T U T O R I A L

File Input/Output (I/O)

An important aspect of any programming language is its ability to access and manipulate files. Visual Basic provides all the statements and commands necessary for storing information to and reading it from files. I didn't discuss this topic in the book because Visual Basic's commands for file input/output (file I/O) are not new; they have existed in BASIC for many years. Moreover, Visual Basic makes it easy to store information in databases and access it with the Data Access Objects (I discuss DAO in Chapter 17). In some cases, however, you want to be able to store a few lines of text to a file or read a few numbers from a binary file. Setting up an entire database is an overkill in these situations. It's much simpler to create a file, store the information there, and recall it as needed.

File Types

Visual Basic supports three types of files:

- Sequential
- Random access
- Binary

Sequential files are mostly text files (the ones you can open with a text editor such as Notepad). These files store information as it's entered, one byte per character. Even the numbers in a sequential file are stored as a string and not as numeric values (that is, the numeric value 33.4 is not stored as a single or double value, but as the string "33.4"). These files are commonly created by text-processing applications and are used for storing mostly text, not numbers.

Sequential files are read from the beginning to the end. Therefore, you can't read and write at the same time to a sequential file. If you must read from and write to the file simultaneously, you must open two sequential files: one for reading from and another one for writing to.

If your application requires frequent access to the file's data (as opposed to reading all the data into memory and saving them back when it's done), you should use *random access files*. Like sequential files, random access files store text as characters, one byte per character. Numbers, however, are stored in their native format (as integers, doubles, and so on). You can display a random access file in a DOS window with the TYPE command and see the text, but you won't be able to read the numbers. Random access files are used for storing data that are organized in segments of equal length. These segments are called *records*. Random access files allow

you to move to any record, as long as you know where the record is located. Since all records are of the same length, it's easy to locate any record in the file by its index. Moreover, unlike sequential files, random access files can be opened for reading and writing at the same time. If you decide to change a specific record, you can write the new record's data on top of the old record, without affecting the adjacent records.

Binary files, finally, are similar to sequential files, and they make no assumption as to the type of data stored in them. The bytes of a binary file can be characters or the contents of an executable file. Images, for instance, are stored in binary files.

TIP

If your application deals with text files only, you should also consider the TextStream object of VBScript, which is described in Chapter 20 of *Mastering Visual Basic 6*. Using this object you can also access text files from within Windows scripts.

File Manipulation Commands

The various file types differ not as much in how they store data, but in how we access them. A file that was created as a random access file can also be opened as a binary file. Of course, how data are encoded is the programmer's responsibility, but it's possible to create and open a file in different modes. We'll look at the three file types shortly along with examples and the commands required to access their contents. But let's start with the basic commands. How you manipulate a file is more or less independent of its type and involves three stages:

Opening the file: The operating system reserves some memory for storing the file's data. If the file does not exist, it's first created and then opened. To open a file (and create it if necessary), use the Open command.

Processing the file: A file can be opened for reading from, writing to, or reading and writing. Data are read, processed, and then stored back to the same or to another file.

Closing the file: When the file is closed, the operating system releases the memory reserved for the file. To close an open file, use the Close command.

In the following sections, we'll look at Visual Basic's file-manipulation statements and functions.

The Open Statement

To use a file, you must first open it—or create it, if it doesn't already exist. The Open statement, which opens files, accepts a number of arguments:

```
Open file_name, For file_type, Access access_type, lock_type,_
      As #file_number, Len=record_length
```

The *For*, *Access*, *As,* and *Len* arguments are Visual Basic keywords. Most arguments are optional.

The simplest form of the Open statement is:

```
Open fileName As #1
```

This line opens a file and assigns it the number 1. Subsequent statements use the numeric value 1 to identify this file.

The argument *file_name* is the name of the file (the name of a disk file). The argument *file_type* determines the file's type and can be one of the following constants:

- Input
- Output
- Append
- Random
- Binary

The first three refer to sequential files; Random is used with random access files, and Binary is used with binary files. When you open a sequential file, you can't change its data. You can either read them (and store them to another file) or overwrite the entire file with the new data. To do so, you must open the file for Input, read its data, and then close the file. To overwrite it, open it again (this time for Output) and save the new data to it.

If you don't want to overwrite an existing file, but append data to it (without changing any of the existing data), open it for Append.

WARNING

If you open a file for Output, Visual Basic wipes out its contents, even if you don't write anything to it. Moreover, VB won't warn you that it's about to overwrite a file, as applications do. This is how the Open statement works, and you can't change your mind after opening a sequential file for Output.

The *access_type* argument is used with random access files and determines whether the file can be opened for reading from (Read), writing to (Write), or

both (Read Write). If you open a file with Read access, your program can't modify it even by mistake. The access method has nothing to do with file types. Sequential files are open for Input or Output only, because they can't be opened in both modes. The access type is specified for reasons of safety. If you need to open a file only to read data from it, open it with Read access (there's no reason to risk modifying the data).

The *lock_type* argument allows you to specify the rights of other Windows applications, while your application keeps the file open. Under Windows, many applications can be running at the same time, and one of them may attempt to open a file that is already open. In this case, you can specify how other applications are to access the file. The *lock_type* argument can have one of the following values:

Shared Other applications can share the file.

Lock Read The file is locked for reading.

Write Lock The file is locked for writing.

Lock Read Write Other applications can't access this file.

File locking is a very important function, especially in a networked environment. Imagine two users attempting to write to the same file at the same time. Using the file-locking features you can write programs that work properly in networked environments. However, if you are going to build applications that will be run by many users who access the same files, you should consider building a database.

After the As keyword follows a numeric value that uniquely identifies the file. Every file you open must have its own, unique number. This number is used by subsequent commands to identify the specific file. The Close command, for example, which closes an open file when it's no longer needed, must know which file to close. The statement:

```
Close #1
```

closes the file that was opened as #1. The file number has nothing to do with the actual file on the disk. The same file can be opened later with another number. Without the file number, you'd have to specify the file's name with each command that accesses the file. The file's number is therefore a shorthand notation for identifying files from within our code.

Finally, if the file is a random access one, you must declare the length of the record with the Len keyword. The *record_length* argument is the record's length in bytes. When you create a random access file, Visual Basic doesn't record any information regarding the record's length or structure to the file. You should know, therefore, the structure of each record in a random access file before you can open it. The record's

length is the sum of the bytes taken by all record fields. You can either calculate it, or you can use the function Len (*record*) to let Visual Basic calculate it. The *record* argument is the name of the structure you use with the random access file.

The following command opens the file c:\samples\vb\cust.dat as a sequential file with the number 1:

```
Open "c:\samples\vb\cust.dat" For Input As #1
```

To open a random access file, you must know its record's length. The record is the basic element of a random access file, and the record is the smallest piece of information you can write to a random access file. To find the length of the record, you must first decide how the data will be organized in fields and declare the record's type. Let's say you want to create a random access file for storing music records. You can use the following structure for the purposes of this application:

```
Type Mrecord
    Title As String*60
    Group As String*30
    Style As String*3
    Year As Date
End Type
```

This structure is simple, but you can extend it by adding any other field you need. The length of this structure can be easily calculated if you know how many bytes a Date type takes. To avoid mistakes, let Visual Basic calculate the record's length, with the Len() function—the same function that returns the length of a string. The following statement opens a random access file for reading from and writing to, using the record structure shown above:

```
Open "c:\samples\vb\cust.dat" For Random As #1 Len=Len(Mrec)
```

Mrec is a variable declared as Mrecord type.

The FreeFile() Function

During the course of an application, you may open and close many files and you may not always know in advance which file numbers are available. Visual Basic provides the FreeFile() function, which returns the next available file number. The FreeFile() function is used in conjunction with the Open statement to open a file:

```
fNum = FreeFile()
Open "c:\samples\vb\cust.dat" For Random As #fNum Len=Len(Mrec)
```

After these two statements execute, all subsequent commands that operate on the specified file can refer to it as *fNum*. The FreeFile() function returns the next available

file number, and unless this number is assigned to a file, FreeFile()returns the same number if called again. The following statements will not work:

```
fNum1 = FreeFile()
fNum2 = FreeFile()
Open "file1" For Input As #fNum1
Open "file2" For Output As #fNum2
```

Each time you call FreeFile() to get a new file number, you must use it. The previous statements should have been coded as follows:

```
fNum1 = FreeFile()
Open "file1" For Input As #fNum1
fNum2 = FreeFile()
Open "file2" For Output As #fNum2
```

The Close(file_number) Statement

The Close statement closes an open file, whose number is passed as argument. The statement

```
Close #fNum1
```

closes the file opened as #fNum1. You can also call the Close statement with multiple file numbers, as in:

```
Close #fNum1, fNum2, fNum3
```

This statement closes the three files that were opened as #fNum1, #fNum2, and #fNum3.

The EOF(file_number) and LOF(file_number) Functions

These are two more frequently used functions in file manipulation. The EOF() function accepts as an argument the number of an open file and returns True if the end of the file (EOF) has been reached. The LOF() function returns the length of the file, whose number is passed as argument.

You use the EOF() function to determine whether the end of the file has been reached, with a loop such as the following:

```
{get first record}
While Not EOF(fNum)
    {process current record}
    {get next record}
Wend
```

With the help of the LOF() function, you can also calculate the number of records in a random access file:

```
Rec_Length = LOF(file_number) / Len(record)
```

In the following sections, we are going to look at the commands for writing to and reading from all types of files. These commands are not the same for all types of files. Some commands read an entire line of text, others read a number of characters (or bytes), and others read and write records to a random access file. Let's start with the statements for sequential files:

The Print #file_num, variable, variable, ... Statement

The Print statement writes data to a sequential file. The first argument is the number of the file to be written, and the following arguments (you can supply any number of arguments after the first one) are the variables to be written to the file. After all variable values have been written to the file, the Print # statement inserts a line break. The following statements write two lines of text to the file opened as #fNum and insert a line break between them:

```
Print #fNum, "this is the first line of text"
Print #fNum, "and this is the second line of text"
```

The semicolon (;) and the comma (,) characters determine the *screen position*, where the pointer will be moved before printing the next value. The semicolon specifies that the first character of the new value will be placed right after the last character of the last value. The comma specifies that the next character will be printed in the next *print zone*. Each print zone corresponds to 14 columns. In other words, the Print # statement writes data to the file exactly as the TYPE command (of DOS) displays them on the screen. You must keep in mind that the text will be displayed correctly only when printed with a monospaced typeface, such as Courier. If you place the text on a TextBox with a proportional typeface, the columns will not align.

The Line Input #file_number, strVar Statement

To read from sequential files, use the Line Input # statement. The *file_number* argument is the file's number, and *strVar* is the name of a variable where the data read from the file will be stored. The Line Input statement reads a single line of text from the file and assigns it to a string variable. This statement reads all the characters from the beginning of the file to the first newline character. When you call it again, it returns the following characters, up to the next newline character. The newline characters are not part of the information stored to or read from the file, and they are used only as delimiters. If we close the file of the last example

and open it again, the following lines will read the first two text lines and assign them to the string variables *line1* and *line2:*

```
Line Input #fNum, line1
Line Input #fNum, line2
```

If you want to store plain text to a disk file, create a sequential file and store the text there, one line at a time. To read it back, open the file and read one line at a time with the Lin Input statement, or use the following function to read the entire text.

The Input$(nchars, #file_number) Statement

The Input$() function reads *nchars* characters from the file specified by the second argument. This function is used to read the entire file and is faster then reading the file one line at a time. To read the entire text file, set the number of characters equal to the file's length, using the LOF() function:

```
Text1.Text = Input$(LOF(fNum), fNum)
```

In addition, this function offers more flexibility than the others, since it allows you to read any number of characters, regardless of line breaks.

The Width #fNum, length Statement

This is another useful statement that applies to sequential files only. The Width statement sets the maximum line length that can be written to a file. The maximum line length is specified by the second argument, *length*. A line with fewer characters than *length* is stored to the file as is. Longer lines are broken; Visual Basic automatically inserts newline characters to enforce the specified maximum line length.

The Put and Get Statements

These statements are used for writing records to and reading records from a random access file. Both commands must know the record number you want to access (write or read). The syntax of the Put statement is:

```
Put #file_number, record_number, record
```

The *record_number* argument is the number of the record we are interested in, and *record* is a record variable that is written to the file. The *record_number* argument is optional; if you omit it, the record will be written to the current record position. After a record is written to or read from the file, the next record becomes the current one. If you've read the second record, the Put statement will store the field values in the third record in the file. If you call the Put statement 10 times sequentially

without specifying a record number, it will create (or overwrite) the first 10 records of the random access file.

The syntax of the Get statement is quite similar:

```
Get #file_number, record_number, record
```

and its arguments have the same meaning.

At this point, I'll outline the basics of random access file manipulation, since this is the most flexible file type. Let's say you want to create a random access file for storing a product list. Each product's information is stored in a *ProductRecord* variable, whose declaration is shown next:

```
Type ProductRecord
    ProductID As String*10
    Description As String*100
    Price As Currency
End Type
```

The Type *ProductRecord* will be used for storing each product's information before moving it to the file. Let's start by defining a variable of type *ProductRecord*:

```
Dim PRec As ProductRecord
```

You can then assign values to the fields of the *PRec* variable with statements such as the following:

```
PRec.ProductID = "TV00180-A"
PRec.Description = "SONY Trinitron TV"
PRec.Price = 799.99
```

The *PRec* record variable can be stored to a random access file with the Put statement. Of course, you must first create the file with the following statements:

```
FNum = FreeFile()
Open "c:\products.dat" For Random Len=Len(ProductRecord) As #FNum
```

You can then write the Prec variable to the file with the statement:

```
Put #FNum, , PRec
```

Notice that you can omit the number of the record where the data will be stored. You can change the values of the fields and keep storing additional records with the same Put statement (as long as PRec is populated with a different value). After all the values are stored to the file, you can close the file with this statement:

```
Close #FNum
```

To read the records, open the file with the same Open statement you used to open it for saving the records:

```
FNum = FreeFile()
Open "c:\products.dat" For Random Len=Len(ProductRecord) As #FNum
```

You can then set up a loop to read the records in an array of PRec structures. Assuming that the array Products has been declared with the statement:

```
Dim Products(100) As PRec
```

you can scan the records of the file with the following loop:

```
TotRecords = LOF(fNum) / Len(ProductRecord)
For i = 1 to TotRecords
    Get #FNum, , Products(i)
Next
```

The function LOF() returns the length of the file (in bytes). By dividing this number by the length of each record (also in bytes), we get the number of records in the file. This value is used by the For…Next loop to scan all records in the file.

The Seek Statement and the LOC() Function

An important concept in processing random access files is that of the *current record*. Visual Basic maintains a pointer to the current record for each open random access file. Each time you read a record or save one in the file, the pointer is increased by one (unless it's the last record in the file) to point to the next record. The numbers of the records are not stored on disk. Since all records have the same length, the operating system can calculate where each record begins and move to the corresponding byte instantly.

Visual Basic provides the Seek statement, which lets you move to any record in the random access file, and the Loc() function, which returns the number of the current record. In other words, the Seek statement manipulates the record pointer, and the Loc() function simply reads its value.

The syntax of the Seek statement is:

```
Seek #file_number, record_number
```

in which *record_number* is the number of the record to which you want to move. The Seek statement can't locate a record based on its contents. It's like accessing an array element with its index.

The syntax of the Loc() function is:

```
Loc(file_number)
```

To move to the next record in a random access file, use the following statements:

```
fPointer = Loc(fNum) + 1
Seek(fNum, fPointer)
```

The value returned by the Loc() function is a Long Integer, which means a random access file can hold too many records for the average disk.

The Seek statement and Loc() function can be used with random access files, as well as binary files, only in this case there are no record numbers. We usually set the length of the record to one byte, and we seek for a specific byte number in the file. If you want to read a number of bytes starting at a specified location in the file, use the Seek method to move to the first byte you're interested in and then read as many bytes as you need.

The Lock and Unlock Statements

The Lock statement allows you to lock a file or some of the records in a random access file. The locked records are not available to other applications that are currently running. Your application, however, has access to all files. If another application attempts to open a locked file or to access one of the locked records, Visual Basic will generate a trappable runtime error.

With the Lock command you can lock an entire file. The statement

```
Lock #fNum
```

locks the entire file opened as #fNum. If the file was opened as a sequential file, the entire file must be locked. If the file was opened as a random access file, you can lock one or more records only. The following statement locks the records 99 through 110 of the file opened as #fNum:

```
Lock #fNum 99,100
```

If the file was opened as binary, the bytes 99 through 100 will become unavailable to other applications.

The Unlock statement has the same syntax, but the opposite effect. If the file is locked, it unlocks it. The Lock and Unlock statements are essential in building applications that access files in an operating system such as Windows, in which multiple applications can access the same file and act on it. You should always take into consideration the possibility of another application accessing the same files as yours. However, if the situation gets too complicated, you should probably switch to a database.

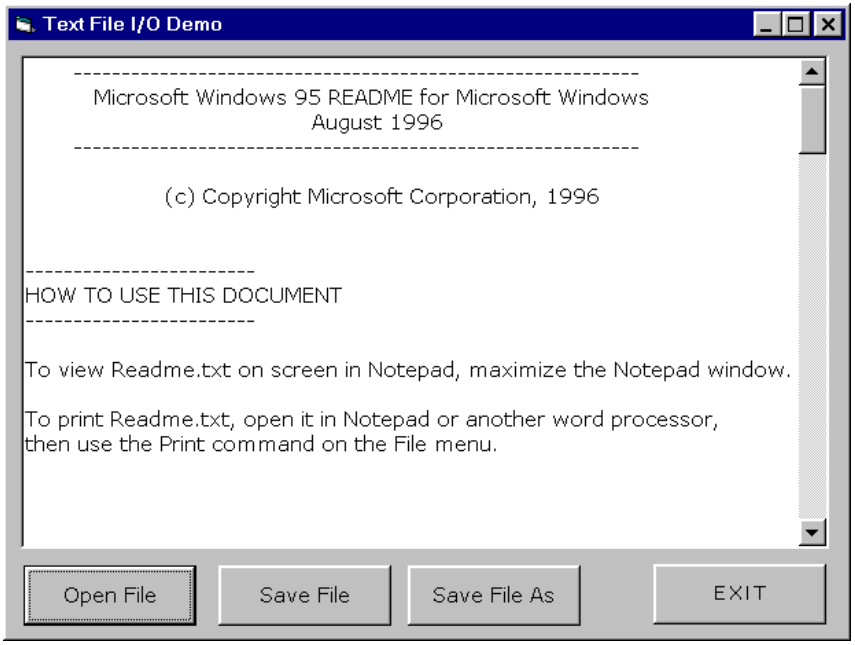
In the following sections, we are going to look at a few examples that put together the information presented so far.

The Textfile Project

The Textfile application, shown in Figure 1, allows you to enter some text in a TextBox control and save it to a text file. The TextBox control can also be loaded with a text file's contents. This project is an adaptation of the TextPad project we developed in Chapter 5, to demonstrate the TextBox control.

FIGURE 1:

The Textfile application demonstrates how to access text files.



The code that stores the contents of the TextBox to a file is quite short:

```
FNum = FreeFile
Open OpenFile For Output As #1
Print #FNum, Text1.Text
Close #FNum
```

The name of the file where the text will be read from, *OpenFile*, is specified by the user with the help of a File Open common dialog box. The code saves the entire text with a single statement and then closes the file.

The lines that load a text file onto the TextBox control are shown next:

```
FNum = FreeFile
Open CommonDialog1.FileName For Input As #1
txt = Input(LOF(FNum), #FNum)
Close #FNum
Text1.Text = txt
```

The code reads the entire file into the *txt* string variable and then assigns it to the Text property of the TextBox control. You can open the project with Visual Basic and see the entire listing.

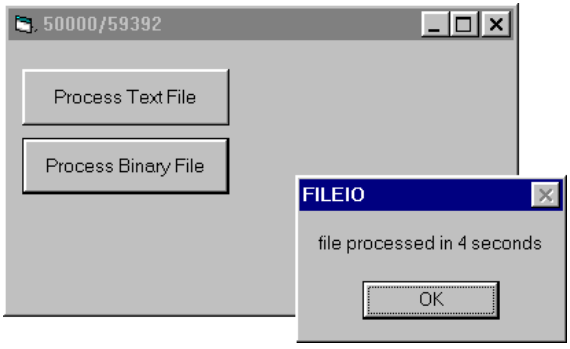
The FileIO Project

The FileIO project, shown in Figure 2, demonstrates how to access text files one line at a time and binary files one character at a time. Processing the file consists of turning text into uppercase and simply copying binary files. You can adjust the Process procedure to perform any type of processing you need.

The sample application prompts the user to select the file to be processed and then to select the output file. You will see the File Open dialog box, followed by the File Save dialog box. This isn't the best user interface, but we'll focus on the file input/output statements and functions, not the application's interface. After you specify the file to be processed and the output file (where the processed data will be written), the program starts processing the file.

FIGURE 2:

The FileIO application demonstrates how to access a text file one line at a time and binary files one byte at a time.



The code that reads the text file and processes it is shown next. The Line Input routine and saved to the output file. Here's the code:

```
Open outfile For Output Access Write As #outfileNumber
While Not EOF(infileNumber)
    Line Input #infileNumber, tline
```

```
    outline = Process(tline)
    Print #outfileNumber, outline
Wend
Close infileNumber
Close outfileNumber
```

Text files are processed very fast, partly because they are read one line at a time, but mostly because they are short, compared with typical binary files, such as images or executable files. If you click the Process Binary File button, however, and select a large file, the processing will most likely take a while. The program displays the progress of the operation in the Form's title bar, as the number of bytes processed versus the total number of bytes in the file. This indicator is updated every 10,000 bytes.

The code that processes the binary file is shown next. It uses the Get statement to retrieve the next byte from the file, processes it (it simply assigns it to another variable), and stores it to the output file. Notice how the code uses the LOF() function to keep track of the percentage of the file already processed.

```
i = 0
FLen = LOF(infileNumber)
While Not EOF(infileNumber)
    Get #infileNumber, , inChar
    outChar = inChar
    Put #outfileNumber, , outChar
    ipos = ipos + 1
    If (ipos Mod 10000) = 0 Then
        Me.Caption = ipos & "/" & FLen
        DoEvents
    End If
Wend
```

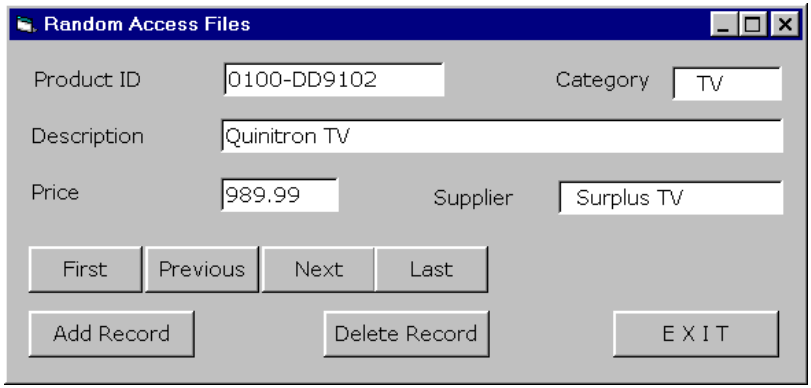
You can open the FileIO project in the Visual Basic IDE and examine the rest of the code (it also appears at the end of this tutorial).

The Products Project

The Products project, shown in Figure 3, demonstrates the basic techniques for manipulating random access files. The Products application maintains a product catalog that is stored in a random access file. The text boxes on the Form in Figure 3 display the fields of the current record. Use the navigation buttons to move to the desired record, and the Add and Delete buttons to add and delete records.

FIGURE 3:

The Products project demonstrates how to access random access files.



Here's the structure of the ProductRecord record:

```
Private Type ProductRecord
    ID As String * 12
    Category As String * 5
    Description As String * 50
    Price As Currency
    Supplier As String * 20
    Deleted As Boolean
End Type
```

The array is declared as:

```
Dim Products(1000) As ProductRecord
```

The records are stored in a random access file on disk, and they are read into an array when the program starts. When the program ends, the records are moved from the array back to the Products file on disk. To avoid moving records in the array when a record is deleted, I use the Deleted flag in the ProductRecord structure. When a record is deleted, this field is set to True, and the program ignores this record in subsequent operations.

The code that's executed when the Form is loaded is shown next. It opens the Products file in the application's path and then populates the Products array. The *totRecords* and *currentRecord* variables are Form variables; all procedures use the *totRecords* variable to keep track of the maximum number of records and the *currentRecord* variable to keep track of the current record's index.

```
Private Sub Form_Load()
    fnum = FreeFile
    Open App.Path & "/products" For Random As #fnum Len = Len(Products(0))
    totRecords = LOF(fnum) / Len(Products(0))
    If totRecords = 0 Then
        currentRecord = 0
```

```
            Close #fnum
            Exit Sub
        End If
        For i = 1 To totRecords
            Get #fnum, i, Products(i)
        Next
        Close #fnum
        If totRecords > 0 Then
            currentRecord = 1
            ShowRecord
        End If
    End Sub
```

The code behind the navigation buttons is simple. It manipulates the variable *currentRecord* and then displays the corresponding element of the array. Here's the code behind all four of the navigation buttons:

```
Private Sub cmdFirst_Click()
    If currentRecord = 0 Then
        MsgBox "There are no records in the Products array"
    Else
        currentRecord = 1
        ShowRecord
    End If
End Sub

Private Sub cmdLast_Click()
    If currentRecord = 0 Then
        MsgBox "There are no records in the Products array"
    Else
        currentRecord = totRecords
        ShowRecord
    End If
End Sub

Private Sub cmdPrevious_Click()
Dim oldRecord As Integer
    oldRecord = currentRecord
    currentRecord = currentRecord - 1
    While currentRecord > 0 And Products(currentRecord).Deleted
        currentRecord = currentRecord - 1
    Wend
    If currentRecord <= 0 Then
        MsgBox "This is the first record in the array"
        currentRecord = oldRecord
    Else
        ShowRecord
```

```
    End If
End Sub

Private Sub cmdNext_Click()
Dim oldRecord As Integer
    oldRecord = currentRecord
    currentRecord = currentRecord + 1
    While currentRecord <= totRecords And Products(current-
Record).Deleted
        currentRecord = currentRecord + 1
    Wend
    If currentRecord > totRecords Then
        MsgBox "This is the last record in the array"
        currentRecord = oldRecord
    Else
        ShowRecord
    End If
End Sub
```

When the Delete Record button is clicked, the program sets the current record's Deleted field to True. It doesn't remove the element from the Products array (you'd have to move all following records one position up in the array), and it doesn't even display the next or previous record, as is customary in similar applications. You might want to implement this feature yourself. You may actually not want to ignore the deleted records, but rather display them in red or with a special mark somewhere on the Form and offer the user an Undelete button. This button must normally be disabled, and you should enabled it from within your code only if the current record's Deleted field is True.

The deleted records are actually removed from the disk file when the records are saved. This action takes place from within the Exit button's Click event handler. As you recall from the discussion of the commands for accessing random access files, there are no statements for removing records. You can only add records to a random access file; therefore, its size can't decrease. To actually get rid of the deleted records, you must read the records into memory, delete the file, and then create a new file with the same structure and name and save the records there. If the file is too long, you can open two random access files at the same time, read the records from the original file, and save the ones that haven't been marked for deletion to the other file. (You may come up with more complicated algorithms, that save new records in the place of deleted records, but when things get this complicated, it's time to switch to a database application.) Here's the code that saves the active records of the Products file:

```
Private Sub cmdExit_Click()
    Kill App.Path & "/products"
    fnum = FreeFile
```

```
    Open App.Path & "/products" For Random As #fnum Len = Len(Prod-
ucts(0))
    For i = 1 To totRecords
        If Not Products(i).Deleted Then
            Put #fnum, , Products(i)
        End If
    Next
    Close #fnum
    End
End Sub
```

The code of the Exit command button deletes the Products file in the application's path and then creates a new random access file with the same name and stores the records of the Products array there. The If statement skips the records that were marked for deletion during execution.

TIP

If power goes off while the program saves the records, you'll lose all the information. In a real application, you should probably make a copy of the original file, and after all records have been successfully saved, delete the copy. When the application starts, it should always look for this temporary file. Normally, it shouldn't exist. If it does, something terrible happened during the last save operation, and you should offer some recovery options to the user.

The Add Record button clears the text boxes in preparation for the entry of a new record, by calling the ClearFields subroutine. It then hides the buttons you see on the Form and makes two other buttons visible: the OK and Cancel buttons. This action is performed by the HideButtons subroutine.

After the user has entered the fields of a new record, he or she must click the OK button to commit the new record. The OK button's code is shown next:

```
Private Sub cmdOK_Click()
    If Not IsNumeric(txtPrice.Text) Then
        MsgBox "Invalid price specified"
        Exit Sub
    End If

    totRecords = totRecords + 1
    If totRecords > 1000 Then
        MsgBox "Maximum record count reached. Can't add more records"
        Exit Sub
    End If
    currentRecord = totRecords
    Products(currentRecord).ID = txtID.Text
    Products(currentRecord).Category = txtCategory.Text
    Products(currentRecord).Description = txtDescription.Text
```

```
    Products(currentRecord).Price = (txtPrice.Text)
    Products(currentRecord).Supplier = txtSupplier.Text
    Products(currentRecord).Deleted = False
    ShowButtons
End Sub
```

This code makes sure that the Price field has a valid value (you can add more data validation here, of course) and then stores the values of the text boxes to the next available element of the Products array. Finally, it hides the OK and Cancel buttons and makes the navigation and the Add and Delete buttons visible again.

The Products project is a simple example that demonstrates the basic commands for accessing and handling random access files. You should try to modify the application so that instead of reading all the records in memory, it seeks the corresponding record in the file and reads it directly from the disk. Here is the complete listing of the Products project:

```
Private Type ProductRecord
    ID As String * 12
    Category As String * 5
    Description As String * 50
    Price As Currency
    Supplier As String * 20
    Deleted As Boolean
End Type

Dim Products(1000) As ProductRecord
Dim currentRecord As Integer
Dim totRecords As Integer

Private Sub cmdAdd_Click()
    ClearFields
    HideButtons
End Sub

Private Sub cmdCancel_Click()
    ShowRecord
    ShowButtons
End Sub

Private Sub cmdDelete_Click()
    Products(currentRecord).Deleted = True
End Sub

Private Sub cmdExit_Click()
    Kill App.Path & "/products"
    fnum = FreeFile
```

```
    Open App.Path & "/products" For Random As #fnum Len = Len(Products(0))
    For i = 1 To totRecords
        If Not Products(i).Deleted Then
            Put #fnum, , Products(i)
        End If
    Next
    Close #fnum
    End
End Sub

Private Sub cmdFirst_Click()
    If currentRecord = 0 Then
        MsgBox "There are no records in the Products array"
    Else
        currentRecord = 1
        ShowRecord
    End If
End Sub

Private Sub cmdLast_Click()
    If currentRecord = 0 Then
        MsgBox "There are no records in the Products array"
    Else
        currentRecord = totRecords
        ShowRecord
    End If
End Sub

Private Sub cmdNext_Click()
Dim oldRecord As Integer
    oldRecord = currentRecord
    currentRecord = currentRecord + 1
    While currentRecord <= totRecords And Products(current-
Record).Deleted
        currentRecord = currentRecord + 1
    Wend
    If currentRecord > totRecords Then
        MsgBox "This is the last record in the array"
        currentRecord = oldRecord
    Else
        ShowRecord
    End If
End Sub

Private Sub cmdOK_Click()
    If Not IsNumeric(txtPrice.Text) Then
```

```
        MsgBox "Invalid price specified"
        Exit Sub
    End If

    totRecords = totRecords + 1
    If totRecords > 1000 Then
        MsgBox "Maximum record count reached. Can't add more records"
        Exit Sub
    End If
    currentRecord = totRecords
    Products(currentRecord).ID = txtID.Text
    Products(currentRecord).Category = txtCategory.Text
    Products(currentRecord).Description = txtDescription.Text
    Products(currentRecord).Price = (txtPrice.Text)
    Products(currentRecord).Supplier = txtSupplier.Text
    Products(currentRecord).Deleted = False
    ShowButtons
End Sub

Private Sub cmdPrevious_Click()
Dim oldRecord As Integer
    oldRecord = currentRecord
    currentRecord = currentRecord - 1
    While currentRecord > 0 And Products(currentRecord).Deleted
        currentRecord = currentRecord - 1
    Wend
    If currentRecord <= 0 Then
        MsgBox "This is the first record in the array"
        currentRecord = oldRecord
    Else
        ShowRecord
    End If
End Sub

Private Sub Form_Load()
    fnum = FreeFile
    Open App.Path & "/products" For Random As #fnum Len = Len(Products(0))
    totRecords = LOF(fnum) / Len(Products(0))
    If totRecords = 0 Then
        currentRecord = 0
        Close #fnum
        Exit Sub
    End If
    For i = 1 To totRecords
        Get #fnum, i, Products(i)
    Next
```

```
    Close #fnum
    If totRecords > 0 Then
        currentRecord = 1
        ShowRecord
    End If
End Sub

Sub ShowRecord()
    txtID.Text = Products(currentRecord).ID
    txtCategory.Text = Products(currentRecord).Category
    txtDescription.Text = Products(currentRecord).Description
    txtPrice.Text = Products(currentRecord).Price
    txtSupplier.Text = Products(currentRecord).Supplier
End Sub

Sub ClearFields()
    txtID.Text = ""
    txtCategory.Text = ""
    txtDescription.Text = ""
    txtSupplier.Text = ""
    txtPrice.Text = ""
End Sub

Sub HideButtons()
    cmdAdd.Visible = False
    cmdDelete.Visible = False
    cmdFirst.Visible = False
    cmdLast.Visible = False
    cmdPrevious.Visible = False
    cmdNext.Visible = False
    cmdExit.Visible = False
    cmdOK.Visible = True
    cmdCancel.Visible = True
End Sub

Sub ShowButtons()
    cmdAdd.Visible = True
    cmdDelete.Visible = True
    cmdFirst.Visible = True
    cmdLast.Visible = True
    cmdPrevious.Visible = True
    cmdNext.Visible = True
    cmdExit.Visible = True
    cmdOK.Visible = False
    cmdCancel.Visible = False
End Sub
```