuthor
DataSource	denBooks
DataMember	comAuthors
DataField	Author
Locked	True

Label3:

Caption	Year Born
---------	-----------

Text3:

Name	txtYearBorn
DataSource	datAuthors
DataField	Year Born
Locked	True

ADO Data Environment Modifications**Text3:**

Name	txtYearBorn
DataSource	denBooks
DataMember	comAuthors
DataField	Year Born
Locked	True

Command1:

Name	cmdPrevious
Caption	<= Previous

Command2:

Name	cmdNext
Caption	Next =>

Command3:

Name	cmdEdit
Caption	&Edit

Command4:

Name	cmdSave
Caption	&Save

Command5:

Name	cmdCancel
Caption	&Cancel

Command6:

Name	cmdAddNew
Caption	&Add New

Command7:

Name	cmdDelete
Caption	&Delete

Command8:

Name	cmdDone
Caption	Do&ne

Note, we lock (**Locked = True**) all the text boxes. We will unlock them when we (as the programmer) decide the user can change a value (remember, we are in control). At this point, the form should appear as:

The screenshot shows a Visual Basic form titled "Authors". It features three text boxes for data entry: "Author ID" (containing "Text1"), "Author Name" (containing "Text2"), and "Year Born" (containing "Text3"). To the right of the "Author ID" box is a "Data1" control with navigation buttons. Below the text boxes are six command buttons arranged in two rows: "<= Previous", "Next =>", "Edit", "Save", "Cancel" in the first row, and "Add New", "Delete", "Done" in the second row. The form has a dotted grid background.

3. We will add features to this input application as we progress through the chapter. At this point, we add code to allow us to navigate through the Authors table records. There are two event procedures to code. First, the **cmdPrevious_Click** event:

```
Private Sub cmdPrevious_Click()  
datAuthors.Recordset.MovePrevious  
If datAuthors.Recordset.BOF Then  
    datAuthors.Recordset.MoveFirst  
End If  
End Sub
```

And, the **cmdNext_Click** event:

```
Private Sub cmdNext_Click()  
datAuthors.Recordset.MoveNext  
If datAuthors.Recordset.EOF Then  
    datAuthors.Recordset.MoveLast  
End If  
End Sub
```

ADO Data Environment Modification

In above code, replace all occurrences of **datAuthors.Recordset** with **denBooks.rscomAuthors**. This incorporates the data environment's convention for naming the recordset.

4. Save the application. Run it. Navigate among the records. Note you cannot edit anything. The text boxes are locked. As we progress through this chapter (and the next), we will continue to add features to this example until it is complete.

Message Box

- Many times, in a database application, you will want to impart some information to your user. That information may be a courtesy message (“New record written”) or a question requiring feedback (“Do you really want to delete this record?”). Visual Basic (and Windows) provides an excellent medium for providing such information – the **message box**.
- A message box displays a message, an optional icon, and a selected set of command buttons. The user responds to the message box by clicking one of the button(s). If you’ve done any work in the Windows environment, you have seen message boxes. For example, here’s one that appears in Visual Basic when the floppy disk drive is not ready:



The user responds by fixing the problem and clicking **Retry** (the default response) or by clicking **Cancel**. Visual Basic then takes the necessary actions depending on user response. We can add this same capability to our Visual Basic database applications. The great thing about the message box is that it is familiar to the user (familiarity is good) and it is easy to use.

- To use a message box in BASIC code requires just one line of code. There are two forms for the message box. The **statement** form returns no value (it simply displays the box). The code syntax is:

MsgBox Message, Type, Title

where

Message	Text of message to be displayed (string)
Type	Type of message box (integer, discussed in a bit)
Title	Text in title bar of message box (string)

You have no control over where the message box appears on the screen (it is usually centered).

- The **function** form of the message box returns an integer value (corresponding to the button clicked by the user). Example of use (Response is returned value):

```
Dim Response as Integer
Response = MsgBox(Message, Type, Title)
```

- The **Type** argument is formed by summing four values corresponding to the button(s) to display, any icon to show, which button is the default response, and the modality of the message box.
- The first component of the **Type** value specifies the **buttons** to display:

Value	Meaning	Symbolic Constant
0	OK button only	vbOKOnly
1	OK/Cancel buttons	vbOKCancel
2	Abort/Retry/Ignore buttons	vbAbortRetryIgnore
3	Yes/No/Cancel buttons	vbYesNoCancel
4	Yes/No buttons	vbYesNo
5	Retry/Cancel buttons	vbRetryCancel

Pick the set of buttons that meets your need.

- The second component of **Type** specifies the **icon** to display in the message box:

Value	Meaning	Symbolic Constant
0	No icon	(None)
16	Critical icon	vbCritical
32	Question mark	vbQuestion
48	Exclamation point	vbExclamation
64	Information icon	vbInformation

Pick an icon that corresponds to the displayed message.

- The third component of **Type** specifies which button is **default** (i.e. pressing Enter is the same as clicking the default button):

Value	Meaning	Symbolic Constant
0	First button default	vbDefaultButton1
256	Second button default	vbDefaultButton2
512	Third button default	vbDefaultButton3

Always try to make the default response the “least damaging,” if the user just blindly accepts it.

- The fourth and final component of **Type** specifies the **modality**:

Value	Meaning	Symbolic Constant
0	Application modal	vbApplicationModal
4096	System modal	vbSystemModal

If the box is **Application Modal**, the user must respond to the box before continuing work in the current application. If the box is **System Modal**, all applications are suspended until the user responds to the message box.

- Note for each option in **Type**, there are numeric values listed and symbolic constants. It is strongly suggested that the symbolic constants be used instead of the numeric values. You should agree that **vbOKOnly** means more than the number 0 when selecting the button type.
- The value returned by the function form of the message box is related to the button clicked:

Value	Meaning	Symbolic Constant
1	OK button selected	vbOK
2	Cancel button selected	vbCancel
3	Abort button selected	vbAbort
4	Retry button selected	vbRetry
5	Ignore button selected	vbIgnore
6	Yes button selected	vbYes
7	No button selected	vbNo

- Message boxes should be used whenever your application needs to inform the user of action or requires user feedback to continue. It is probably better to have too many message boxes, than too few. You always want to make sure your application is performing as it should and the more information you have, the better.

Example 6-3

Authors Table Input Form (Message Box)

There are two places where we could use message boxes in the Authors Table example. A statement form after saving an update to let the user know the save occurred and a function form related to deleting records.

1. Load Example 6-2 completed earlier. We will modify this example to include message boxes.

ADO Data Control Modification

Load Example6-2AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-2DE (the ADO data environment version).

2. Attach this code to the **cmdSave_Click** event:

```
Private Sub cmdSave_Click()  
MsgBox "Record saved.", vbOKOnly + vbInformation, "Save"  
End Sub
```

Obviously, there will be more code in this event as we continue with this example. This code just implements the message box.

3. Attach this code to the **cmdDelete_Click** event:

```
Private Sub cmdDelete_Click()  
Dim Response As Integer  
Response = MsgBox("Are you sure you want to delete this  
record?", vbYesNo + vbQuestion + vbDefaultButton2,  
"Delete")  
If Response = vbNo Then  
Exit Sub  
End If  
End Sub
```

Note we exit the procedure if the user selects **No**. And, notice the **No** button is default – this makes the user think a bit before hitting **Enter**. Like above, there will be more code in this procedure as we proceed.

4. Save the application and run it. Click the **Save** and **Delete** buttons to see how the message boxes appear.

Application State

- When presenting a Visual Basic database interface to a user, it should be obvious, to the user, what needs to be done. Options should be intuitive and the possibility of mistakes minimized, if not completely eliminated. To maintain this obvious quality, you should always be aware of what **state** your application is in.
- **Application state** implies knowing just what is currently being done within the interface. Are you adding a record, editing a record, deleting a record, or perhaps leaving the application? Once you know the state the application is in, you adjust the interface so that options needed for that particular state are available to the user. You also need to know when and how to transition from one state to another.
- What options are adjusted to reflect application state? A primary option is a control's **Enabled** property. By setting **Enabled** to **False**, you disable a control, making it unavailable to the user. So, if the user is not able to save a current record, the command button that does the save should have an Enabled property of False. A more drastic disabling of a control is setting its Visible property to False. In this case, there is no misunderstanding about application state. As the application moves from one state to another, you need to determine which controls should be enabled and which should be disabled.
- For **text box** controls, a property of importance is the **Locked** property. If a value in a text box is not to be edited, set Locked to True. When editing is allowed (the state changes), toggle the Locked property to False. For text boxes that are always locked (used for display, not editing purposes), use color (red is good) to indicate they are not accessible. When editing in a text box, use the **SetFocus** method to place the cursor in the box, making it the active control (giving it focus) and saving the user a mouse click. The **SetFocus** method can also be used to programmatically move the user from one text box to the next in a desired order.

- Another mechanism for moving from one control to another in a prescribed order is the **TabIndex** property, in conjunction with **TabStop**. If **TabStop** is **True**, **TabIndex** defines the order controls become active (only one control can be active at a time) as the <Tab> key is pressed (the order is reversed when <Shift>-<Tab> is pressed). When controls are placed on a form at design time, they are assigned a **TabIndex** value with **TabStop** = **True**. If you don't want a control to be made active with <Tab>, you need to reset its **TabStop** property to **False**. If the assigned order is not acceptable, reset the **TabIndex** properties for the desired controls, starting with a low number and increasing that value with each control added to the <Tab> sequence. A primary application for <Tab> sequencing is moving from one text box to the next in a detailed input form.
- If the concepts of control focus and tab movements are new or unfamiliar, try this. Start a new application in Visual Basic. Add three command buttons (**Command1**, **Command2**, **Command3**), then three text boxes (**Text1**, **Text2**, **Text3**). Run the application. The first command button (**Command1**) should have focus (a little outline box is around the caption). If you press <Enter> at this point, this button is 'clicked.' Press the <Tab> key and the focus moves to the second command button. Press <Tab> twice. The focus should now be in the first text box (the cursor is in the box). Keep pressing <Tab> and watch the focus move from one control to the other, always in the same order. Pressing <Shift>-<Tab> reverses the order. Now, for each command button, set the **TabStop** property to **False** (removing them from the tab sequence). Re-run the application and you should note the focus only shifts among the text boxes. Try resetting the **TabIndex** properties of the text boxes to change the shift direction. Always use the idea of focus in your applications to indicate to the user what control is active.
- All of this application state talk may sound complicated, but it really isn't. Again, it's all just a matter of common sense. After you design your interface, sit back and step through your application in the Visual Basic environment, exercising every option available. With each option, ask yourself what the user needs to see. Implement the necessary logic to make sure this is all the user sees. Make sure moves from one state to another are apparent and correct. Try lots of things until you are comfortable with the finished product. The Visual Basic environment makes performing such tasks quite easy.

Example 6-4

Authors Table Input Form (Application State)

The Authors Table Input Form can operate in one of three states: **View** state, **Add** state or **Edit** state. In **View** state, the user can navigate from record to record, access adding and/or editing records, delete a record, or exit the application. In **View** state, data cannot be changed. In both **Add** and **Edit** states, no navigation should be possible, data can be changed, and the user should have access to the **Save** and **Cancel** functions. Each of these states can be implemented using command button **Enabled** properties and text box **Locked** properties. We use **TabIndex** (and **TabOrder**) to control shift of focus in the text box controls. We will use a general procedure to allow switching from one state to another.

1. Open Example 6-3 in the Visual Basic environment. We will modify this example to include state considerations.

ADO Data Control Modification

Load Example6-3AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-3DE (the ADO data environment version).

2. Remove the command buttons from tab sequencing by setting all (eight buttons) of their **TabStop** properties to **False**. Also set **TabStop** to **False** for the **txtAuthorID** text box (we will not edit this value - we'll explain why later). Set **TabIndex** for **txtAuthor** to **1** and **TabIndex** for **txtYearBorn** to **2**.
3. Add a general Sub procedure named **SetState** with string argument **AppState**. To add this procedure, with the code window active, select **Tools** and **Add Procedure**. Fill in the blanks appropriately and a framework appears in the code window.

4. Add this code to the **SetState** procedure (note, the added **AppState** argument):

```
Private Sub SetState(AppState As String)
Select Case AppState
Case "View"
    txtAuthorID.BackColor=vbWhite
    txtAuthor.Locked = True
    txtYearBorn.Locked = True
    cmdPrevious.Enabled = True
    cmdNext.Enabled = True
    cmdAddNew.Enabled = True
    cmdSave.Enabled = False
    cmdCancel.Enabled = False
    cmdEdit.Enabled = True
    cmdDelete.Enabled = True
    cmdDone.Enabled = True
    txtAuthor.SetFocus
Case "Add", "Edit"
    txtAuthorID.BackColor=vbRed
    txtAuthor.Locked = False
    txtYearBorn.Locked = False
    cmdPrevious.Enabled = False
    cmdNext.Enabled = False
    cmdAddNew.Enabled = False
    cmdSave.Enabled = True
    cmdCancel.Enabled = True
    cmdEdit.Enabled = False
    cmdDelete.Enabled = False
    cmdDone.Enabled = False
    txtAuthor.SetFocus
End Select
End Sub
```

This code sets the application in View, Add or Edit state. Note which buttons are available and which are not. Notice the **Author ID** box is red in Add and Edit state to indicate it cannot be changed. Notice that the Add and Edit states are the same (for now) and are just a 'togglng' of the View state – this will occur quite often – a great place for 'cut and paste' coding. We now need to modify the application code to use this procedure to move from state to state.

4. We want to be in the **View** state when the application is initialized. Attach this code to the **Form_Activate** event:

```
Private Sub Form_Activate()  
Call SetState("View")  
End Sub
```

5. When the **Add New** button is clicked, we want to switch to **Add** state. Add this line of code at the top of the **cmdAddNew_Click** event procedure:

```
Call SetState("Add")
```

6. When the **Edit** button is clicked, we switch to **Edit** state. Add this line of code at the top of the **cmdEdit_Click** event procedure:

```
Call SetState("Edit")
```

7. Following a **Cancel** or **Save** operation (in **Add** or **Edit** state), we want to return to **View** state. Place this line at the end of the **cmdCancel_Click** and **cmdSave_Click** event procedures:

```
Call SetState("View")
```

The **Delete** button does not need any change of state code – it only works in **View** state and stays in that state following a delete.

8. We're almost done. This is a small change, but an important one that gives your application a professional touch. Notice that if you click the **Previous** button and the recordset pointer is at the first record, nothing changes. Similarly, at the end of the recordset, if you click **Next**, nothing changes. This lack of change might confuse the user. To give the user some feedback that they've reached a limit, I like to provide some audible feedback. In both the **cmdPrevious_Click** and **cmdNext_Click** procedures, add the Visual Basic **Beep** statement within the **If/End If** structure. Then, when the user bumps a limit, a little beep is heard.
9. Save and run the application. Notice how the various buttons change state as different functions are accessed on the interface form. In **Add** and **Edit** state (the ID box is red), check the tab order of the two text boxes (a very short tab order!). A warning – if you change any value in Add or Edit mode, it will be saved in the database (a feature of the Jet engine). In each state, it is obvious to the user what functions are available and when they are available. Do you hear the beep when you try to move past a limit at the end or beginning of the recordset?

Entry Validation

- Throughout your work with databases, you will find that viewing database information is an easy task with Visual Basic. Things quickly become difficult, though, when you want to modify information in a database. And, things become very difficult when you allow your user to type information. That's why, if at all possible, don't allow your user to type things. Use point and click type controls whenever possible.
- Checking input information from a user requires programming on your part. You must insure information being put in a database is correct. There are two steps to checking information from a user: **entry** validation and **input** validation. Entry validation is associated with text box controls and checks for proper keystrokes. Input validation is associated with several control types and checks to make sure entries and choices meet certain requirements. In this section, we address entry validation. Input validation is addressed in the next section of this chapter.
- As mentioned, entry validation checks for proper keystrokes. For example, if a numerical entry is needed, only allow the pressing of number keys. If spaces are not allowed, don't allow them. If an input must be in upper case letters, don't allow lower case letters to be typed. Restricting keystrokes is referred to as **key trapping**.
- Key trapping is done in the **KeyPress** event procedure of a text box. Such a procedure has the form (for a text box named **txtText**):

```
Sub txtText_KeyPress (KeyAscii as Integer)
    .
    .
    .
End Sub
```

In this procedure, every time a key is pressed in the corresponding text box, the ASCII code for the pressed key is passed to the procedure as the argument **KeyAscii**. With key trapping, if KeyAscii is an acceptable value, we do nothing. If KeyAscii is not acceptable, we set KeyAscii equal to zero and exit the procedure. Doing this has the same result of not pressing a key at all. ASCII values for all keys are available via the on-line help in Visual Basic. The BASIC function **Asc** can also be used to determine a key's ASCII code. And some keys are also defined by symbolic constants. Where possible, we will use symbolic constants; else, we will use the ASCII values.

- As an example, say we have a text box (named **txtExample**) and we only want to be able to enter upper case letters (ASCII codes 65 through 90, or, correspondingly, symbolic constants **vbKeyA** through **vbKeyZ**). The **KeyPress** procedure would look like (the **Beep** causes an audible tone if an incorrect key is pressed):

```
Sub txtExample_KeyPress(KeyAscii as Integer)
If KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ Then
    Exit Sub
Else
    KeyAscii = 0
    Beep
End If
End Sub
```

- In key trapping, it's advisable to always allow the backspace key (ASCII code 8; symbolic constant **vbKeyBack**) to pass through the **KeyPress** event. Else, you will not be able to edit the text box properly. Modifying the above example, this code would be:

```
Sub txtExample_KeyPress(KeyAscii as Integer)
If (KeyAscii >= vbKeyA And KeyAscii <= vbKeyZ) Or KeyAscii
= vbKeyBack Then
    Exit Sub
Else
    KeyAscii = 0
    Beep
End If
End Sub
```

- Rather than just beep when an unacceptable keystroke is encountered, you can also use key trapping to automatically correct invalid inputs. For example, if the input requires all upper case letters, you can use the BASIC **UCase** function to convert any lower case letters to upper case letters.

Example 6-5**Authors Table Input Form (Entry Validation)**

In the Authors Table Input Form, the **Year Born** field can only be numeric data.

1. Load Example 6-4 completed earlier. We will modify this example to include entry validation.

ADO Data Control Modification

Load Example6-4AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-4DE (the ADO data environment version).

2. Attach this code to the **txtYearBorn_KeyPress** event (make sure you select the proper event in the code window – don't use the Change event!):

```
Private Sub txtYearBorn_KeyPress(KeyAscii As Integer)
If (KeyAscii >= Asc("0") And KeyAscii <= Asc("9")) Or
KeyAscii = vbKeyBack Then
    Exit Sub
Else
    Beep
    KeyAscii = 0
End If
End Sub
```

3. Save and run the application. Click **Edit** to switch to Edit state.. Click the **Year Born** text box. Try some typing. You should only be able to type numbers (or use the backspace key) in the Year Born entry box.

Input Validation

- In the example just studied, although the user can only input numeric data for the Year Born field, there is no guarantee the final input would be acceptable. What if the input year is past the current year? What if the year is 1492? A second step in validation is to check values in context. Do the input values make sense? Do the values meet established rules? This step is **input validation**.
- Some common validation rules are:
 - ⇒ Is this field required? If a field is required and no input is provided, this could cause problems.
 - ⇒ Is the input within an established range? For example, if entering a day number for the month of April, is the value between 1 and 30?
 - ⇒ Is the input the proper length? Social security numbers (including hyphens) require 11 characters. If 11 characters are not detected, the input is not a valid social security number. The BASIC **Len** function can be used here, as can a text box **MaxLength** property (to limit the length).
 - ⇒ Is the input conditional? Some fields only need to be filled in if other fields are filled in. For example, if a user clicks to ship to another address, you need to make sure that address exists.
 - ⇒ Is the input a primary key? If so, and the user has the capability of entering a value, we must insure it is a unique value. Each primary key value in a table must be different.
- The amount of input validation required is dependent on the particular field. Many times, there is none needed. You, as the programmer, need to examine each input field and answer the questions posed above: is the field required, must it be within a range, is its length restricted, is it conditional? Any Yes answers require BASIC code to do the validation. You will probably find additional questions as you develop your database skills.
- Where does the validation code go? It really depends on what database technology (DAO or ADO) you are using and how you implement database editing. We will discuss this topic in detail in Chapter 7. For our example we have been creating, we will write a general procedure named **ValidateData** that is called in the **Click** event of the **Save** button. The user clicks this button when done editing, making it a great place to check validity. If any validation rules are violated, we don't allow the requested change(s).

- We see entry and input validation require a bit of programming on our part. But, it is worth it. Field validation insures the integrity of the information we are putting in a database. We always need to maintain that integrity. And, one last time for emphasis (are you getting the idea this is important) – if you can eliminate user typing – do it!

Example 6-6

Authors Table Input Form (Input Validation)

As mentioned, the **Year Born** must be validated. We will make sure that, if an input is attempted (we won't require a year be input), the year has no more than four characters, is less than the current year and greater than 150 years prior to the current year (by not hard coding a year, the code automatically upgrades itself). We will also make sure the user enters an Author Name.

1. Load Example 6-5 completed earlier. We will modify this example to include input validation.

ADO Data Control Modification

Load Example6-5AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-5DE (the ADO data environment version).

2. Add a procedure named **ValidateData** with a Boolean argument **AllOK** (if True, all validation rules were met). Add this code:

```
Private Sub ValidateData(AllOK As Boolean)
Dim Message As String
Dim InputYear As Integer, CurrentYear As Integer
AllOK = True
Message = ""
'Check for name
If Len(txtAuthor.Text) = 0 Then
    Message = "You must enter an Author Name." + vbCrLf
    txtAuthor.SetFocus
    AllOK = False
End If
'Check length and range on Year Born
InputYear = Val(txtYearBorn.Text)
CurrentYear = Val(Format(Now, "yyyy"))
If Len(txtYearBorn.Text) <> 0 Then
    If InputYear > CurrentYear Or InputYear < CurrentYear -
150 Then
```

```
        Message = Message + "Year Born must be between" &  
Str(CurrentYear - 150) & " and" & Str(CurrentYear)  
        txtYearBorn.SetFocus  
        AllOK = False  
    End If  
End If  
If Not (AllOK) Then  
    MsgBox Message, vbOKOnly + vbInformation, "Validation  
Error"  
End If  
End Sub
```

In this code, we first check to see if an **Author Name** is entered and then validate the **Year Born** field. If either validation rule is violated, the variable **AllOK** is set to **False** and a message box displayed. If any of this code is unfamiliar, try Visual Basic on-line help for assistance.

3. Set **MaxLength** property for **txtYearBorn** text box to 4.
4. Modify the **cmdSave_Click** event to read (new lines are italicized)::

```
Private Sub cmdSave_Click()  
    Dim Valid As Boolean  
    Call ValidateData(Valid)  
    If Not (Valid) Then Exit Sub  
    MsgBox "Record saved.", vbOKOnly + vbInformation, "Save"  
    Call SetState("View")  
End Sub
```

This calls the validation routine. If the **Valid** variable is **False** upon return from the routine, the data is not valid and we exit the procedure.

4. Save and run the application. Click **Edit**. Click **Edit** and blank out the **Author Name**. Click **Save**. A message box should appear. Type an invalid numeric value in the **Year Born** box. Click **Save**. A new message should be displayed. Click **OK** and the focus is reset on the **Author Name** text box, helping the user. Try a valid year and valid name – make sure they are accepted.

5. After typing a new Author name, to type a Year Born, you need to click in that text box. This clicking (especially when working with lots of text boxes) is cumbersome. A preferred method would be a programmatic shift of focus. Add this code at the top of the **txtYearBorn_KeyPress** event:

```
If KeyAscii = vbKeyReturn Then
    txtAuthor.SetFocus
    Exit Sub
End If
```

In this code, if the **<Enter>** key is pressed, the focus is shifted from the Year Born text box to the Author text box (if a valid year is input). This programmatic change of focus is used all the time in database interfaces. Users like to see the focus move when they press **<Enter>**. It is an additional step in maintaining proper application state. To shift from the Author box to the Year Born box, add this code to the **Author_KeyPress** event:

```
Private Sub txtAuthor_KeyPress(KeyAscii As Integer)
If KeyAscii = vbKeyReturn Then
    txtYearBorn.SetFocus
    Exit Sub
End If
End Sub
```

6. Save and run the example again. Click **Edit**. Notice how the focus shifts between the two text boxes as you change the values and press **<Enter>**. Pressing **<Tab>** should also change the focus appropriately.

Error Trapping and Handling

- Even with a well-designed, 'user-proof' interface, errors can still occur. This is especially true when working with databases. Occasionally, data cannot be written to, or deleted from, the database or invalid fields are encountered. Without any action on our part, these **run-time errors** might bring our application to an unceremonious end. If, however, we recognize an error has occurred and inform the user of the problem, we might be able to recover.
- The process of detecting errors before they cause big problems is **error trapping** and **handling**. This process differs from entry and input validation. Entry and input validation are used to prevent errors from occurring. Error trapping and handling is what is needed if an error still occurs.
- There are two steps: error **trapping** (detecting an error has occurred) and error **handling** (deciding what to do about the detected error). Error trapping and handling must be implemented in every procedure in your application where you think it might be needed. Visual Basic does not allow global error trapping. At a minimum, you should implement error trapping and handling in every procedure that writes to or reads from the database.
- Error trapping is enabled with the BASIC **On Error** statement:

On Error GoTo *errlabel*

Yes, this uses the dreaded **GoTo** statement! Any time a run-time error occurs following this line, program control is transferred to the line labeled ***errlabel***. Recall a labeled line is simply a line with the label followed by a colon (:).

- The best way to explain how to use error trapping is to look at an outline of an example procedure with error trapping.

```
Sub SubExample()  
    [Declare variables, ...]  
    On Error GoTo HandleErrors  
    [Procedure code]  
Exit Sub  
HandleErrors:  
    [Error handling code]  
End Sub
```

Once you have set up the variable declarations, constant definitions, and any other procedure preliminaries, the **On Error** statement is executed to enable error trapping. Your normal procedure code follows this statement. The error handling code goes at the end of the procedure, following the **HandleErrors** statement label. This is the code that is executed if an error is encountered anywhere in the Sub procedure. Note you must exit (with **Exit Sub**) from the code before reaching the HandleErrors line to avoid inadvertent execution of the error handling code.

- Since the error handling code is in the same procedure where an error occurs, all variables in that procedure are available for possible corrective action. If at some time in your procedure, you want to **turn off** error trapping, that is done with the following statement:

```
On Error GoTo 0
```

You don't need a line labeled '0' for this to work.

- Once a run-time error occurs, we would like to know what the error is and attempt to fix it. This is done in the **error handling** code. Visual Basic offers, through the **Err** object, help in identifying run-time errors. Three Err object properties of particular interest are:

Number	Visual Basic error number
Source	Name of Visual Basic file in which error has occurred
Description	A textual description of the error

Consult on-line help for more details Visual Basic run-time error numbers and their descriptions.

- Once an error has been trapped and some action taken, control must be returned to your application. That control is returned via the **Resume** statement. There are three options:

Resume	Lets you retry the operation that caused the error. That is, control is returned to the line where the error occurred. This could be dangerous in that, if the error has not been corrected (via code or by the user), an infinite loop between the error handler and the procedure code may result.
Resume Next	Program control is returned to the line immediately following the line where the error occurred.
Resume <i>label</i>	Program control is returned to the line labeled <i>label</i> .

You can also use **Exit Sub** as an error handling statement. With this, you immediately exit the routine in which the error occurred.

- Development of an adequate **error handling procedure** is application dependent. You need to know what type of errors you are looking for and what corrective actions must be taken if these errors are encountered. For example, if a 'divide by zero' is found, you need to decide whether to skip the operation or do something to reset the offending denominator. What we develop here is a generic framework for an error handling procedure. It simply informs the user that an error has occurred, provides a description of the error, and allows the user to **Abort**, **Retry**, or **Ignore**. This framework is a good starting point for designing custom error handling for your database applications.
- The generic code (begins with label **HandleErrors**) is:

```
HandleErrors:
Select Case MsgBox(Err.Description, vbCritical +
vbAbortRetryIgnore, "Error Number" + Str(Err.Number) + "
in " + Err.Source)
Case vbAbort
    Exit Sub
Case vbRetry
    Resume
Case vbIgnore
    Resume Next
End Select
```

Let's look at what goes on here. First, this routine is only executed when an error occurs. A message box is displayed, using the Visual Basic provided error description (**Err.Description**) as the message, uses a **critical icon** along with the **Abort**, **Retry**, and **Ignore** buttons, and uses the error number **Err.Number** and **Err.Source** in the title. This message box returns a response indicating which button the user selected. If Abort is selected, we simply exit the procedure. If Retry is selected, the offending program line is retried (in a real application, you or the user would have to change something here to correct the condition causing the error). If Ignore is selected, program operation continues with the line following the error causing line.

- To use this generic code in an existing procedure, you need to do three things:
 1. Copy and paste the error handling code into the end of your procedure.
 2. Place an **Exit Sub** line immediately preceding the **HandleErrors** labeled line.
 3. Place the line, **On Error GoTo HandleErrors**, at the beginning of your procedure.

For example, if your procedure is the **SubExample** seen earlier, the modified code will look like this:

```
Sub SubExample()  
    [Declare variables, ...]  
    On Error GoTo HandleErrors  
    [Procedure code]  
    Exit Sub  
HandleErrors:  
    Select Case MsgBox(Err.Description, vbCritical +  
vbAbortRetryIgnore, "Error Number" + Str(Err.Number) + "  
in " + Err.Source)  
    Case vbAbort  
        Exit Sub  
    Case vbRetry  
        Resume  
    Case vbIgnore  
        Resume Next  
    End Select  
End Sub
```

Again, this is a very basic error-handling routine. You must determine its utility in your applications and make any modifications necessary. Specifically, you need code to clear error conditions before using the Retry option.

- One last thing. Once you've written an error handling routine, you need to test it to make sure it works properly. But, creating run-time errors is sometimes difficult and perhaps dangerous. Visual Basic comes to the rescue! The Visual Basic **Err** object has a method (**Raise**) associated with it that simulates the occurrence of a run-time error. To cause an error with value **Number**, use:

Err.Raise Number

We can use this function to completely test the operation of any error handler we write. Don't forget to remove the Raise statement once testing is completed, though!

Example 6-7

Authors Table Input Form (Error Trapping and Handling)

As mentioned, error trapping and handling should be included within every procedure where database information is read or written. It should also be included in procedures where database files are being opened or saved (covered in later chapters).

1. Load Example 6-6 completed earlier. We will modify this example to include error trapping handling

ADO Data Control Modification

Load Example6-6AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-6DE (the ADO data environment version).

2. Modify the **cmdAddNew_Click**, **cmdSave_Click**, and **cmdDelete_Click** event procedures to allow error trapping and handling. Use the generic code developed in this section, taking advantage of 'cut and paste' editing.
3. Save the application. After the 'On Error' line in the **cmdAddNew_Click** procedure, add this line:

```
Err.Raise 11
```

Run the application. Click **Add New**. You should see a message box reporting a 'divide by zero' error. Check how the application responds when you click **Abort** (stays in View state), **Retry** (keeps displaying an error), or **Ignore** (goes to Add state). Play around with different error numbers, if you like. Before leaving this example, remove this line of code that generates an error.

On-Line Help Systems

- So, at this point, we know how to build a powerful, intuitive interface, insure valid inputs, and handle any run-time errors that might occur. Even with all this work, there still may be times when the user is stumped as to what to do next. Instinct tells the user to press the <F1> function key. Long ago, someone in the old DOS world decided this would be the magic “Help Me!” key. Users expect help when pressing <F1> (I’m sure you rely on it a lot when using Visual Basic). If nothing appears after pressing <F1>, user frustration sets in – not a good thing.
- All applications written for other than your personal use should include some form of an on-line help system. It doesn’t have to be elegant, but it should be there. Adding a **help file** to your Visual Basic application will give it real polish, as well as making it easier to use. In this section, we will show you how to build a very basic on-line help system for your database applications. This system will simply have a list of help topics the user can choose from.
- The on-line help system will be built using the **Microsoft Help Compiler Workshop** which ships with Visual Basic. It is located in the \Tools\HCW folder of the Visual Basic installation CD and must be installed separately (see the CD for instructions). Your help file can contain text and graphics information needed to be able to run your application. The help file will be displayed by the built-in Windows help utility that you (and your users – again, familiarity is good) use with every Windows application, hence all functions available with that utility are available with your help system. For example, each file can contain one or more topics that your user can select by clicking a **hot spot**, using a **keyword search**, or **browsing** through text. And, it’s easy for your user to print any or all help topics.

- Creating a complete help file is a major task and sometimes takes as much time as creating the application itself! Because of this, we will only skim over the steps involved, generate a simple example, and provide guidance for further reference. There are six major steps involved in building your own help file:
 1. Create your application and develop an outline of help system topics.
 2. Create the **Help Text File** (or Topic File) in RTF format (RTF extension).
 3. Create a **Help Contents File** (CNT extension).
 4. Create the **Help Project File** (HPJ extension).
 5. Compile the Help File using the **Help Compiler** and Project File.
 6. **Attach** the Help File to your Visual Basic application.

Step 1 is application-dependent. We'll look briefly at the last five steps here.

- Creating a Help Text File:

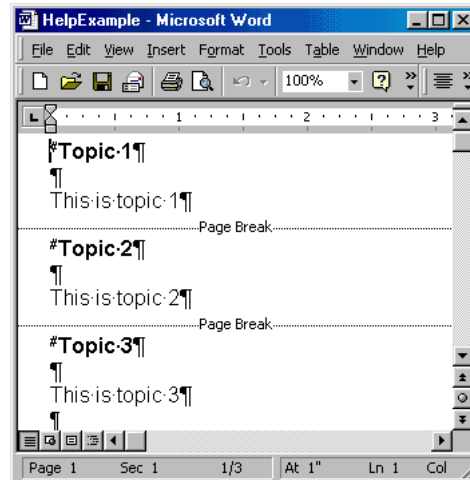
To create a **Help Text File**, you need to use a word processor capable of saving documents in rich-text format (**RTF**). **Word** and **WordPerfect** do admirable jobs. The Help Text File is basically a list of footnoted topics, with associated topic ID's. Some general rules of Help Text Files:

- Topics are separated by hard page breaks (no break after last topic!)
- Each topic must have a Topic ID.
- Each topic can have a title.

Once completed, your Help Text File is saved as an RTF (rich text format) file. Make your text file as complete and accurate as possible and please check for misspellings – nothing scares a user more than a poorly prepared help file. They quickly draw the conclusion that if the help system is not built with care, the application must also be sloppily built.

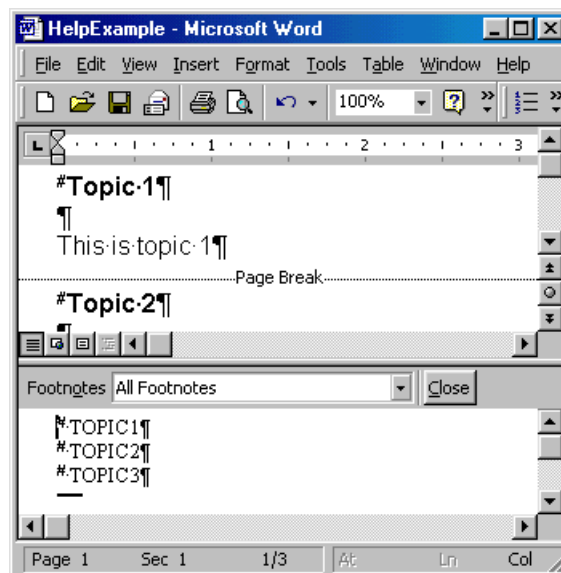
- Help Text File Example:

We'll create a very simple help text file with three topics. I used Word 2000 in this example. Create a document with the following structure and footnotes:



Note page breaks separate each section. Do **not** put a page break at the end of the file.

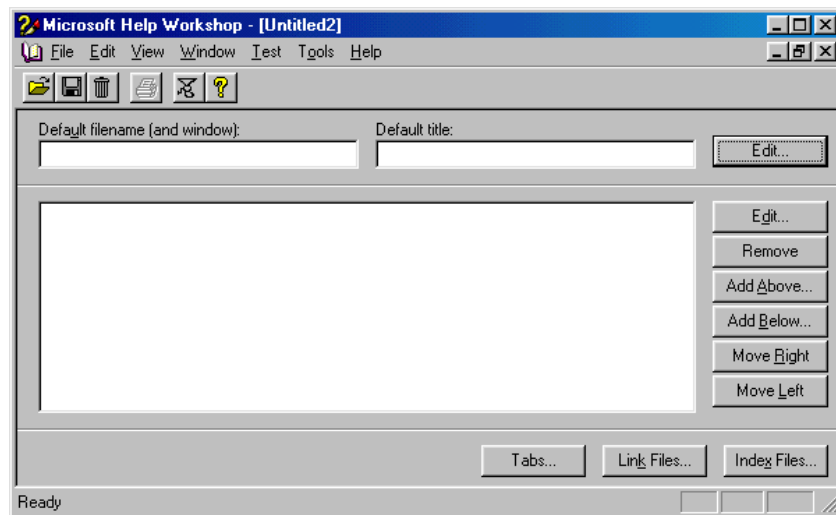
Also, note the use of footnotes. The # footnote is used to specify a Topic ID. The footnotes for this example are:



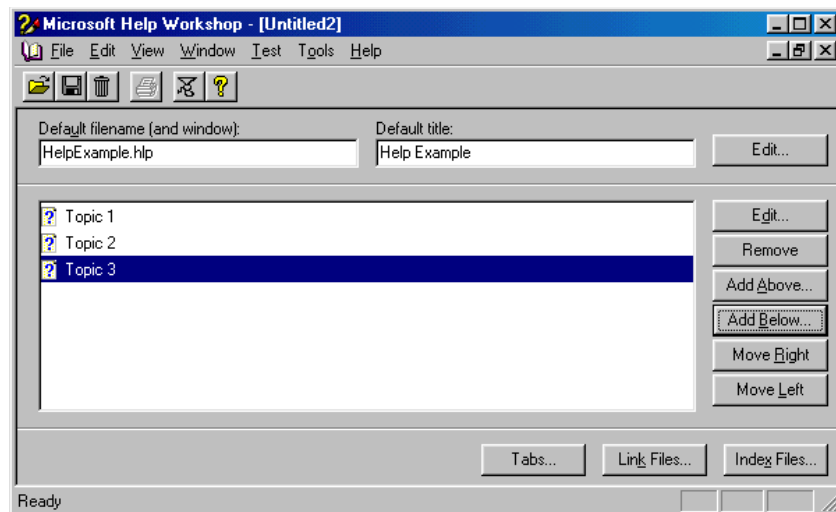
When done, save this file as **HelpExample.rtf** (Rich Text Format).

- Creating the Help Contents File:

The Help Contents File specifies what topics to display when the help system's **Contents** tab is selected. This file is created using the Microsoft Help Compiler Workshop. Start the **Microsoft Help Workshop**. Choose the **New** option under the **File** menu. Choose **Help Contents**, then click **OK**. The following window appears:



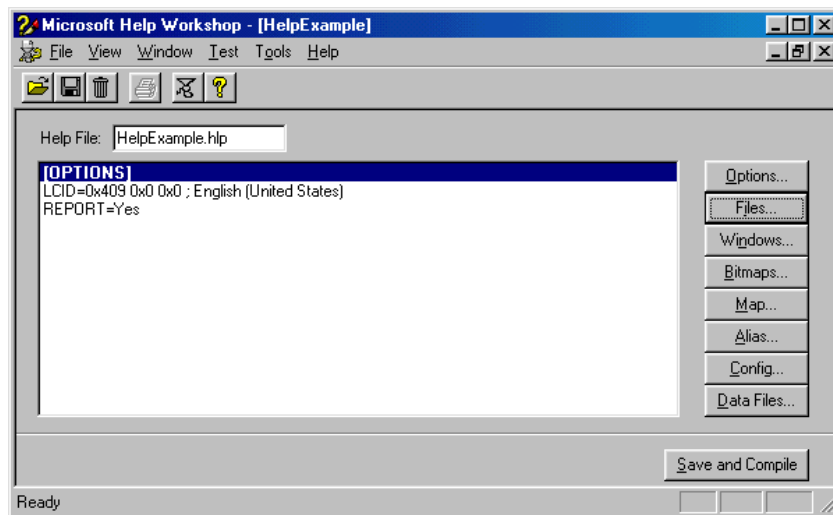
Click the top **Edit** button. For **Default Filename**, type **HelpExample.hlp**. For **Default Title**, type **Help Example**. Click **OK**. We'll now add the three topics. Click **Add Below**. In the window that appears, type **Topic 1** for **Title** and **TOPIC1** for **Topic ID**. Click **OK**. Repeat these steps for Topic 2 and Topic 3. When complete, the window should look like this:



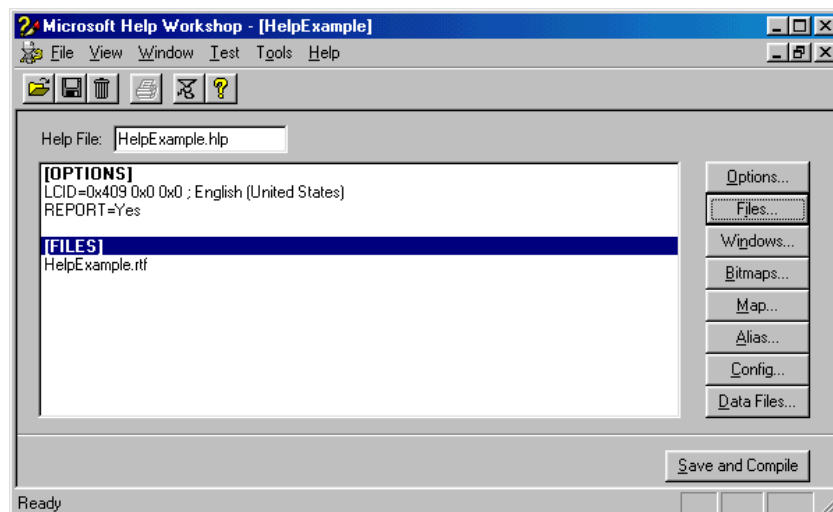
Choose **Close** under the **File** menu and assign a name of **HelpExample.cnt** to this contents file.

- Creating the Help Project File:

The **Help Project File** contains the information required by the Help Compiler to create the Help file. It, too, is created using the Microsoft Help Compiler Workshop. Choose the **New** option under the **File** menu. Choose **Help Project**, then click **OK**. Assign a **Project File Name** (for this example, use **HelpExample**). The following window will appear:



We need to specify where our help topic file is located. Click **Files**, then **Add**. Locate **HelpExample.rtf**, then click **Open**. You will return to the **Topic Files** window. HelpExample.rtf should be selected. Click **OK**. The project file is now listed:



- Compiling the Help File:

This is the easiest step. After creating the project file in the Help Compiler Workshop, click the button marked **Save and Compile**. This saves your listed project file and creates the help file. The created file has the same name as your Help Project File with an **HLP** extension. Create **HelpExample.hlp**. You can now exit the Help Compiler Workshop.

- Attaching the Help File:

The final step is to **attach** the compiled **help file** to your Visual Basic application. As a first step, open the **Project Properties** window under the **Project** menu. Under **Help File**, select the name of your **HLP** file by clicking the ellipsis (...). This ties the help file to the application, enabling the user to press <F1> for help.

You can also add a Help item to your application using a command button (or some other control) that invokes help via its **Click** event. If you do this, you must write code to invoke the help file. The code involves a call to the Windows API function, **WinHelp**. The function declaration (from the API Text Viewer) for WinHelp is:

```
Declare Function WinHelp Lib "user32" Alias  
"WinHelpA" (ByVal hwnd As Long, ByVal lpHelpFile As  
String, ByVal wCommand As Long, ByVal dwData As Long)  
As Long
```

We also need a constant:

```
Const HELP_FINDER = &HB
```

This constant (not in the API Text Viewer, by the way) will display the Help files **Contents** tab upon invocation of WinHelp. There are other constants that can be used with WinHelp - this is just a simple example.

The **Declare** statement and constant definitions usually go in the general declarations area of a code module and made **Public**. If you only have one form in your application, then put these statements in the General Declarations area of your form (and declare them **Private**). Once everything is in-place, to invoke the Help file from code, use the function call:

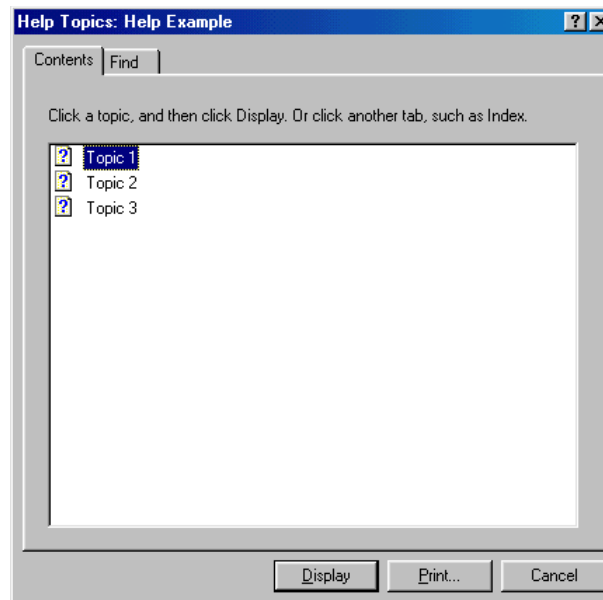
Dim R As Long

```
.  
.  
.  
R = WinHelp(myform.hWnd, filename.HLP, HELP_FINDER, CLng(0))
```

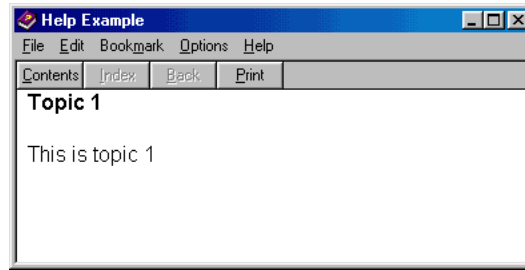
where *myform* is the name of the project 'startup' form and *filename* is the help file name, including path information (a string variable).

- Help File Example:

We can now try our example help file in a Visual Basic application. We'll only use the <F1> option here. Start a new application. Bring up the **Project Properties** window via the **Project** menu. Select the correct Help File (**HelpExample.hlp**) by clicking the ellipsis and finding your newly created file. Click **OK**. Now, run your application (I know there's nothing in the application, but that's all right). Once, it's running press <F1>. This Help screen should appear:



Double-click **Topic 1** and the corresponding Topic 1 screen appears:



Look at the other topics, if you'd like (yeah, I know they're not real exciting to look at). Click the **Find** tab and a searchable list of words will be built for your help system. All this power comes free with the Windows help system.

- After all this work, you will still only have a simple help file, nothing that rivals those seen in most applications. But, it is a very adequate help system. To improve your help system, you need to add more features. Investigate the Help Compiler Workshop for suggestions such as context-sensitive help (we'll use this in the next chapter), graphics, and help macros.
- For Visual Basic 6 users, an additional help building system is available: the **HTML Help Workshop**. This new wave of help systems incorporates Internet browser technology for display. You might like to study this system to see the "latest and greatest."

Example 6-8

Authors Table Input Form (On-Line Help Systems)

We will build a simple help system for our Authors Table Input Form and attach it to our application. Refer back to the notes to complete each step listed here.

1. Using your word processor, write a **Help Text File** with a single topic. The topic and text I used are:

#Authors Input Form

Available options for managing Authors database table:

Add New Record

Click the **Add New** button to add a record to the Authors database. Type in the Author Name (required), then Year Born (optional). Click **Save** to save the new record; click **Cancel** to ignore the new record.

Edit Record

Click the **Edit** button to edit the displayed record. Make any needed changes. The Author Name is required and the Year Born is optional. Click **Save** to save the changes; click **Cancel** to ignore the changes.

Delete Record

Click the **Delete** button to delete the displayed record.

Exit Program

Click the **Done** button to quit the application.

#Topic1

Establish a topic ID (we used **Topic1**) footnote for the one topic. Save this file as **Authors.rtf**.

2. Create a contents file for this example. Name it **Authors.cnt**.
3. Create a project file for this example. Name it **Authors.hpj**.
4. Save and compile your help file (named **Authors.hlp**).

5. Load Example 6-7 completed earlier.

ADO Data Control Modification

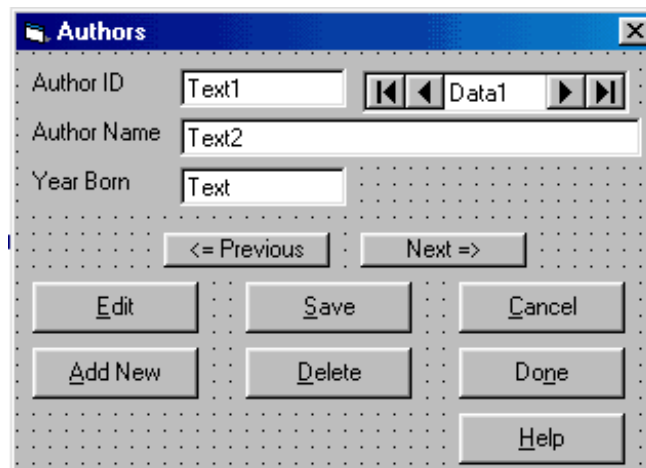
Load Example6-7AD (the ADO data control version).

ADO Data Environment Modification

Load Example6-7DE (the ADO data environment version).

We will modify this example to include our help system. Choose the project **Properties** option under the **Project** menu item. Select **Authors.hlp** as your help file. Save the project. Run it. Press **<F1>** and the help system should magically appear.

6. Add a command button to the form. Assign a **Caption** of **&Help** and a **Name** of **cmdHelp**. The form now looks like this:



7. Using the API Text Viewer (we assume you know how to use this – if not, consult the Visual Basic documentation, or on-line help), copy the Declare statement for the **WinHelp** API function. Place this statement in the General Declarations area (also add the constant **HELP_FINDER**, which for some reason is not included with the API Text Viewer):

```
Private Declare Function WinHelp Lib "user32" Alias  
"WinHelpA" (ByVal hwnd As Long, ByVal lpHelpFile As  
String, ByVal wCommand As Long, ByVal dwData As Long) As  
Long  
Private Const HELP_FINDER = &HB
```

8. Attach this code to the **cmdHelp_Click** event procedure:

```
Private Sub cmdHelp_Click()  
Dim Rtn As Long  
Rtn = WinHelp(frmAuthors.hwnd, App.HelpFile, HELP_FINDER,  
CLng(0))  
End Sub
```

We've used the Visual Basic **App** object here to point to the help file. In Step 5, we set the help file location as a project property. When this is done, Visual Basic saves the path to the help file in the variable **App.HelpFile**. By not 'hard-coding' the help file path in the WinHelp call, no code change is needed if the help file location or name is changed at a later time.

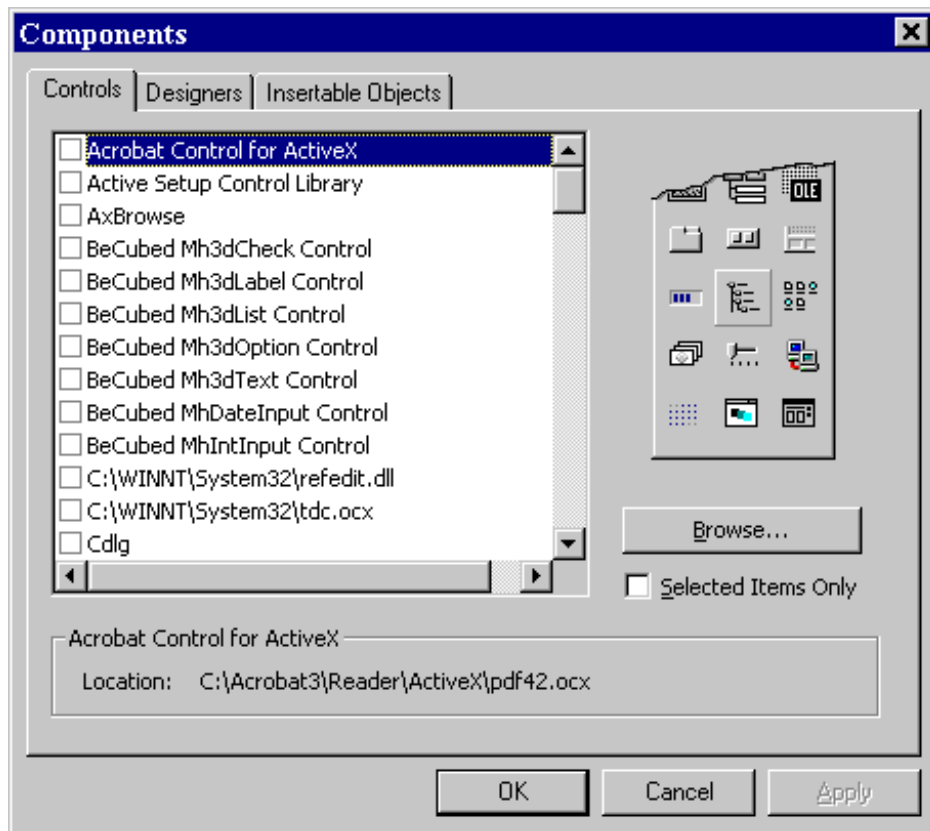
9. Save and run the application. Click **Help**. You now have two ways to access your on-line help system.

Application Testing

- Our discussion of Visual Basic interface design has, for now, come to an end. And, we have a fairly complete interface for the books database Authors table. We, obviously, still need the remainder of the code that goes behind the command buttons. We'll do that in the next chapter.
- Once you have completed an application, you need to test it to make sure it performs as expected. If you are careful in building your application, no big surprises should appear in this final testing. In fact, the Visual Basic environment helps achieve this goal. The event-driven nature of Visual Basic makes it easy to build an application in stages, testing and debugging each stage as it is built. In other words, you don't have to have a complete application before testing can begin. We have done this with the Authors Table example.
- The event-driven nature of Visual Basic also makes it easy to modify an application. We will see in Chapter 7, as we modify the books database example, that we have made some omissions and errors in our design. But these omissions and errors will be easily corrected using the Visual Basic environment. These corrections will give you additional insight into application building and testing process.
- Let others (particularly potential users) try your application and see if its use is as obvious as you planned it to be. Are the inputs and outputs of the project appropriate? Is application state clear? Implement and retest any necessary changes, based on user feedback. And, keep track of all feedback after you 'release' your application. This information can be used in future updates of your product.
- Before leaving this chapter, let's look at some other Visual Basic controls that you might like to use in your database interface arsenal – so-called, custom controls.

Custom Controls

- In addition to the standard Visual Basic controls discussed earlier in this chapter, there are many **custom controls** that can be used to build a database interface. These controls must be added to the Visual Basic toolbox before they can be used. In most cases, there are separate sets of custom controls, depending on whether you are using DAO or ADO data access.
- To load a custom control, choose **Components** from the Visual Basic **Project** menu. The **Components** (custom control) dialog box is displayed (make sure the **Controls** tab is selected):



To add a control or controls, select the check box next to the desired selection(s). When done, choose **OK** and the selected control(s) will now appear in the toolbox. Each custom control has its own set of properties, events, and methods. The Visual Basic on-line help system can provide details for usage.

- Here, we look at several custom controls (some data bound, some not) and how they can be used with a Visual Basic interface. Several examples using the DAO versions of the controls will be provided.

Masked Edit Control



- The **masked edit control** is a data bound control used to prompt users for data input using a mask pattern. It works like a text box, but the mask allows you to specify exactly the desired input format. In a **database**, this control could be used to prompt for a date, a time, number, or currency value. Or it could be used to prompt for something that follows a pattern, like a phone number or social security number. Use of this control can eliminate many of the entry validation problems mentioned earlier in the chapter. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO/ADO Version Microsoft Masked Edit Control

- Masked Edit Properties:

DataField	Field in database table, specified by DataSource (or DataMember), bound to masked edit control (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the masked edit control is bound to.
Mask	Determines the type of information that is input into the control. It uses characters to define the type of input. Check on-line help for mask formatting.
Text	Contains data entered into the control (including all prompt characters of the input mask). This is the property bound to the database.

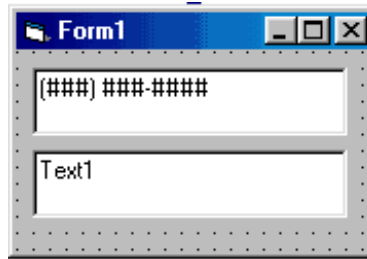
- Masked Edit Events:

Change	Event called when the data in the control changes.
Validation Error	Event called when the data being entered by the user does not match the input mask.

- This control features built-in input validation to lessen your tasks as a programmer. We will use the masked edit control in some of our example applications in later chapters.

Quick Example 1 - Masked Edit Control

1. We'll use the masked edit control to input a phone number. Start a new project. Add a masked edit control to the toolbox. Place a masked edit control and text box on your form.
2. Set the masked edit control **Mask** property to: **(###) ###-####**. This mask will allow an area code, followed by a phone number. Your form should look like this:



3. Place this code in the **MaskedTextBox1_Change** event:

```
Private Sub MaskedTextBox1_Change()  
    Text1.Text = MaskedTextBox1.Text  
End Sub
```

4. Save the application and run it. Notice how simple it is to fill in a correct phone number. The text box reflects what is typed in the masked edit control. The **Text** property of the masked edit control is bound to the database.

UpDown Control



- The **UpDown control** is a pair of arrow buttons that the user can click to increment or decrement a value. It works with a **buddy control** that uses the UpDown control's **Value** property. It is used in **databases** to select from a relatively small (less than 30 or so) list of items. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO Version **Microsoft Windows Common Controls-2 5.0**
ADO Version **Microsoft Windows Common Controls-2 6.0**

Several other controls are loaded with this group.

- UpDown Control Properties:

BuddyControl	Name of buddy control
BuddyProperty	Property in buddy control established by Value property of UpDown control.
Max	Establishes upper limit for value property.
Min	Establishes lower limit for value property.
Increment	Amount control value changes each time an arrow is clicked.
Orientation	Determines whether arrows lie horizontally or vertically.
Value	Current control value.

- UpDown Control Events:

Change	Invoked when value property changes.
UpClick	Invoked when up arrow is clicked.
DownClick	Invoked when down arrow is clicked.

- The UpDown control is a 'point-and-click' type control that can be used in place of a user's typed input. We will use the UpDown control in some of our example applications in later chapters.

Quick Example 2 - UpDown Control

1. We can use two UpDown controls to input a user's birthdate. Start a new application. Add the UpDown control to the toolbox. Place two UpDown controls and two label controls on the form.
2. Set these properties for the UpDown controls (**UpDown1** will set the birth month, **UpDown2** will set the day):

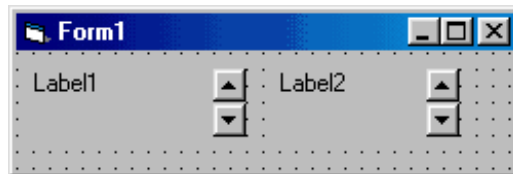
UpDown1:

Min	1
Max	12
Value	1

UpDown2:

BuddyControl	Label2
BuddyProperty	Caption
Value	1
Min	1
Max	31
Value	1

The form should look something like this:



3. Add these lines to the **General Declarations** area:

```
Option Explicit  
Dim Month(12) As String
```

4. Add this code to the **Form_Load** event (to name the months):

```
Private Sub Form_Load()  
Month(1) = "January": Month(2) = "February"  
Month(3) = "March": Month(4) = "April"  
Month(5) = "May": Month(6) = "June"  
Month(7) = "July": Month(8) = "August"  
Month(9) = "September": Month(10) = "October"  
Month(11) = "November": Month(12) = "December"  
Label1.Caption = Month(1)  
Label2.Caption = "1"  
End Sub
```

5. Add this code to the **UpDown1_Change** event:

```
Private Sub UpDown1_Change()  
Label1.Caption = Month(UpDown1.Value)  
End Sub
```

6. Save the application. Run the application. Birthdates (or any date for that matter) are now easy to input. Can you think of an additional validation you would need (Hint - 30 days has September, April, June and November ...)?

Tabbed Dialog Control



- The **tabbed dialog control** provides an easy way to present several dialogs or screens of information on a single form using the same interface seen in many commercial Windows applications. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box (notice this itself is a tabbed dialog control!):

DAO/ADO Version Microsoft Tabbed Dialog Control

- The tabbed dialog control provides a group of tabs, each of which acts as a container (works just like a frame or separate form) for other controls. Only one tab can be active at a time. Using this control is easy. Just build each tab container as a separate mini-application: add controls, set properties, and write code like you do for any application. Navigation from one container to the next is simple: just click on the corresponding tab.

- Tabbed Dialog Control Properties:

TabOrientation	Determines where the 'folder' tabs are positioned.
Tabs	Total number of displayed tabs.
TabsPerRow	Number of tabs in single row.

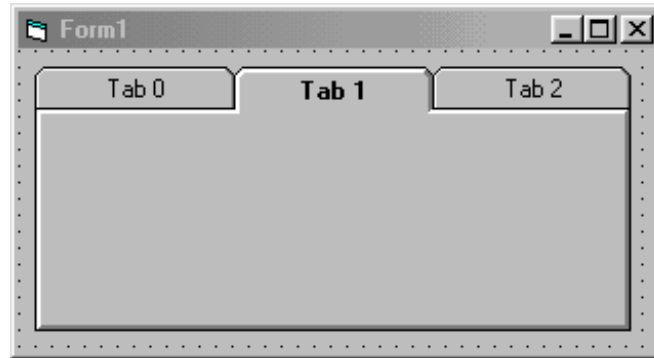
- Tabbed Dialog Control Event:

Click	Invoked when folder tab is clicked (allows you to determine which tab is active).
--------------	---

- The tabbed dialog control is becoming a very popular control in Windows applications. It allows you to put a lot of 'input power' into a single form - minimizing the need for multi-form applications. We will use the tabbed dialog control in some of our example applications in later chapters.

Quick Example 3 - Tabbed Dialog Control

1. Start a new application. Add the tabbed dialog control to the toolbox. Put a tabbed dialog control on the form:



Play with the various properties to see how different styles and layouts appear on the form.

2. Design each tab with some controls, then run the application. Note how each tab in the folder has its own working space.

Toolbar Control



- Almost all Windows applications these days use toolbars. A toolbar provides quick graphical access (by clicking a picture) to the most frequently used commands in an application. In a database application, it could be used to add, delete, or edit records. It could be used to access database reports or obtain different database views. The **toolbar control** is a mini-application in itself. It provides everything you need to design and implement a toolbar into your application. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO Version **Microsoft Windows Common Controls 5.0**
ADO Version **Microsoft Windows Common Controls 6.0**

Several other controls are loaded with this group.

- To create a basic toolbar, you need to follow a sequence of steps. You add buttons to a **Button** collection - each button can have optional text and/or an image, supplied by an associated **ImageList** control (another custom control). Buttons can have **tooltips**. In more advanced applications, you can even allow your user to customize the toolbar to their liking!
- Toolbar Control Properties:

Align	Establishes location of toolbar (usually at top of form).
ButtonHeight	Height of buttons on toolbar.
ButtonWidth	Width of buttons on toolbar.
ImageList	ImageList control containing pictures that appear on the toolbar (images are bitmap files).
Key	Keyword used to identify button (used to determine which button is clicked).
ShowTips	If True, tooltips appear as to what a particular button's function is.

- Toolbar Control Event:

ButtonClick	Invoked when one of the toolbar buttons is clicked.
--------------------	---

Quick Example 4 - Toolbar Control

1. We'll look at the simplest use of the toolbar control - building a fixed format toolbar (pictures only) at design time. We'll create a toolbar that replicates the function of the five command buttons used in Example 6-2: one to add a new record, one to save a record, one to cancel an edit, one to edit a record, and one to delete a record. Start a new project. Add the toolbar control to the toolbox. Place a toolbar, an imagelist control, and a label control on the form.
2. Right click on the image list control to set the pictures to be used. Using the **Images** tab, assign the following five images (click **Insert Picture**, then select graphic). The needed files are included with the course example code.

Image 1 - NEW.BMP

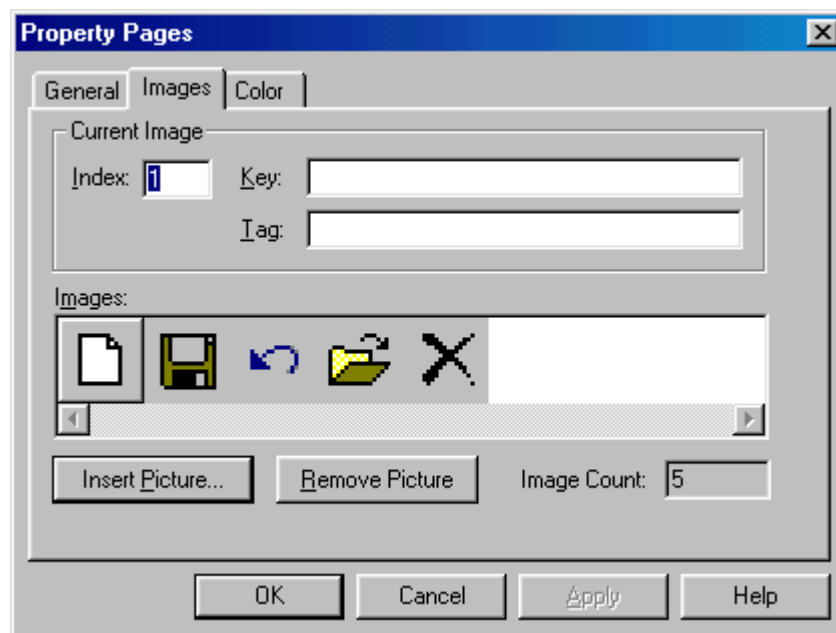
Image 2 - SAVE.BMP

Image 3 - CANCEL.BMP

Image 4 - EDIT.BMP

Image 5 - DELETE.BMP

When done, the image control should look like this:



Click **OK** to close this box.

3. Right mouse click on the toolbar control. The **Property Pages** dialog box will appear. Using the **General** tab, select the image list control (ImageList1) just formed. Now, choose the **Buttons** tab to define each button. A new button is added to the toolbar by clicking **Insert Button**. At a minimum, for each button, specify the **Key** (used to identify which button is clicked), **ToolTipText** property, and the **Image** number. Values I used are:

Index	Key	ToolTipText	Image
1	New	Add New Record	1
2	Save	Save Record	2
3	Cancel	Cancel	3
4	Edit	Edit Record	4
5	Delete	Delete Record	5

When done, my form looks like this:



4. Add this code to the **Toolbar1_ButtonClick** event:

```
Private Sub Toolbar1_ButtonClick(ByVal Button As
ComctlLib.Button)
Label1.Caption = Button.Key + " Clicked"
End Sub
```

5. Save and run the application. Click a button on the toolbar - the label control specifies which button was clicked. Check out how the tool tips work (hold the cursor over a button for a bit). The toolbar is a very powerful and professional tool. And, it's easy to implement and use. Try to use it whenever it fits the design of your interface.

Data Bound List Control



- The data bound list control allows you to display multiple rows of data in the same control, giving you a 'pick list' of values. It is automatically filled with a field from a specified data control. Selections from the list box can then be used to update another field from the same data control or, optionally, used to update a field from another data control. Note this is not the same as the standard list control, which is not data bound. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO Version	Microsoft Data Bound List Controls
ADO Version	Microsoft DataList Controls

- Data Bound List Control Properties:

BoundColumn	Name of field in Recordset specified by RowSource to be passed to DataField, once selection is made.
BoundText	Text value of BoundColumn field. This is the value passed to DataField property.
DataField	Name of field in table specified by DataSource (or DataMember) updated by selection.
DataSource	Name of data control (DAO or ADO) or data environment (ADO) that is updated by the selection.
DataMember	Specifies the Command object establishing the table updated by the selection (ADO data environment only).
ListField	Name of field in table specified by RowSource (or RowMember) used to fill list box.
RowMember	Specifies the Command object establishing the table used by ListField (ADO data environment only).
RowSource	Name of data control (DAO or ADO) or data environment (ADO) used as source of items in list box.
Text	Text value of selected item in list.

- Data Bound List Control Event:

Click	Invoked when item in list control is clicked.
--------------	---

- One use for the data bound list control is to fill the list (**ListField**) from the database (**RowSource**), then allow selections. This allows us to list all values of a particular field in a database recordset. The selections can be used by any control on a form, whether it is data bound or not.
- A powerful feature of the data bound list control is linking to (and updating) other fields in a database. This involves setting three more properties (**DataSource**, **DataField**, and **BoundColumn**). Here's the logic: **RowSource** is set to the data control establishing the recordset providing the information listed (**ListField**) in the list box. This establishes the **source** table. **BoundColumn** is the field name from the source table record used as a link. **DataSource** is the recordset linked by the BoundColumn (it can be the same recordset as RowSource or another recordset). Then, **DataField** is the field in DataSource that is linked by BoundColumn.
- The linking property of the data bound list box is widely used when adding to or updating database records. As an example, say you build a form to add books to the BIBLIO.MDB database. A new book arrives, but you don't that particular book publisher's identification (PubID), but you do know the publisher's name. To solve this problem, you use a data bound list box with Publisher names. When you pick a name from the list, you have the application find the PubID using the linking capabilities of the list box control. You are essentially hiding the key values (primary and foreign keys) of the database from the user and allowing the user to access information familiar to them, such as the publisher's name. Let's build this example in stages.

Quick Example 5 - Data Bound List Control

1. Start a new project. Add the data bound list control to the toolbox. Add a DAO data control and a data bound list control to the form.

2. Set these properties for the data control and list control:

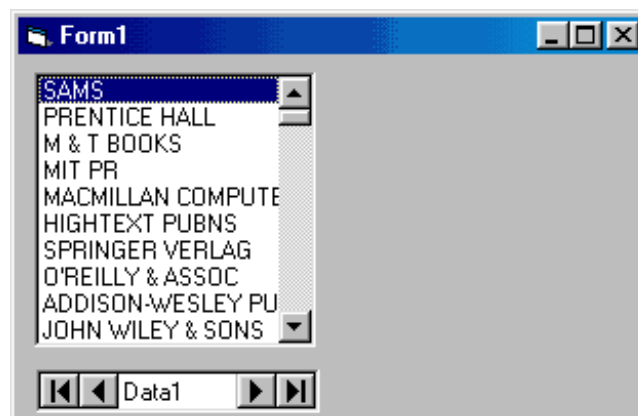
Data1:

DatabaseName	BIBLIO.MDB
RecordSource	Publishers
ReadOnly	True (to save us from mistakes)

DBList1:

RowSource	datPublishers
ListField	Company Name

3. Save the application and run it. You should see this:

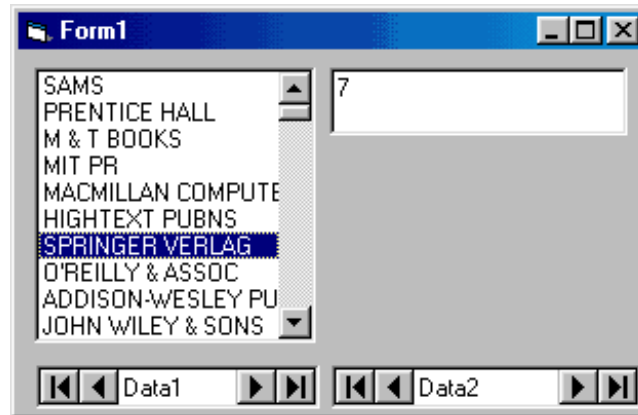


We have filled the list box with the Company Name field (**ListField**) from the Publishers database table (**RowSource**), simply by setting two properties!

4. Add a text box to the form. Set the **BoundColumn** property to **PubID** (this creates an output value based on the selected item in the list box). Add this code to the **DBList1_Click** event:

```
Private Sub DBList1_Click()  
Text1.Text = DBList1.BoundText  
End Sub
```

5. Save and run the application. Now, click on a publisher name in the list box and the corresponding **PubID** value will appear in the text box:



6. We now take this example one step further. We use the data produced by the **BoundText** property to link to a field in a separate database table. Recall the **Titles** table of the BIBLIO.MDB database lists a **PubID** field, but no publisher name information. We can link **PubID** in the **Titles** table with the **Company Name** field in the **Publishers** table by modifying this example. Add a second DAO data control - set it's properties to:

Data2:

DatabaseName	BIBLIO.MDB
RecordSource	Titles
ReadOnly	True (to save us from mistakes)

Set the data bound list control's **DataSource** property to **Data2** and the **DataField** property to **PubID**.

7. Save and run the application. Click the **Data2** data control navigation arrows to move from record to record in the **Titles** table. Notice the corresponding publisher name is highlighted in the list control and its PubID appears in the text box. Try adding a bound text box to also list the corresponding book title as you are navigating.

Data Bound Combo Control



- The **data bound combo control** is nearly identical to the data bound list box, hence we won't look at a separate set of properties or another example. A primary difference between the two controls is the way data is displayed – the combo control has a list box portion and a text box portion that displays the selected item. And, with the combo control, the user is (optionally) given the opportunity to type in a choice not in the list box. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO Version **Microsoft Data Bound List Controls**
ADO Version **Microsoft DataList Controls**

- As mentioned, data display is different with the combo control. Display is established by the **Style** property:

Style	Symbolic Constant	Description
0	VbComboDropDown	Drop-down list box, user can change selection
1	VbComboSimple	Displayed list box, user can change selection
2	vbComboDropDownList	Drop-down list box, user cannot change selection

When using Style = 1, make sure you sufficiently size the control (so the list box portion appears) when it is placed on the form.

- When should you use the combo control instead of the list box control? The data bound combo control is an excellent data entry control. Its advantage over the list box is that it provides experienced users the ability to type in values they know are correct, speeding up the data entry process. The list box control does not allow any typing. It is also a good control when you are short on form space. Using Style = 2 replicates the functionality of the list box control without needing space for the list box.

Data Bound Grid Control

- The **data bound grid control** tool is one of the most useful data bound controls. It can display an entire database table, referenced by a data control. The table can then be edited as desired. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog box:

DAO Version **Microsoft Data Bound Grid Control**
ADO Version **Microsoft DataGrid Control 6.0**

- The data bound grid control is in a class by itself, when considering its capabilities. It is essentially a separate, highly functional program. It has two primary properties:

DataMember Specifies the Command object establishing the database table (ADO data environment only).
DataSource Name of data control (DAO or ADO) or data environment (ADO) grid control is bound to.

- The data bound grid control is a collection of **Column** objects, corresponding to fields in the table, and rows, corresponding to records. Cells can be accessed and edited via mouse operations or programmatically.
- The data bound grid control allows you to monitor editing activities via several events:

BeforeInsert Occurs before a new row is inserted in the grid. Use to confirm addition of new record.
AfterInsert Occurs after a new row is inserted in the grid.
BeforeUpdate Occurs before changes are written to data control. Use to perform data validation.
AfterUpdate Occurs after changes are written to data control.
BeforeDelete Occurs before selected record(s) are deleted. Use to confirm deletion.
AfterDelete Occurs after selected record(s) are deleted.

- You are encouraged to further study the data bound grid control (properties, events, methods) as you progress in your database studies. We will use it in applications studied in later chapters.

Quick Example 6 – Data Bound Grid Control

1. Start a new project. Add a data control and a data bound grid control. Set these properties:

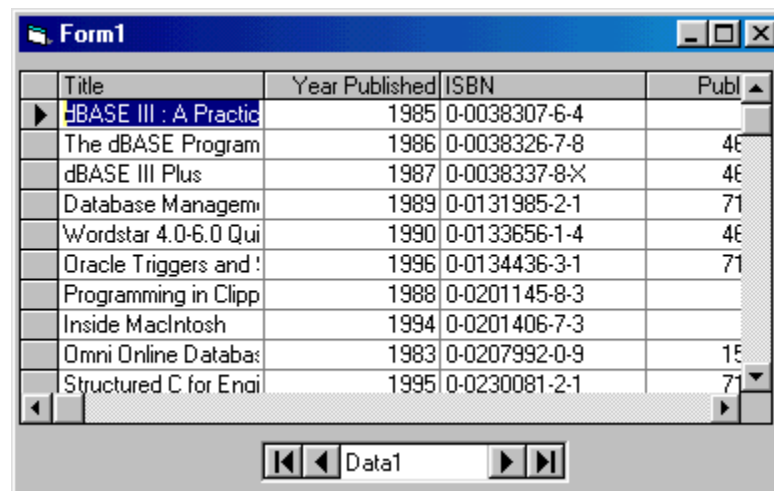
Data1:

DatabaseName	BIBLIO.MDB
RecordSource	Titles
ReadOnly	True

DBGrid1:

DataSource	Data1
------------	-------

2. Save and run the application. The following will appear:



At this point, we can scroll through the table and edit any values we choose. Any changes are automatically reflected in the underlying database. Column widths can be changed at run-time! Multiple row and column selections are possible! Like we said, a very powerful tool.

Data Bound FlexGrid Control



- The data bound grid control is an excellent tool for providing full-editing features in a spreadsheet like fashion. However, it is memory intensive and not real fast. If you only need display capabilities (no editing), consider the **data bound flexgrid control**. This control offers an amazing array of variations in how data can be displayed. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog:

DAO/ADO Version Microsoft FlexGrid Control

- Display of data in the flexgrid control is established by two properties:

DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Name of data control (DAO or ADO) or data environment (ADO) grid control is bound to.
- The flexgrid control offers a wide number of properties that allows you to display the data in any format you desire. You can change the color and style of nearly every piece of information in the grid (gridlines, fonts, colors, selections, ...). Examine the on-line help file for this control to enhance your knowledge of these properties.

Quick Example 7 – Data Bound FlexGrid Control

1. Start a new project. Add a data control and a data bound flexgrid control. Set these properties:

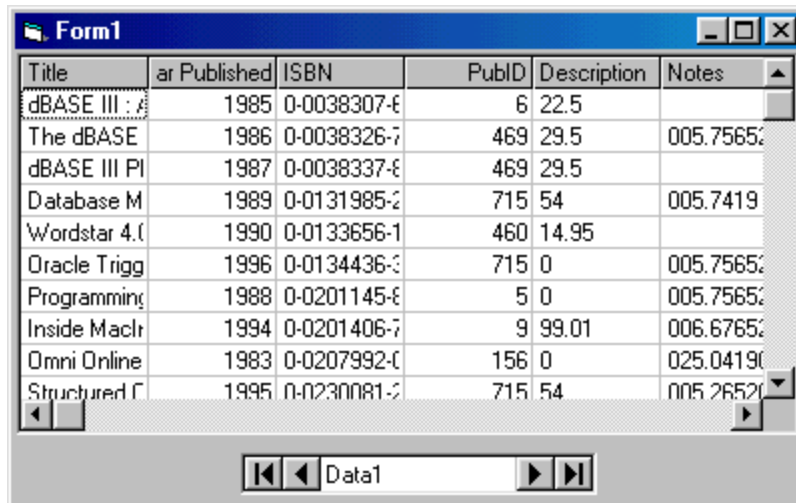
Data1:

DatabaseName	BIBLIO.MDB
RecordSource	Titles
ReadOnly	True

MSFlexGrid1:

AllowUserResizing	1 - flexResizeColumn
DataSource	Data1
FixedCols	0 (we only want data columns displayed)

2. Save and run the application. The following will appear:



All records (with columns labeled) from the Titles table appear. Notice you can only view the data – no editing is possible.

Calendar Control



- The **calendar control** is another tool for displaying and determining dates. It provides a simple to use interface to display any monthly calendar and select a date. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog:

DAO/ADO Version Microsoft Calendar Control

- Calendar Control Properties:

DataField	Field in database table, specified by DataSource (or DataMember), bound to calendar control (DAO or ADO).
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control (DAO or ADO) or data environment (ADO) the calendar control is bound to.
Day	The number (1-31) of the currently selected day.
Month	The number (1-12) of the currently selected month.
Value	The value of the selected date (property bound to data control or data environment). This is set to the current date by default.
Year	The number of the currently selected year.

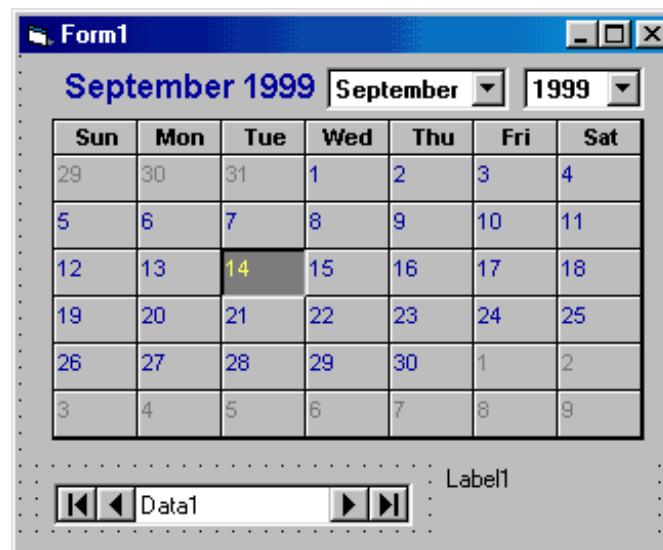
- Calendar Control Event:

Click	Invoked when user clicks on a date in the calendar control.
--------------	---

Quick Example 8 – Calendar Control

The books database we have been using throughout this chapter does not have a field with dates. Hence, here (and in some later examples needing dates and plotting values) we use the **Northwind Traders** database used at the end of previous chapters. This database (**NWIND.MDB**), like **BIBLIO.MDB**, ships with Visual Basic. If you have not already done so, move a copy of this file to your working directory.

1. Start a new application. Add the calendar control a DAO data control to your toolbox. Now add a calendar control, the data control, and a label control to your form. It should look like this:



2. Set these properties:

Data1:

DatabaseName NWIND.MDB (point to working copy)
RecordSource Orders

Calendar1:

DataSource Data1
DataField OrderDate

Label1:

DataSource Data1
DataField OrderDate

3. Save and run the application. Click the data control to move through the records of the Orders table. As you do, the Order Date will be shown on both the calendar control and in the label control.

Common Dialog Control



- In all examples studied in this course, the database name has been assumed to be known at design time (before running the application). There will be times when this is not true. For example, say a schoolteacher uses a database application to keep track of grades. There will database files for each class. When the teacher starts the application, he or she needs to specify which particular database file is being accessed and that file needs to be opened at run-time. Not only do we need the capability to open a user specified file, but we also need to be able to save database files with user specified names. At this point, we will not address how to open or save databases at run-time. This will be discussed in later chapters. As part of our discussion of custom controls, however, we will address how to get user specified file information.
- What we need from the user, whether opening or saving files, is a complete path to the filename of interest. We could provide a text box and ask the user to type the path, but that's only asking for trouble. We would have to validate existence of drives, directories, and files! Fortunately, we can use the Windows standard interface for working with files. Visual Basic provides this interface through the **common dialog control**. This control displays the same interface you see when opening or saving a file in any Windows application. Such an interface is familiar to any Windows user and gives your application a professional look. And, some context-sensitive help is available while the interface is displayed.
- Like other custom controls, if the common dialog control does not appear in the Visual Basic toolbox, you need to add it. To load this control, select **Components** from the **Projects** menu, then select from the **Components** dialog:

DAO/ADO Version Microsoft Common Dialog Control

- The common dialog control, although it appears on your form, is invisible at run-time. The control is used to display a dialog box using a **Show** method. For a common dialog box named **cdlExample**, to see the **Open** dialog box, use:

```
cdlExample.ShowOpen
```

To see the **Save As** dialog box, use:

```
cdlExample.ShowSave
```

You cannot control where the common dialog box appears on your screen. Once the dialog box is closed in some manner, control to the program returns to the line immediately following this line. Common dialog boxes are system modal.

- Common Dialog Control Properties:

CancelError	If True, generates an error if the Cancel button is clicked. Allows you to use error-handling procedures to recognize that Cancel was clicked.
DefaultExt	Sets the default extension of a file name if a file is listed without an extension.
DialogTitle	The string appearing in the title bar of the dialog box.
FileName	File name that appears in the File name box.
Filter	Used to restrict the filenames that appear in the file list box.
FilterIndex	Indicates which filter component is default.
Flags	Values that control special features of the common dialog control.

- The **Flags** property is an especially important one. Using proper values can insure any file name provided by the user is a valid one. For opening files with a common dialog control named cdlExample, we suggest:

```
cdlExample.Flags = cdlOFNFileMustExist +  
cdlOFNPathMustExist
```

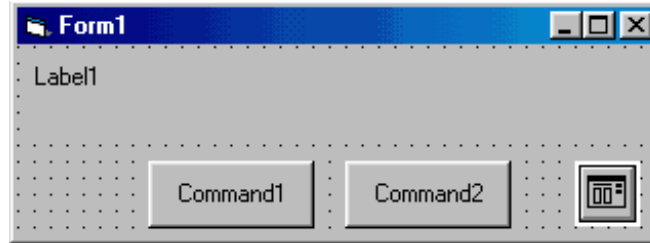
These values on the right are Visual Basic constants that insure both the specified file and path exist before trying to open it. For saving files, use:

```
cdlExample.Flags = cdlOFNOverwritePrompt +  
cdlOFNPathMustExist
```

The constants here will insure files are not overwritten without user concurrence and that any specified path must exist.

Quick Example 9 – Common Dialog Control

1. Start a new project. Add a common dialog control (set **CancelError** to **True**), two command buttons and a label control:



2. Attach the following code to the **Command1_Click** event procedure:

```
Private Sub Command1_Click()  
CommonDialog1.Flags = cdIOFNFileMustExist +  
cdIOFNPathMustExist  
On Error GoTo CancelPressed  
CommonDialog1.ShowOpen  
Label1.Caption = CommonDialog1.filename  
Exit Sub  
CancelPressed:  
Label1.Caption = "Cancel Pressed"  
Exit Sub  
End Sub
```

This code will display the **Open** dialog box when the command button is clicked. Note the use of error trapping to determine if **Cancel** is clicked.

3. Attach the following code to the **Command2_Click** event procedure:

```
Private Sub Command2_Click()  
CommonDialog1.Flags = cdIOFNOverwritePrompt +  
cdIOFNPathMustExist  
On Error GoTo CancelPressed  
CommonDialog1.ShowSave  
Label1.Caption = CommonDialog1.filename  
Exit Sub  
CancelPressed:  
Label1.Caption = "Cancel Pressed"  
Exit Sub  
End Sub
```

This code will display the **Save** dialog box when the command button is clicked.

4. Save the application and run it. Click **Command1**. The **Open** dialog box appears. Type in garbage for a filename. Click **Open**. The filename will not be accepted. Type (or select) a valid name. Click **Open**. The complete path to the file will be displayed in the label control. We would use this name to open a database file. Click **Command1** again and make sure **Cancel** works.
5. Click **Command2**. The **Save** dialog box appears. Type in garbage for a file path. Click **Save**. The entry will not be accepted. Type in (or select) an existing filename. You will be asked if you want to overwrite the file. Answer **No** (even though there is no possibility of overwriting – we have no code to do such). Type (or select) a valid name. Click **Save**. The complete path to the file will be displayed in the label control. We would use this name to save a database file. Click **Command2** again and make sure **Cancel** works.

Additional ADO Custom Controls

- The introduction of Visual Basic 6 brought several new custom controls that can be of use to us in our database applications. These controls only work with the ADO data control and ADO data environment discussed in Chapter 4.
- Here, we look at several ADO custom controls (some data bound, some not) and how they can be used with a Visual Basic interface. Several examples using the ADO data control will be provided.

Hierarchical FlexGrid Control



- The **hierarchical flexgrid control** is a new version of the flexgrid control studied earlier. It features the ability to merge, sort, and format data tables. The displayed data still cannot be edited. However, entire rows can be selected, range selections (like Excel spreadsheets) can be made, and 'drag and drop' of cells is possible. To load this control, select **Components** from the **Projects** menu, and then select **Microsoft Hierarchical FlexGrid Control 6.0**.
- Display of data in the hierarchical flexgrid control is established by two properties:

DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Name of data control (DAO or ADO) or data environment (ADO) grid control is bound to.
- The hierarchical capabilities of this control require creation of hierarchical recordsets that are somewhat complicated. We will not address such recordsets here. We will, however, develop an example that illustrates some of the merging power of the control.

Quick Example 10 – Hierarchical FlexGrid Control

1. Start a new application. Add a hierarchical flexgrid control and an ADO data control to the form. Both of these controls will need to be added to your toolbox. Set these properties:

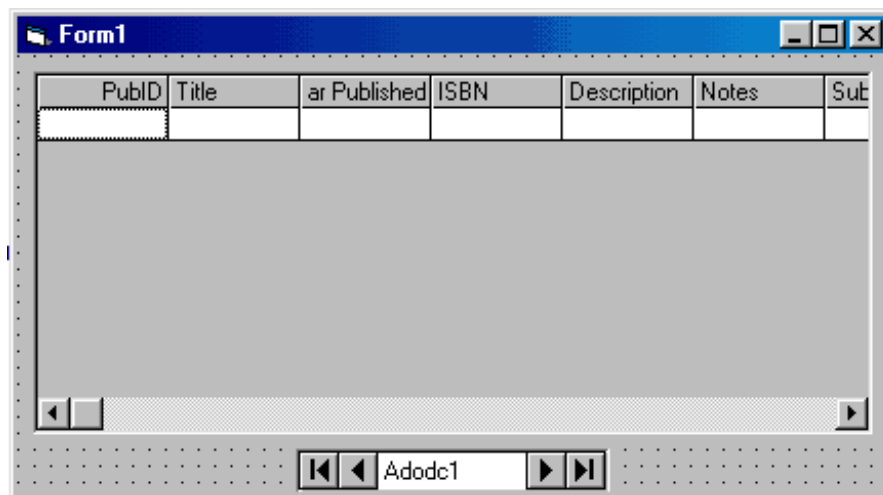
Adodc1:

ConnectionString	BIBLIO.MDB (point to working copy)
CommandType	2-adCmdTable
RecordSource	Titles

MSHFlexGrid1:

DataSource	Adodc1
------------	--------

2. Right-click the flexgrid control and click **Retrieve Structure**. This control allows you to see the table structure at design time (not possible with the old flexgrid control). Right-click again and choose **Properties**. Make the General tab active. Set **Fixed Cols** to **0** (to eliminate the blank column at the left). Set **AllowUserResizing** to **1-Columns**.
3. Click the **Bands** tab. We want to merge fields. The first column is used for merging. We want it to be a key, **PubID**. Select **PubID** in **Column Caption**, then click the **Up** arrow until that field moves to the top of the list. PubID should now be the first field listed:



4. We need some code to do the merge. Here it is (don't get too worried if this isn't obvious) – it goes in the **Form_Activate** event:

```
Private Sub Form_Activate()
Dim I As Integer
MSHFlexGrid1.Row = 0
For I = 0 To MSHFlexGrid1.Cols - 1
    MSHFlexGrid1.Col = I
    MSHFlexGrid1.CellAlignment = 4
    MSHFlexGrid1.MergeCol(I) = True
Next I
MSHFlexGrid1.Col = 0
MSHFlexGrid1.ColSel = MSHFlexGrid1.Cols - 1
MSHFlexGrid1.Sort = flexSortGenericAscending
MSHFlexGrid1.MergeCells = flexMergeRestrictColumns
End Sub
```

This code sets the alignment (right justified) on all the columns and makes them 'mergeable.' The last line tells cells to merge if they match cells above (in this case, all records with the same PubID will merge).

5. Save the application and run it. You should see:

PubID	Title	Year Published	ISBN	Description	Notes
1	dBASE III : /	1985	0-0038307-6	22.5	
	Advanced C	1986	0-2621321-5	45	005.7419
	Advances in	1994	0-2621118-8	47.5	006.320
	Commentary	1991	0-2621327-1	30	005.13320
4	Data Models	1980	0-2620215-1	42.5	001.642
	Essentials of	1992	0-0702244-3	0	005.1320
			0-2620614-5	65	
	Evolutionary	1995	0-2621331-7		006.320
	From Logic to		0-2620414-2	32.5	005.120

All the publishers are now grouped. Not bad for just a few lines of code.

Chart Control



- The **chart control** is an amazing tool. In fact, it's like a complete program in itself. It allows you to design all types of graphs interactively on your form. Then, at run-time, draw graphs, print them, copy them, and change their styles. To load this control, select **Components** from the **Projects** menu, and then select **Microsoft Chart Control 6.0**.

- Chart Control Properties:

Column	Returns or sets the current column in the grid of data being charted.
ChartType	Establishes the type of chart to display.
Data	Returns or sets the data in the current row and column of the grid of data being charted.
DataMember	Specifies the Command object establishing the database table (ADO data environment only).
DataSource	Specifies the data control or data environment the chart control is bound to.
RandomFill	Used to fill chart with random values (good for checking out chart at design-time). Data is normally filled from established array.
Row	Returns or sets the current row in the grid of data being charted.

Visual Basic 5 users might be asking: isn't there a Chart control we can use with the DAO data control? Yes, but it is not data bound and requires programmatic storing of data for plotting. It is a straightforward task, but will not be addressed here.

- The chart control is valuable. It allows a user a quick overview of any numerical data within a database. We will use it in some of our later applications.

Quick Example 11 – Chart Control

Like an earlier example, this example uses the **Northwind Traders** database to have a value for charting.

1. Start a new project. Add a chart control (make sure it is the Visual Basic 6 version) and ADO data control to the form. Set these properties:

Adodc1:

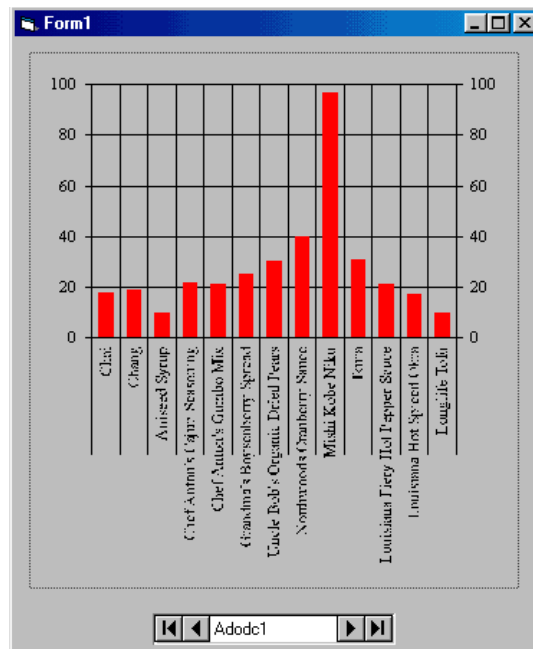
ConnectionString	NWIND.MDB (point to working copy)
CommandType	1-adCmdText
RecordSource	SELECT ProductName, UnitPrice FROM Products WHERE SupplierID < 5

MSChart1:

DataSource	Adodc1
------------	--------

We restrict the SupplierID value to limit the number of records returned.

2. Save the application, then run it. A bar chart of UnitPrice versus ProductName should appear:



Magic, huh? If all the labels don't appear in your example, you need to enlarge the size of the control on your form.

Month View Control



- A common need for a database field is a properly formatted date or time. We could allow the user to type the information in a text box, but we know that is asking for trouble (remember the validations needed for typed input). Visual Basic offers several data bound date entry tools. We look at the first of three here: the **month view control**. This control provides a clickable calendar that lets users view and set date information. Single dates or date ranges may be selected. Up to 12 months at a time can be displayed. To load this control, select **Components** from the **Projects** menu, and then select **Microsoft Windows Common Controls-2 6.0**.

- Month View Control Properties:

DataField	Field in database table, specified by DataSource (or DataMember), bound to month view control.
DataMember	Specifies the Command object establishing the database table (data environment only).
DataSource	Specifies the data control or data environment the month view control is bound to.
DateClicked	Date value clicked in DateClick event.
Day	The number (1-31) of the currently selected day.
Month	The number (1-12) of the currently selected month.
Value	The value of the selected date (property bound to data control or data environment). This is set to the current date by default.
Week	The number (1-52) of the currently selected week.
Year	The number of the currently selected year.

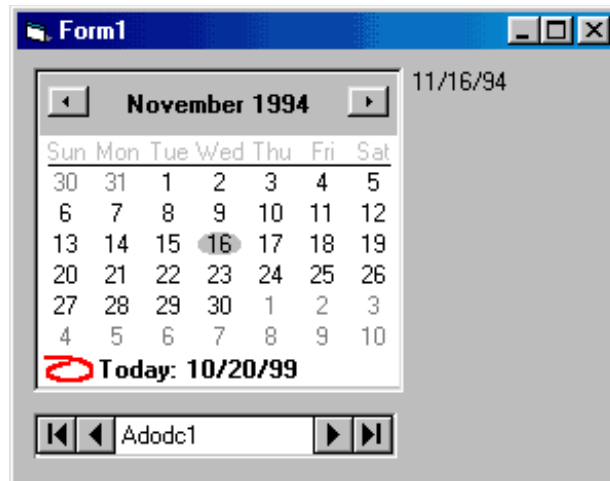
- Month View Control Event:

DateClick	Invoked when user clicks on a date in the month view control.
------------------	---

- This control has many more properties for multiple month displays, multiple date selections, and appearance, that you might like to investigate. The month view control will be used in some of our later examples.

Quick Example 12 – Month View Control

1. Start a new application. Add the month view control and ADO data control to your toolbox. Now add a month view control, the data control, and a label control to your form. It should look like this (note the current month will display – you now know when this was being written):



2. Set these properties:

Adodc1:

ConnectionString	NWIND.MDB (point to working copy)
CommandType	2-adCmdTable
RecordSource	Orders

MonthView1:

DataSource	Adodc1
DataField	OrderDate

Label1:

DataSource	Adodc1
DataField	OrderDate

3. Save and run the application. Click the data control to move through the records of the Orders table. As you do, the Order Date will be shown on both the month view control and in the label control. Notice you can change the displayed month using the arrows at the top (or try the standard cursor control keys). If you click a new date, then move off that record using the data control, that date becomes the new order date (field is automatically updated). Try it if you like, recognizing you are changing the database.

Date Time Picker Control

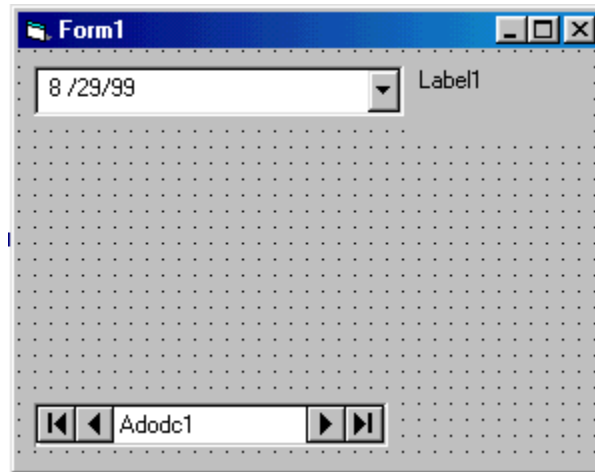


- The **date time picker control** is related to the month view control and shares most of the same properties. The difference here is that the selected date appears in a masked edit control and the calendar portion works in dropdown mode. The returned value can be formatted in many ways.
- Date Time Picker Control Properties (in addition to those given previously for the month view control):

CheckBox	Specifies whether control returns a date (allows optional use of control).
CustomFormat	Formatting string that determines how date and/or time will be displayed.
Format	Determines how date and/or time will be displayed. You can choose a predefined option or use the custom format feature.
UpDown	Determines control mode. When False, operates in Dropdown Calendar Mode (default). When True, user can increment or decrement displayed value using UpDown control (Time Format Mode).

Quick Example 13 – Date Time Picker Control

1. Start a new application. Add the date time picker control and ADO data control to your toolbox. Now add a date time picker control, the data control, and a label control to your form. It should look like this:



2. Set these properties:

Adodc1:

ConnectionString	NWIND.MDB (point to working copy)
CommandType	2-adCmdTable
RecordSource	Orders

DTPicker1:

DataSource	Adodc1
DataField	OrderDate
Format	3-dtpCustom
CustomFormat	'Order Date is:'MMMMdd', 'yy" (these are all single quotes in this string)

Label1:

DataSource	Adodc1
DataField	OrderDate

3. Save and run the application. Click the data control to move through the records of the Orders table. As you do, the Order Date will be shown in the date time picker control and in the label control. Play with some of the other features if you like.

Data Repeater Control



- The **data repeater** control allows you to display more than one database record at a time, without using a grid type control. You decide how you want to display each record by building your own Visual Basic control! The data repeater then hosts this new control (displayed in the Visual Basic toolbox like all other controls), displaying several instances in a Visual Basic application. To load this control, select **Components** from the **Projects** menu, and then select **Microsoft DataRepeater Control**.
- To use the data repeater control requires a bit of work, but it is worth it. There are four steps: (1) build the ActiveX control to display the database record, (2) expose properties in the ActiveX control, (3) compile the ActiveX control and (4) bind the ActiveX control to the data repeater control. Rather than discuss these steps in general, we will build an example using the Publishers table from the books database. This is, by far, the best way to illustrate how this powerful control is used.

Example 6-9

Data Repeater Control

In this example, we will use the data repeater control to display multiple publisher names and phone numbers (a 'computer Rolodex'). This information will come from the Publishers table of the BIBLIO.MDB database. This example will follow the four steps just outlined: (1) build the ActiveX control that displays the record, (2) expose the ActiveX control properties, (3) compile the ActiveX control, and (4) bind the ActiveX control to the data repeater control.

Build an ActiveX Control

Building an ActiveX control is not any harder than building a Visual Basic application. We build the control, compile it, and register it in the Windows registry. The control we build here will contain two pieces of information from the Publishers table of BIBLIO.MDB – the Company Name field and the Telephone field.

1. Start a new Visual Basic project. In the **New Project** window, select **ActiveX Control**. Select the **Project** menu item and choose **Project Properties**. In this window, assign a **Project Name** of **vbdbPublisher**. This name will be used as a prefix for the compiled control.
2. Place two label controls and two text boxes on the user control form. Assign these properties:

UserControl1:

Name	ctlPublisher
------	--------------

Label1:

Caption	Publisher
---------	-----------

Text1:

Name	txtPublisher
------	--------------

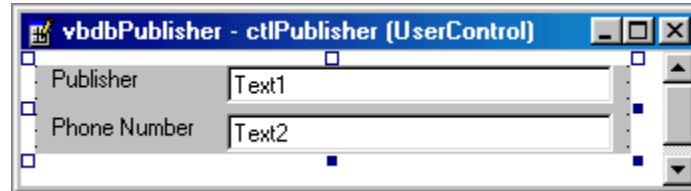
Label2:

Caption	Phone Number
---------	--------------

Text1:

Name	txtPhone
------	----------

Resize the control form so it just fits the space. My form looks like this:



3. Save the control (**ctlPublisher.ctl**) and project (**vbdbPublisher.vbp**) to make sure you don't lose anything.

Expose ActiveX Control Properties

We need to add code to our ActiveX control to make the text box controls **Text** properties available for database binding within the data repeater control. We need to make it possible to both read and set these properties.

1. Open the form code window. Under **Tools** menu, select **Add Procedure**. Set **Type** to **Property**, **Scope** to **Public**, **Name** to **PublisherName**. Repeat these steps to create a **Property** named **PublisherPhone**.
2. The previous steps created the code structures that allow us to read (**Property Get** statements) and set (**Property Let** statements) property values. The code for these structures is:

```
Public Property Get PublisherName() As String
PublisherName = txtPublisher.Text
End Property
```

```
Public Property Let PublisherName(ByVal vNewValue As
String)
txtPublisher.Text = vNewValue
End Property
```

```
Public Property Get PublisherPhone() As String
PublisherPhone = txtPhone.Text
End Property
```

```
Public Property Let PublisherPhone(ByVal vNewValue As
String)
txtPhone.Text = vNewValue
End Property
```

This code is pretty obvious – it just makes sure proper values are available. One thing to note: by default, all arguments and returned values are defined to be **Variant**. Make sure, in this example, you change all these default types to **String** (our property types).

3. We also need to let the data repeater control know when any of the **Text** properties change. Add this code to the two text box **Change** events:

```
Private Sub txtPhone_Change()  
    PropertyChanged "PublisherName"  
End Sub
```

```
Private Sub txtPublisher_Change()  
    PropertyChanged "PublisherName"  
End Sub
```

Under the **Tools** menu, choose **Procedure Attributes**. Click on **Advanced** button. For both properties, under **Data Binding** check **Property Is data bound** and **Show in data bindings collection at design time**.

4. Resave the application.

Compile the ActiveX Control

We're now ready to compile and register the control so it becomes available for the data repeater (and any other Visual Basic project you may build). Choose the **File** menu option and click **Make vbdbPublisher.ocx**. Click **OK**. If you there are no errors in your code, the control will be compile and an OCX file saved in the directory you specify. Your control is now ready to use with the data repeater.

Bind the ActiveX Control to the Data Repeater Control

1. Start a new **Standard EXE** application (a Visual Basic project). Add the data repeater control and an ADO data control to your toolbox, then add them to your form. Set these properties:

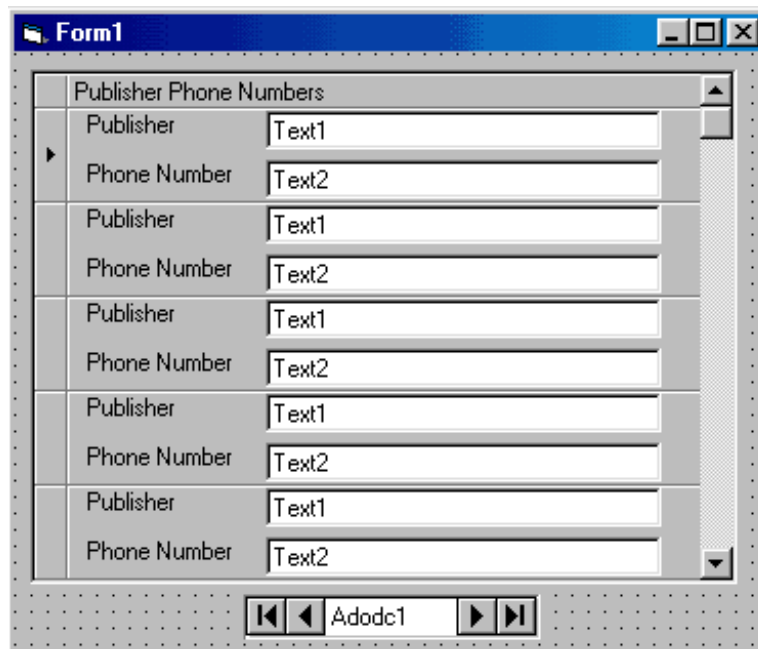
Adodc1:

ConnectionString	BIBLIO.MDB (point to working copy)
CommandType	2-adCmdTable
RecordSource	Publishers

DataRepeater1:

Caption	Publisher Phone Numbers
DataSource	Adodc1
RepeatedControlName	vbdbPublisher.ctlPublisher

To set the RepeatedControlName property, click the drop-down arrow that appears. You will be shown a list of all qualifying controls. Select the one we just created (**vbdbPublisher.ctlPublisher**). When you do this, the form is populated with the control. My form looks like this (you may have to do some resizing to get things to fit nicely):



2. We're almost done – just one last step. We need to bind the properties of our ActiveX control to fields in the database table connected to the data control (which is connected to the data repeater). Right-click the data repeater control and choose **Properties**. Select the **RepeaterBindings** tab. In the **PropertyName** combo box, choose **PublisherName**. In the **DataField** combo box, choose **Company Name**. Click **Add** to add the binding to the list box. Next, bind the **PublisherPhone** property to the **Telephone** data field. Click **OK** when done.

3. Save the application. Run it and you should see:

Publisher Phone Numbers	
Publisher	O'REILLY & ASSOC
Phone Number	
Publisher	ADDISON-WESLEY PUB CO
Phone Number	617-944-3700
Publisher	JOHN WILEY & SONS
Phone Number	212-850-6000
Publisher	SINGULAR PUB GROUP
Phone Number	
Publisher	Duke Press
Phone Number	

Navigation: [First] [Previous] Adodc1 [Next] [Last]

You now have a scrollable list of Publisher phone numbers. Pretty cool, huh? This is a great tool for display multiple records without using the grid tools.

Summary

- There is wealth of material covered here. You now have a complete reference to the Visual Basic toolbox and how those tools can be used for proper interface design. The Visual Basic interface is very important and we wanted to make sure you have many tools at your disposal. This will make your (and your user's) task much easier.
- Even with all this work, our interface is not complete. We still need code that allows us to edit, add, and delete records from a database. We need to know how to validate and save changes properly. We need to know how to 'undo' unwanted changes. We need to know how to properly exit an application. These topics, and more, are covered in the next chapter where we learn to design the total database management system.

Exercise 6

Publishers Table Input Form

In this chapter, we built the framework for an interface that allows us to maintain the **Authors** table in the books database (BIBLIO.MDB). This framework will be modified in the next chapter and implemented as part of a complete database management system. This database management system will also need interfaces to maintain the **Publishers** and **Titles** tables. The Titles table interface is a little tricky, in that it uses foreign keys to reference information in other tables. We will develop this interface in the next chapter. As an exercise here, we will begin the interface to maintain the **Publishers** table.

We will follow the same steps used in this chapter to build the Authors table input form:

- Build interface
- Add message box(es)
- Code application state
- Perform entry validation
- Perform input validation
- Add error trapping and handling
- Add on-line help system
- Application testing

Rather than starting from scratch, however, we will follow a 'tried and true' programming method – adapting an existing application to a new use. The Publishers table interface will essentially be the same as the Authors table interface. It will just have more (and different) input fields. Adapting an existing application saves us programmers a lot of time. You do have to make sure the modification implements the needs of the new application while at the same time eliminates vestiges of the old application. This exercise illustrates the modification steps followed and crosschecks required. An important step: **Save** your work often. You want to make sure your changes are always there.

Recall these examples use the **DAO data control**. Modifications needed to use the **ADO data control** or **ADO data environment** are given in shaded boxes. Recall the naming convention we use for our files: no suffix (DAO), AD suffix (ADO data control), and DE (ADO data environment).

Build Interface

1. Open Example 6-8 (the last version of the Authors table input form). Immediately resave the form with the name **Exercise6.frm**. Resave the project as **Exercise6.vbp**.

ADO Data Control Modifications

Load Example6-8AD (the ADO data control version). Resave the form as **Exercise6AD.frm**. Resave the project as **Exercise6AD.vbp**.

ADO Data Environment Modifications

Load Example6-8DE (the ADO data environment version). Resave the form as **Exercise6DE.frm**. Resave the project as **Exercise6DE.vbp**.

We now have a copy of the Authors table input form project to modify to a Publishers table input form. The Publishers table has ten (10) fields that must be input:

PubID
Name
Company Name
Address
City
State
Zip
Telephone
Fax
Comments

We need a label and text box for each of these fields. We could use ten separate label and text box controls, each with separate names and properties. A better solution is to use control arrays to simplify bookkeeping chores and navigation among controls.

2. Delete the label and text box control for the **Author** and **Year Born** fields (leaving only the **Author ID** label and text box). Resize the form so it is much taller (tall enough to hold ten labels and text boxes). Move the command buttons to the bottom of the resized form. Don't worry where things are right now – they can always be resized and/or moved later. Copy and paste the label control until you have ten elements (**Index 0 to 9**) of a **Label1** control array. Set the following **Caption** properties:

Label1 Index	Caption
0	Publisher ID
1	Name
2	Company Name
3	Address
4	City
5	State
6	Zip
7	Telephone
8	Fax
9	Comments

Do any repositioning or resizing necessary.

3. At this point, my modified form looks like this:

The screenshot shows a Visual Basic form titled "Authors". It features a grid of labels and text boxes for data entry. The labels are: "Publisher ID", "Name", "Company Name", "Address", "City", "State", "Zip", "Telephone", "FAX", and "Comments". The "Publisher ID" text box is labeled "Text1" and is connected to a data grid control labeled "Data1". At the bottom of the form, there are several command buttons: "<= Previous", "Next =>", "Edit", "Save", "Cancel", "Add New", "Delete", "Done", and "Help".

Change these properties:

frmAuthors (current name):

Name	frmPublishers
Caption	Publishers

datAuthors (current name):

Name	datPublishers
RecordSource	SELECT * FROM Publishers ORDER BY Name

ADO Data Environment Modifications

Add a new **command** object to **conBooks** connection object. Assign these properties:

Command1:

Name	comPublishers
ConnectionName	conBooks
CommandType	1-adCmdText
CommandText	SELECT * FROM Publishers ORDER BY Name
LockType	2-adLockOptimistic

4. In the **cmdPrevious_Click** and **cmdNext_Click** events, replace all occurrences of **datAuthors** with **datPublishers**.

ADO Data Environment Modification

Replace all occurrences of **rscomAuthors** with **rscomPublishers**.

5. Now, we'll create the text box control array for each field. Set these properties for the one text box on the form:

txtAuthorID (current name):

Name	txtInput
DataSource	datPublishers
DataField	PubID
TabStop	True

ADO Data Environment Modifications

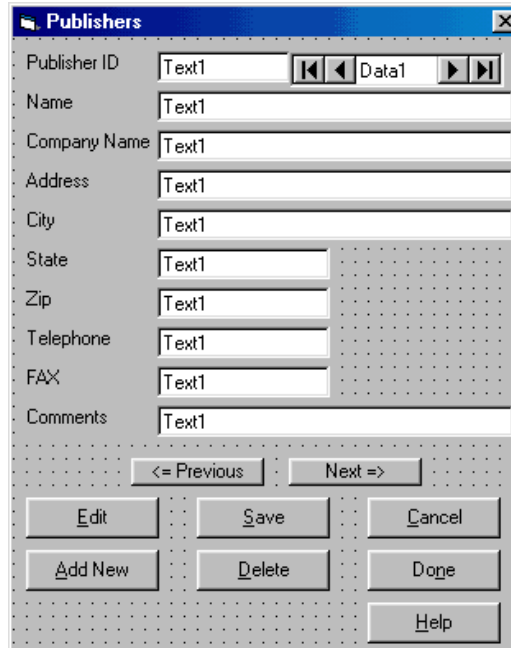
txtAuthorID:

Name	txtInput
DataSource	denBooks
DataMember	comPublishers
DataField	PubID
TabStop	True

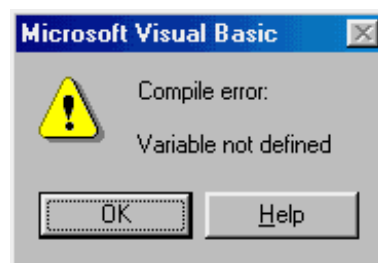
Copy and paste this control until you have ten elements in the **txtInput** control array. Set the **TabIndex** and **DataField** properties for these controls:

txtInput	Index	TabIndex	DataField
	1	1	Name
	2	2	Company Name
	3	3	Address
	4	4	City
	5	5	State
	6	6	Zip
	7	7	Telephone
	8	8	Fax
	9	9	Comments

We now have ten text boxes bound to the fields of the Publishers database table, completing the interface appearance:

A screenshot of a Visual Basic form titled "Publishers". The form contains several text boxes for data entry: "Publisher ID", "Name", "Company Name", "Address", "City", "State", "Zip", "Telephone", "FAX", and "Comments". Each text box is followed by a "Text1" label. Below the text boxes are navigation buttons: "<= Previous", "Next =>", "Edit", "Save", "Cancel", "Add New", "Delete", "Done", and "Help".

6. Save the application. Try running it – you'll see this:



Click **OK** and you will be told that the variable **txtAuthor** (or perhaps some other variable) is not recognized. And, of course, it's not. We just deleted it. This is what we meant by the need to remove vestiges (old code) from the application we are modifying. Trying to run the application will many times point out these "vestiges" to us. We'll begin eliminating old code (as well as adding new code) in the next section.

Add Message Box(es)

In its current state, all the message boxes within our code, except one, are generic in nature. These generic message boxes can be left as is. The one exception is the message box we added to inform the user if they typed an invalid date for the old **Year Born** field. This message box will be deleted in the next step.

Code Application State

1. In this step, we modify the code to reflect proper application state. We will eliminate all old code, so when we are done the application will run without errors. The biggest changes are in the **SetState** procedure (this is the procedure that gave us the error message seen earlier). The modification locks and unlocks the text boxes, depending on state. The procedure is (new code is italicized):

```
Private Sub SetState(AppState As String)
    Dim I As Integer
    Select Case AppState
    Case "View"
        txtInput(0).BackColor = vbWhite
        For I = 1 To 9
            txtInput(I).Locked = True
        Next I
        cmdPrevious.Enabled = True
        cmdNext.Enabled = True
        cmdAddNew.Enabled = True
        cmdSave.Enabled = False
        cmdCancel.Enabled = False
        cmdEdit.Enabled = True
        cmdDelete.Enabled = True
        cmdDone.Enabled = True
        txtInput(1).SetFocus
    Case "Add", "Edit"
        txtInput(0).BackColor = vbRed
        For I = 1 To 9
            txtInput(I).Locked = False
        Next I
        cmdPrevious.Enabled = False
        cmdNext.Enabled = False
        cmdAddNew.Enabled = False
        cmdSave.Enabled = True
        cmdCancel.Enabled = True
        cmdEdit.Enabled = False
        cmdDelete.Enabled = False
        cmdDone.Enabled = False
        txtInput(1).SetFocus
    End Select
End Sub
```

2. Add this code to the **txtInput_KeyPress** event. This implements the code to programmatically move from text box to text box using the <Enter> key (as an alternate to using <Tab>):

```
Private Sub txtInput_KeyPress(Index As Integer, KeyAscii  
As Integer)  
If KeyAscii = vbKeyReturn Then  
    If Index <> 9 Then  
        txtInput(Index + 1).SetFocus  
    Else  
        txtInput(1).SetFocus  
    End If  
End If  
End Sub
```

3. Save and run the application. You should now be able to move from record to record and use the other command buttons to switch from state to state (don't click **Save** or **Help** yet).

Perform Entry Validation

We need to eliminate any old entry validations done and add required new ones. The only field that appears to need entry validation is **Zip** (it only uses numbers and hyphens, for 9 digit zips). We won't add any validation, though. Why? Perhaps, in the future, the post office will develop a zip code with letters. We want to be ready for this possibility. And, other countries have a wide variety of zip formats. Since we are doing nothing but displaying this value, validation is not that important. If we were doing math with a value or using it in some other function, validation would take on greater importance.

The old validation we need to eliminate is in the **KeyPress** event procedure for the **txtYearBorn** control. In fact, we can eliminate this entire procedure since the control no longer exists. Do this - note it is listed as a general procedure since there is no associated control. Also eliminate the **txtAuthor_KeyPress** event procedure (a general procedure).

Perform Input Validation

Again, we need to eliminate any old input validations done and add required new ones. All of the inputs here are generic in nature and don't need much validation. We will just insure a publisher **Name** field is entered.

1. Modify the **ValidateData** procedure to read (just eliminate the **Year Born** validation and modify the **Author Name** validation a bit):

```
Private Sub ValidateData(AllOK As Boolean)
Dim Message As String
AllOK = True
'Check for name
If Len(txtInput(1).Text) = 0 Then
    Message = "You must enter a Publisher Name." & vbCrLf &
Message
    txtInput(1).SetFocus
    AllOK = False
End If
If Not (AllOK) Then
    MsgBox Message, vbOKOnly + vbInformation, "Validation
Error"
End If
End Sub
```

You may be asking – isn't the **PubID** field important enough to be validated? Well, yes, but being a primary key, it is treated differently. We will see how to handle this in Chapter 7.

2. Save the application and run it. Click Edit. Blank out the Publisher Name field and click Save. A message box should appear. Stop the application.

Add Error Trapping and Handling

The error trapping and handling code in the old application still applies to the new application, hence no change is needed here. This is often the case in modifying existing applications. For other applications, you may need to modify existing error trapping schemes or add new ones.

Add On-Line Help System

Use the Help Workshop to develop a help system named **Publishers.hlp**.

1. In your word processor, prepare a single topic file (**Publishers.rtf**). The topic I used is:

#Publishers Input Form

Available options for managing Publishers database table:

Add New Record

Click the **Add New** button to add a record to the Publishers database. Type in the requested fields. The Publisher Name is a required field. Click **Save** to save the new record; click **Cancel** to ignore the new record.

Edit Record

Click the **Edit** button to edit the displayed record. Make any needed changes. The Publisher Name is a required field. Click **Save** to save the changes; click **Cancel** to ignore the changes.

Delete Record

Click the **Delete** button to delete the displayed record.

Exit Program

Click the **Done** button to quit the application.

#Topic1

Establish a topic ID (we used **Topic1**) footnote for the one topic.

2. In the Help Workshop, prepare a contents file (**Publishers.cnt**) and a project file (**Publishers.hpj**). For each file, you might like to modify the corresponding Authors files and resave them with new names. Create the help file (**Publishers.hlp**).

3. Go back to your application in Visual Basic. Click **Project**, then **Properties**. Set the **Help File** name. Change the code in the **cmdHelp_Click** procedure to read:

```
Private Sub cmdHelp_Click()  
Dim Rtn As Long  
Rtn = WinHelp(frmPublishers.hwnd, App.HelpFile,  
HELP_FINDER, CLng(0))  
End Sub
```

4. Save the application. Run it. Make sure both the <F1> key and Help button bring up the help system properly.

Application Testing

If you did all the above steps carefully, the application should be running properly. If not, make the changes required to get in running. As with the Authors form, we still need code to add the database management functions. This is addressed in the next chapter.

This page intentionally not left blank.