CHAPTER

12

Working with Files

IN THIS CHAPTER

Working with files, directories, and drives is a common programming task that you'll undoubtedly have to do at some time. This chapter illustrates how to work with the different file types: text files, typed files, and untyped files. The chapter covers how to use a `TFileStream` to encapsulate file I/O and how to take advantage of one of Win32's nicest features: memory-mapped files. You'll create a `TMemoryMappedFile` class that you can use, which encapsulates some of the memory-mapped functionality, and you'll learn how to use this class to perform text searches in text files. This chapter also demonstrates some useful routines to determine available drives, walk directory trees to search for files, and obtain version information on files. By the end of this chapter, you'll have a strong feel for working with files, directories, and drives.

Dealing with File I/O

You will probably need to deal with three types of files. These file types are text files, typed files, and binary files. The next few sections cover file I/O with these types of files. *Text files* are exactly what the name implies. They contain ASCII text that can be read by any text editor. *Typed files* are files that contain programmer-defined data types. *Binary files* cover just about anything else. This is a general name that covers any file that can contain data in any given format or no format at all.

Working with Text Files

This section shows you how to manipulate text files using the procedures and functions built into Object Pascal's Runtime Library. Before you can do anything with a text file, you must open it. First, you must declare a variable of type `TextFile`:

```
var
  MyTextFile: TextFile;
```

You can now use this variable to refer to a text file.

You need to know about two procedures in order to open the file. The first procedure is `AssignFile()`. `AssignFile()` associates a filename with the file variable:

```
AssignFile(MyTextFile, 'MyTextFile.txt');
```

After you've associated the file variable with a filename, you can open the file. You can open a text file in three ways. First, you can create and open a file using the `Rewrite()` procedure. If you use `Rewrite()` on an existing file, it will be overwritten and a new one will be created with the same name. You can also open a file for read-only access by using the `Reset()` procedure. You can append to an existing file by using the `Append()` procedure.

NOTE

`Reset()` opens typed and untyped files with read-write access.

To close a file after you've opened it, you use the `CloseFile()` procedure. Take a look at the following examples, which illustrate each procedure.

To open for read-only access, use this procedure:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Reset(MyTextFile);
  try
    {manipulate the file }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

To create a new file, do the following:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Rewrite(MyTextFile);
  try
    {manipulate the file }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

To append to an existing file, use this procedure:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Append(MyTextFile);
  try
    {manipulate the file }
  finally
```

```
    CloseFile(MyTextFile);
  end;
end;
```

Listing 12.1 shows how you would use `Rewrite()` to create a file and add five lines of text to it.

LISTING 12.1 Creating a Text File

```
var
  MyTextFile: TextFile;
  S: String;
  i: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Rewrite(MyTextFile);
  try
    for i := 1 to 5 do
    begin
      S := 'This is line # ';
      Writeln(MyTextFile, S, i);
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

This file would now contain the following text:

```
This is line # 1
This is line # 2
This is line # 3
This is line # 4
This is line # 5
```

Listing 12.2 illustrates how you would add five more lines to that same file.

LISTING 12.2 Appending to a Text File

```
var
  MyTextFile: TextFile;
  S: String;
  i: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Append(MyTextFile);
  try
    for i := 6 to 10 do
    begin
      S := 'This is line # ';
      Writeln(MyTextFile, S, i);
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

This file's contents are shown here:

```
This is line # 1
This is line # 2
This is line # 3
This is line # 4
This is line # 5
This is line # 6
This is line # 7
This is line # 8
This is line # 9
This is line # 10
```

Notice that in both listings, you were able to write both a string and an integer to the file. The same is true for all numeric types in Object Pascal. To read from this same text file, you would do as shown in Listing 12.3.

LISTING 12.3 Reading from a Text File

```
var
  MyTextFile: TextFile;
  S: String[15];
  i: integer;
  j: integer;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Reset(MyTextFile);
  try
    while not Eof(MyTextFile) do
    begin
      Readln(MyTextFile, S, j);
      Memo1.Lines.Add(S+IntToStr(j));
    end;
  finally
    CloseFile(MyTextFile);
  end;
end;
```

In Listing 12.3, you'll notice that the string variable `S` is declared as `String[15]`. This was required to prevent reading the entire line from the file into the variable, `S`. Not doing so would have caused an error when attempting to read a value into the integer variable `J`. This illustrates another important feature of text file I/O: You can write columns to text files. You can then read these columns into strings of a specific length. It's important that each column is set to a specific length even though the actual strings stored there might be of a different length. Also, notice the use of the `Eof()` function. This function performs a test to determine whether the file pointer is at the end of the file. If it is, you must break out of the loop because there's no more text to read.

To illustrate reading a columnar-formatted text file, we've created a text file named `USCaps.txt`, which contains a list of U.S. capitals in a columnar arrangement. A portion of this file is shown here:

```
Alabama             Montgomery
Alaska              Juneau
Arizona             Phoenix
Arkansas            Little Rock
California          Sacramento
Colorado            Denver
Connecticut         Hartford
Delaware            Dover
```

The state name column has exactly 20 characters. This way, the capitals line up vertically. We've created a project that reads this file and stores the states into a Paradox table. You'll find this project on the CD as `Capitals.dpr`. Its source is shown in Listing 12.4.

NOTE

Before you can run this demo, you will need to create the BDE alias, DDGData. Otherwise, the program will fail. If you installed the software from this book's CD, this alias has already been created for you.

LISTING 12.4 Source Code for the Capitals Project

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables;
```

```
type

  TMainForm = class(TForm)
    btnReadCapitals: TButton;
    tblCapitals: TTable;
    dsCapitals: TDataSource;
    dbgCapitals: TDBGrid;
    procedure btnReadCapitalsClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnReadCapitalsClick(Sender: TObject);
var
  F: TextFile;
  StateName: String[20];
  CapitalName: String[20];
begin
  tblCapitals.Open;
  // Assign the file to the columnar text file.
  AssignFile(F, 'USCAPS.TXT');
  // Open the file for read access.
  Reset(F);
  try
    while not Eof(F) do
    begin
      { Read a line of the file into the two strings each of whose length
        matches the number of characters that make up the column. }
      Readln(F, StateName, CapitalName);
      // Now store both strings into separate columns in a Paradox table
      tblCapitals.Insert;
      tblCapitals['State_Name']    := StateName;
      tblCapitals['State_Capital'] := CapitalName;
      tblCapitals.Post;
    end;
  finally
    CloseFile(F); // Close the file when finished.
  end;
end;
```

continues

LISTING 12.4 Continued

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Empty the table when project starts.
  tblCapitals.EmptyTable;
end;

end.
```

Although this book hasn't covered Delphi database programming yet, the preceding code is straightforward. The point we're trying to make here is that often, text file processing might serve a very useful purpose. This text file just as well might have been a file containing bank account information downloaded from a bank's online banking service, for example.

Working with Typed Files (Files of Record)

You can store Object Pascal data structures in disk files. You can then read data from these files directly into your data structures. This enables you to use typed files for storing and retrieving information as though the data were records in a table. Files that store Pascal data structures are referred to as *files of record*. To illustrate the use of such files, look at the record structure defined here:

```
TPersonRec = packed record
  FirstName: String[20];
  LastName: String[20];
  MI: String[1];
  BirthDay: TDateTime;
  Age: Integer;
end;
```

NOTE

Records that contain ANSI strings, variants, class instances, interfaces, or dynamic arrays may not be written to a file.

Now suppose you wanted to store one or more such records in a file. You've already seen how you might do this using a text file in the previous section. However, you can also do this using a file of record defined like this:

```
DataFile: File of TPersonRec;
```

To read a single record of the type `TPersonRec`, you would do the following:

```
var
  PersonRec: TPersonRec;
  DataFile: File of TPersonRec;
begin
  AssignFile(DataFile, 'PersonFile.dat');
  Reset(DataFile);
  try
    if not Eof(DataFile) then
      read(DataFile, PersonRec);
  finally
    CloseFile(DataFile);
  end;
end;
```

The following code illustrates how you would append a single record to a file:

```
var
  PersonRec: TPersonRec;
  DataFile: File of TPersonRec;
begin
  AssignFile(DataFile, 'PersonFile.dat');
  Reset(DataFile);
  Seek(DataFile, FileSize(DataFile));
  try
    write(DataFile, PersonRec);
  finally
    CloseFile(DataFile);
  end;
end;
```

Note the use of the `Seek()` procedure to move the file position to the end of the file before writing the record to the file. This function usage is well documented in Delphi's online help, so we won't go into detail on it here.

To illustrate the use of typed files, we've created a small application that stores information on persons in an Object Pascal format. This application allows you to browse, add, and edit these records. We also illustrate the use of a `TFileStream` descendant, which we use to encapsulate the file I/O for such records.

Defining a TFileStream Descendant for Typed File I/O

`TFileStream` is a streaming class that can be used to store items that aren't objects. Record structures don't have methods with which they can store themselves to disk or memory. One solution would be to make the record an object instead. Then, you could attach the storage functionality to that object. Another solution is to use storage functionality of a `TFileStream` to store the records. Listing 12.5 shows a unit that defines a `TPersonRec` record and a `TRecordStream`, a descendant of `TFileStream`, which handles the file I/O for storing and retrieving records.

NOTE

Streaming is a topic that we cover in greater depth in Chapter 22, “Advanced Component Techniques.”

LISTING 12.5 The Source Code for `PersRec.PAS`: `TRecordStream`, a `TFileStream` Descendant

```
unit persrec;

interface
uses Classes, dialogs, sysutils;

type

  // Define the record that will hold the person's information.
  TPersonRec = packed record
    FirstName: String[20];
    LastName: String[20];
    MI: String[1];
    BirthDay: TDateTime;
    Age: Integer;
  end;

  // Create a descendant TFileStream which knows about the TPersonRec

  TRecordStream = class(TFileStream)
  private
    function GetNumRecs: Longint;
    function GetCurRec: Longint;
    procedure SetCurRec(RecNo: Longint);
  protected
    function GetRecSize: Longint; virtual;
  public
    function SeekRec(RecNo: Longint; Origin: Word): Longint;
    function WriteRec(const Rec): Longint;
    function AppendRec(const Rec): Longint;
    function ReadRec(var Rec): Longint;
    procedure First;
    procedure Last;
    procedure NextRec;
    procedure PreviousRec;
    // NumRecs shows the number of records in the stream
    property NumRecs: Longint read GetNumRecs;
    // CurRec reflects the current record in the stream
    property CurRec: Longint read GetCurRec write SetCurRec;
  end;

implementation

function TRecordStream.GetRecSize:Longint;
begin
  { This function returns the size of the record that this stream
    knows about (TPersonRec) }
  Result := SizeOf(TPersonRec);
end;

function TRecordStream.GetNumRecs: Longint;
begin
  // This function returns the number of records in the stream
  Result := Size div GetRecSize;
end;

function TRecordStream.GetCurRec: Longint;
begin
 { This function returns the position of the current record. We must
   add one to this value because the file pointer is always at the
   beginning of the record which is not reflected in the equation:
   Position div GetRecSize }
  Result := (Position div GetRecSize) + 1;
end;

procedure TRecordStream.SetCurRec(RecNo: Longint);
begin
  { This procedure sets the position to the record in the stream
    specified by RecNo. }
  if RecNo > 0 then
    Position := (RecNo - 1) * GetRecSize
  else
    Raise Exception.Create('Cannot go beyond beginning of file.');
end;

function TRecordStream.SeekRec(RecNo: Longint; Origin: Word): Longint;
begin
  { This function positions the file pointer to a location
    specified by RecNo }

  { NOTE: This method does not contain error handling to determine if this
    operation will exceed beyond the beginning/ending of the streamed
    file }
```

continues

LISTING 12.5 Continued

```
  Result := Seek(RecNo * GetRecSize, Origin);
end;

function TRecordStream.WriteRec(Const Rec): Longint;
begin
  // This function writes the record Rec to the stream
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.AppendRec(Const Rec): Longint;
begin
  // This function writes the record Rec to the stream
  Seek(0, 2);
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.ReadRec(var Rec): Longint;
begin
  { This function reads the record Rec from the stream and
    positions the pointer back to the beginning of the record }
  Result := Read(Rec, GetRecSize);
  Seek(-GetRecSize, 1);
end;

procedure TRecordStream.First;
begin
  { This function positions the file pointer to the beginning
     of the stream }
  Seek(0, 0);
end;

procedure TRecordStream.Last;
begin
  // This procedure positions the file pointer to the end of the stream
  Seek(0, 2);
  Seek(-GetRecSize, 1);
end;

procedure TRecordStream.NextRec;
begin
  { This procedure positions the file pointer at the next record
    location }

  { Go to the next record as long as it doesn't extend beyond the
```

```
    end of the file. }
  if ((Position + GetRecSize) div GetRecSize) = GetNumRecs then
    raise Exception.Create('Cannot read beyond end of file')
  else
    Seek(GetRecSize, 1);
end;

procedure TRecordStream.PreviousRec;
begin
{ This procedure positions the file pointer to the previous record
  in the stream }

  { Call this function as long as we don't extend beyond the
    beginning of the file }
  if (Position - GetRecSize >= 0) then
    Seek(-GetRecSize, 1)
  else
    Raise Exception.Create('Cannot read beyond beginning of the  file.');
end;

end.
```

In this unit, you first declare the record that you want to store, `TPersonRec`. `TRecordStream` is the `TFileStream` descendant you use to perform the file I/O for `TPersonRec`. `TRecordStream` has two properties: `NumRecs`, which indicates the number of records in the stream, and `CurRec`, which indicates the current record that the stream is viewing.

The `GetNumRecs()` method, which is the access method for the `NumRecs` property, determines how many records exist in the stream. It does this by dividing the total size of the stream in bytes, as determined from the `TStream.Size` property, by the size of the `TPersonRec` record. Therefore, given that the `TPersonRec` record is 56 bytes, if the `Size` property has the value of `162`, there would be four records in the stream. Note, however, that the record is guaranteed to be 56 bytes only if it's packed. The reason behind this is that structured types, such as records and arrays, are aligned on word or double-word boundaries to allow for faster access. This can mean that the record consumes more space than it actually needs. By using the reserved word `packed` before the record declaration, you can ensure compressed and accurate data storage. Not using the `packed` keyword might cause inaccurate results from the `GetNumRecs()` method.

The `GetCurRec()` method determines which record is the current record. You do this by dividing the `TStream.Position` property by the size of the `TPersonRec` property and adding `1` to the value. The `SetCurRec()` method places the file pointer at the position in the stream at the beginning of the record specified by the `RecNo` property.

The SeekRec() method allows the caller to place the file pointer at a position determined by the RecNo and Origin parameters. This method moves the file pointer forward or backward in the stream from the beginning, ending, or current position of the file pointer, as specified by the value of the Origin property. This is done by using the Seek() method of the TStream object. The use of the TStream.Seek() method is explained in the online "Component Writers Guide" help file.

The WriteRec() method writes the contents of the TPersonRec parameter to the file at the current position, which will be the position of an existing record, so that it will overwrite that record.

The AppendRec() method adds a new record to the end of the file.

The ReadRec() method reads the data from the stream into the TPersonRec parameter. It then repositions the file pointer at the beginning of the record by using the Seek() method. The reason for this is that in order to use the TRecordStream object in a database manner, the file pointer must always be at the beginning of the current record (that is, the record being viewed).

The First() and Last() methods place the file pointer at the beginning and ending of the file, respectively.

The NextRec() method places the file pointer at the beginning of the next record provided that it's not already sitting at the last record in the file.

The PreviousRec() method places the file pointer at the beginning of the preview record provided that the file pointer is not already at the first record in the file.

Using a TFileStream Descendant for File I/O

Listing 12.6 is the source code for the main form of an application that uses the TRecordStream object. This project is FileOfRec.dpr on the CD.

LISTING 21.6 The Source Code for the Main Form of the FileOfRec.dpr Project

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask, Persrec, ComCtrls;

const
  // Declare the file name as a constant
  FName = 'PERSONS.DAT';

type
```

```
  TMainForm = class(TForm)
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtMI: TEdit;
    meAge: TMaskEdit;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblMI: TLabel;
    lblBirthDate: TLabel;
    lblAge: TLabel;
    btnFirst: TButton;
    btnNext: TButton;
    btnPrev: TButton;
    btnLast: TButton;
    btnAppend: TButton;
    btnUpdate: TButton;
    btnClear: TButton;
    lblRecNoCap: TLabel;
    lblRecNo: TLabel;
    lblNumRecsCap: TLabel;
    lblNoRecs: TLabel;
    dtpBirthDay: TDateTimePicker;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure btnAppendClick(Sender: TObject);
    procedure btnUpdateClick(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure btnPrevClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  public
    PersonRec: TPersonRec;
    RecordStream: TRecordStream;
    procedure ShowCurrentRecord;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
```

continues

LISTING 21.6 Continued

```
begin
  { If the file does not exist, then create it, otherwise, open it for
    both read and write access. This is done by instantiating
    a TRecordStream }
  if  FileExists(FName) then
    RecordStream := TRecordStream.Create(FName, fmOpenReadWrite)
  else
    RecordStream := TRecordStream.Create(FName, fmCreate);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  RecordStream.Free; // Free the TRecordStream instance
end;

procedure TMainForm.ShowCurrentRecord;
begin
  // Read the current record.
  RecordStream.ReadRec(PersonRec);
  // Copy the data from the PersonRec to the form's controls
  with PersonRec do
  begin
    edtFirstName.Text := FirstName;
    edtLastName.Text  := LastName;
    edtMI.Text        := MI;
    dtpBirthDay.Date  := BirthDay;
    meAge.Text        := IntToStr(Age);
  end;
  // Show the record number and total records on the main form.
  lblRecNo.Caption  := IntToStr(RecordStream.CurRec);
  lblNoRecs.Caption := IntToStr(RecordStream.NumRecs);
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Display the current record only if one exists.
  if RecordStream.NumRecs <> 0 then
     ShowCurrentRecord;
end;


procedure TMainForm.btnAppendClick(Sender: TObject);
begin
  // Copy the contents of the form controls to the PersonRec record
  with PersonRec do
  begin
    FirstName := edtFirstName.Text;
    LastName  := edtLastName.Text;
    MI        := edtMI.Text;
    BirthDay  := dtpBirthDay.Date;
    Age       := StrToInt(meAge.Text);
  end;
  // Write the new record to the stream
  RecordStream.AppendRec(PersonRec);
  // Display the current record.
  ShowCurrentRecord;
end;

procedure TMainForm.btnUpdateClick(Sender: TObject);
begin
  { Copy the contents of the form controls to the PersonRec and write
    it to the stream }
  with PersonRec do
  begin
    FirstName := edtFirstName.Text;
    LastName  := edtLastName.Text;
    MI        := edtMI.Text;
    BirthDay  := dtpBirthDay.Date;
    Age       := StrToInt(meAge.Text);
  end;
  RecordStream.WriteRec(PersonRec);
end;

procedure TMainForm.btnFirstClick(Sender: TObject);
begin
  { Go to the first record in the stream and display it as long as
    there are records that exist in the stream }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.First;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnNextClick(Sender: TObject);
begin
  // Go to the next record as long as records exist in the stream
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.NextRec;
    ShowCurrentRecord;
```

continues

LISTING 21.6 Continued

```
  end;
end;

procedure TMainForm.btnLastClick(Sender: TObject);
begin
  { Go to the last record in the stream as long as there are records
    in the stream }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.Last;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnPrevClick(Sender: TObject);
begin
  { Go to the previous record in the stream as long as there are records
    in the stream }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.PreviousRec;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  // Clear all controls on the form
  edtFirstName.Text := '';
  edtLastName.Text  := '';
  edtMI.Text        := '';
  meAge.Text        := '';
end;

end.
```

Figure 12.1 shows the main form for this sample project.

The main form contains both a `TPersonRec` field and a `TRecordStream` class. The `TPersonRec` field holds the contents of the current record. The `TRecordStream` instance is created in the form's `OnCreate` event handler. If the file does not exist, it is created. Otherwise, it is opened.

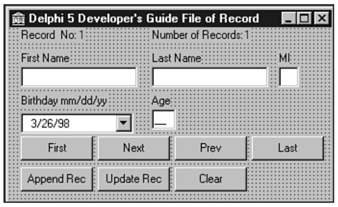


FIGURE 12.1
The main form for the `TRecordStream` example.

The `ShowCurrentRecord()` method is used to extract the current record from the stream by calling the `RecordStream.ReadRec()` method. Recall that the `RecordStream.ReadRec()` method first reads the record, which positions the file pointer to the end of the record after it's read. It then repositions the file pointer at the beginning of the record.

Most of the functionality of this application is discussed in the source commentary. We'll briefly discuss the important points here.

The `btnAppendClick()` adds a new record to the file.

The `btnUpdateClick()` method writes the contents of the form's controls to the file at the position of the current record, thus modifying the contents at that position.

The remaining methods reposition the file pointer to the next, previous, first, and last records in the file, thus enabling you to browse the records in the file.

This example illustrates how you can use typed files to perform simple database operations using standard file I/O. It also illustrates how to make use of the `TFileStream` object to wrap the I/O functionality of the records in the file.

Working with Untyped Files

Up to this point, you've seen how to manipulate both text and typed files. Text files are used to store ASCII character sequences. Typed files store data where each element of that data follows the defined format of a Pascal record structure. In both cases, each file stores a number of bytes that can be interpreted accordingly by applications.

Many files don't follow an ordered format. For example, RTF files, although they do contain text, also contain information about the various attributes of the text within that file. You cannot load these files into any text editor to view them. You must use a view that's capable of interpreting rich text–formatted data.

The next few paragraphs illustrate how to manipulate untyped files.

The following line declares an untyped file:

```
var
  UntypedFile: File;
```

This declares a file consisting of a sequence of blocks, each having 128 bytes of data.

To read data from an untyped file, you would use the `BlockRead()` procedure. To write data to an untyped file, you use the `BlockWrite()` procedure. These procedures are declared as follows:

```
procedure BlockRead(var F: File; var Buf;
➥Count: Integer [; var Result: Integer]);

procedure BlockWrite(var f: File; var Buf;
➥Count: Integer [; var Result: Integer]);
```

Both `BlockRead()` and `BlockWrite()` take three parameters. The first parameter is an untyped file variable, `F`. The second parameter is a variable buffer, `Buf`, which holds the data read from or written to the file. The parameter `Count` contains the number of records to read from the file. The optional parameter `Result` contains the number of records read from the file in a read operation. In a write operation, `Result` contains the number of complete records written. If this value does not equal `Count`, it's possible that the disk has run out of space.

We'll explain what we're referring to when we say that these procedures read or write `Count` records. When you declare an untyped file as follows, by default, this defines a file whose records each consist of 128 bytes of data:

```
UntypedFile: File;
```

This has nothing to do with any particular record structure. It just specifies the size of the block of data that's read in for a single record. Listing 12.7 illustrates how to read one record of 128 bytes from a file:

LISTING 12.7 Reading from an Untyped File

```
var
  UnTypedFile: File;
  Buffer: array[0..128] of byte;
  NumRecsRead: Integer;
begin
  AssignFile(UnTypedFile, 'SOMEFILE.DAT');
  Reset(UnTypedFile);
  try
    BlockRead(UnTypedFile, Buffer, 1, NumRecsRead);
  finally
    CloseFile(UnTypedFile);
  end;
end;
```

Here, you open the file `SOMEFILE.DAT` and read 128 bytes of data (one record or block) into the buffer appropriately named `Buffer`. To write 128 bytes of data to a file, take a look at Listing 12.8.

LISTING 12.8 Writing Data to an Untyped File

```
var
  UnTypedFile: File;
  Buffer: array[0..128] of byte;
  NumRecsWritten: Integer;
begin
  AssignFile(UnTypedFile, 'SOMEFILE.DAT');
  // If file doesn't exist, create it. Otherwise,
  // just open it for read/write access
  if FileExists('SOMEFILE.DAT') then
    Reset(UnTypedFile)
  else
    Rewrite(UnTypedFile);
  try
    // Position the file pointer to the end of the file
    Seek(UnTypedFile, FileSize(UnTypedFile));
    FillChar(Buffer, SizeOf(Buffer), 'Y');
    BlockWrite(UnTypedFile, Buffer, 1, NumRecsWritten);
  finally
    CloseFile(UnTypedFile);
  end;
end;
```

A problem in using the default block size of 128 bytes when reading from a file is that its size must be a multiple of 128 to avoid reading beyond the end of the file. You can get around this by specifying a record size of one byte with the `Reset()` procedure. If you pass a record size of one byte, reading blocks of any size will always be a multiple of one byte. As an example, Listing 12.9 illustrates a simple file-copy routine using the `Blockread()` and `BlockWrite()` procedures.

LISTING 12.9 A File-Copy Demo

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, Gauges;
```

continues

LISTING 12.9 Continued

```
type
  TMainForm = class(TForm)
    prbCopy: TProgressBar;
    btnCopy: TButton;
    procedure btnCopyClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnCopyClick(Sender: TObject);
var
  SrcFile, DestFile: File;
  BytesRead, BytesWritten, TotalRead: Integer;
  Buffer: array[1..500] of byte;
  FSize: Integer;
begin
  { Assign both the source and destination files to their
    respective file variables }
  AssignFile(SrcFile, 'srcfile.tst');
  AssignFile(DestFile, 'destfile.tst');
  // Open the source file for read access.
  Reset(SrcFile, 1);
  try
    // Open destination file for write access.
    Rewrite(DestFile, 1);
    try
      { Encapsulate this into a try..except so that we can erase the file if
        an error occurs. }
      try
        // Initialize total bytes read to zero.
        TotalRead := 0;
        // Obtain the filesize of the source file
        FSize := FileSize(SrcFile);
        { Read SizeOf(Buffer) bytes from the source file
          and add these bytes to the destination file. Repeat this
          process until all bytes have been read from the source
          file. A progress bar is provided to show the progress of the
          copy operation. }
        repeat
          BlockRead(SrcFile, Buffer, SizeOf(Buffer), BytesRead);
          if BytesRead > 0 then
          begin
            BlockWrite(DestFile, Buffer, BytesRead, BytesWritten);
            if BytesRead <> BytesWritten then
              raise Exception.Create('Error copying file')
            else begin
              TotalRead := TotalRead + BytesRead;
              prbCopy.Position := Trunc(TotalRead / Fsize) * 100;
              prbCopy.Update;
            end;
          end
        until BytesRead = 0;
      except
        { On an exception, erase the destination file as it may be
          corrupt. Then re-raise the exception. }
        Erase(DestFile);
        raise;
      end;
    finally
      CloseFile(DestFile); // Close the destination file.
    end;
  finally
    CloseFile(SrcFile);   // Close the source file.
  end;
end;

end.
```

NOTE

One of the demos that ships with Delphi 5 comes with several useful file-handling functions, including a function to copy a file. This demo is in the `\DEMOS\DOC\FILMANEX\` directory. Here are the functions contained in the `FmxUtils.PAS` file:

```
procedure CopyFile(const FileName, DestName: string);
procedure MoveFile(const FileName, DestName: string);
function GetFileSize(const FileName: string): LongInt;
function FileDateTime(const FileName: string): TDateTime;
function HasAttr(const FileName: string; Attr: Word): Boolean;
function ExecuteFile(const FileName, Params,
DefaultDir: string; ShowCmd: Integer): THandle;
```

Also, later in this chapter we show you how to copy files and entire directories using the `ShFileOperation()` function.

First, the demo opens a source file for input and creates a destination file to which the source file's data will be copied. The variables `TotalRead` and `FSize` are used in updating a `TProgressBar` component to indicate the status of the copy operation. Inside the `repeat` loop is where the copy operation is actually performed. First, `SizeOf(Buffer)` bytes are read from the source file. The variable `BytesRead` determines the actual number of bytes read. Then, an attempt is made to copy `BytesRead` to the destination file. The number of actual bytes written is stored in the variable `BytesWritten`. At this point, if no error has occurred, `BytesRead` and `BytesWritten` will have the same values. This process is continued until all bytes of the file have been copied. If an error occurs, an exception is raised and the destination file is erased from the disk.

A sample application illustrating the preceding code exists on the CD as `FileCopy.dpr`.

The TTextRec and TFileRec Record Structures

Most file-management functions are really operating system functions or interrupts that have been wrapped up in Object Pascal routines. The `Reset()` function, for example, is really a Pascal wrapper to `CreateFileA()`, a Win32 function of the KERNEL32 dynamic link library. By wrapping up these Win32 functions into Object Pascal functions, you do not have to worry about the implementation details of these file operations. However, it also obscures how to access certain file details when needed (such as the file handle) because these are hidden for Object Pascal's usage.

When using nonnative Object Pascal functions that require a file handle, such as `LZCopy()`, you can get the file handle by typecasting your text file and binary file variables as `TTextRec` and `TFileRec`, respectively. These record types contain the file handle as well as other file details. Other than the file handle, you rarely will (and probably shouldn't) access the other data fields. The correct procedure for getting to the handle follows:

```
TFileRec(MyFileVar).Handle
```

The definition of the `TTextRec` record is shown here:

```
PTextBuf = ^TTextBuf;
TTextBuf = array[0..127] of Char; // Buffer definition for first 127
                                  //    characters in the file.
TTextRec = record
    Handle: Integer;                // File handle
    Mode: Integer;                  // File mode
    BufSize: Cardinal;              // The following 4 parameters are
    BufPos: Cardinal;               //   used for memory buffering.
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;              // The XXXXFunc are points to file
    InOutFunc: Pointer;            //   access functions. They can be
    FlushFunc: Pointer;            //   modified when writing certain
    CloseFunc: Pointer;            //   file device drivers.
    UserData: array[1..32] of Byte; // Not used.
    Name: array[0..259] of Char;   // File's full path name
    Buffer: TTextBuf;              // Buffer containing the
➥first 127 characters of the file
  end;
```

Here's the definition of the TFileRec record structure:

```
TFileRec = record
  Handle: Integer;                 // File Handle
  Mode: Integer;                   // File mode
  RecSize: Cardinal;               // Size of each file record
  Private: array[1..28] of Byte;   // Used internally by Object Pascal
  UserData: array[1..32] of Byte;  // Not used.
  Name: array[0..259] of Char;     // File's full path name
end;
```

Working with Memory-Mapped Files

Probably one of the most convenient features of the Win32 environment is the ability to access files on disk as if you were accessing the file's contents in memory. This capability is provided through memory-mapped files.

Memory-mapped files enable you to avoid having to perform all the I/O operations on the file. Instead, you reserve a range of virtual address space and commit the physical storage of the file on disk to the address of this reserved memory space. You then reference the contents of the file through a pointer into this reserved region. Shortly, we'll show you how you can use this capability to create a useful text-searching utility for text files, made simple through the use of memory-mapped files.

Purposes for Memory-Mapped Files

Typically, there are three uses for memory-mapped files:

- The Win32 system loads and executes EXE and DLL files by using memory-mapped files. This conserves paging file space and therefore decreases the load time for such files.
- Memory-mapped files can be used to access data residing in the mapped file through a pointer to the mapped memory region. This not only simplifies data access, but also relieves you from having to code various file-buffering schemes.
- Memory-mapped files can be used to provide the ability to share data among different processes running on the same machine.

We won’t discuss the first purpose for memory-mapped files because this really applies to the system behavior. In this chapter, we discuss the second purpose of memory-mapped files because this is a use that you, as a developer, will most likely need at some point. Chapter 9, “Dynamic Link Libraries,” shows you how to share data with other processes by using memory-mapped files. You might want to look back at this example after reading this section so that you fully understand what we showed you.

Using Memory-Mapped Files

When you create a memory-mapped file, you’re essentially associating the file to an area in the process’s virtual memory address space. To create this association, you must create a *file-mapping object*. To view/edit the contents of a file, you must have a *file view* for the file-mapping object. This enables you to access the contents of the file through a pointer as though you were accessing an area of memory.

When you write to the file view, the system handles the caching, buffering, writing and loading of the file’s data, as well as memory allocation and deallocation. As far as you’re concerned, you’re editing data residing in an area of memory. The file I/O is handled entirely by the system. This is the beauty of using memory-mapped files. Your task of file manipulation is greatly simplified over the standard file I/O techniques discussed previously and is usually faster as well.

The following sections cover the steps required to create/open a memory-mapped file.

Creating/Opening the File

The first step in creating/opening a memory-mapped file is to obtain the file handle for the file to be mapped. You can do this by using either the `FileCreate()` or `FileOpen()` functions. `FileCreate()` is defined in the `SysUtils.pas` unit as follows:

```
function FileCreate(const FileName: string): Integer;
```

This function creates a new file with the filename specified by its `FileName` string parameter. If the function is successful, a valid file handle is returned. Otherwise, the value defined by the constant `INVALID_HANDLE_VALUE` is returned.

`FileOpen()` opens an existing file using a specified access mode. This function, when successful, will return a valid file handle. Otherwise, it will return the value defined by the constant `INVALID_HANDLE_VALUE`. `FileOpen()` is defined in the `SysUtils.pas` unit as follows:

```
function FileOpen(const FileName: string; Mode: Word): Integer;
```

The first parameter is the full path name of the file to which the mapping is to be applied. The second parameter is one of the file-access modes described in Table 12.1.

TABLE 12.1 `fmOpenXXXX` File Access Modes

Access Mode	*Meaning*
`fmOpenRead`	Enables you to read only from the file
`fmOpenWrite`	Enables you to write only to the file
`fmOpenReadWrite`	Enables you to read from and write to the file

If you specify a value of `0` as the `Mode` parameter, you won't be able to read from or write to the specified file. You might use this when all you want is to obtain various file attributes. You can specify how a file can be shared with different applications by applying the bitwise `or` operation using the access modes specified in Table 12.1 with one of the `fmShare`*`XXXX`* modes. The `fmShare`*`XXXX`* modes are listed in Table 12.2.

TABLE 12.2 `fmShareXXXX` File Share Modes

Share Mode	*Meaning*
`fmShareCompat`	The file-sharing mechanism is compatible with DOS 1.*x* and 2.*x* file control blocks. This is used in conjunction with other `FmShare`*`XXXX`* modes.
`fmShareExclusive`	No sharing allowed.
`fmShareDenyWrite`	Other attempts to open the file with `fmOpenWrite` access fail.
`fmShareDenyRead`	Other attempts to open the file with `fmOpenRead` access fail.
`fmShareDenyNone`	Other attempts to open the file with any mode succeed.

After a valid file handle is obtained, it's possible to obtain a file-mapping object.

Creating the File-Mapping Object

To create named or unnamed file-mapping objects, you use the `CreateFileMapping()` function. This function is defined as follows:

```
function CreateFileMapping(
hFile: THandle;
lpFileMappingAttributes: PSecurityAttributes;
flProtect,
dwMaximumSizeHigh,
dwMaximumSizeLow: DWORD;
lpName: PChar) : THandle;
```

The parameters passed to `CreateFileMapping()` give the system the necessary information required to create the file-mapping object. The first parameter, `hFile`, is the file handle obtained from the previous call to `FileOpen()` or `FileCreate()`. It's important that the file be opened with the protection flags compatible with the `flProtect` parameter, which we'll discuss momentarily. Another method is to use `CreateFileMapping()` to create a file-mapping object backed by the system paging file. This technique is used to enable the sharing of data among separate processes that we illustrate in Chapter 9, "Dynamic Link Libraries."

The `lpFileMappingAttributes` parameter is a `PSecurityAttributes` pointer, which refers to the security attributes for the file-mapping object. This parameter will almost always be null.

The `flProtect` parameter specifies the type of protection applied to the file view. As we mentioned before, this value must be compatible with the attributes under which the file was opened to obtain a file handle. Table 12.3 lists the various attributes that can be assigned to the `flProtect` parameter.

TABLE 12.3 Protection Attributes

Protection Attribute	*Meaning*
`PAGE_READONLY`	You can read the file's contents. The file must have been created with the `FileCreate()` function or opened with `FileOpen()` and an access mode of `fmOpenRead.`
`PAGE_READWRITE`	You can read and write to the file. The file must have been opened with the `fmOpenReadWrite` access mode.
`PAGE_WRITECOPY`	You can read and write to the file. However, when you write to the file, a private copy of the modified page is created. The significance of this is that memory-mapped files that are shared between processes do not consume twice the resources in system memory or swap file usage. Only the memory required for the pages that are different is duplicated. The file must have been opened with the `fmOpenWrite` or `fmOpenReadWrite` access.

You can also apply section attributes to the `flProtect` parameter by using the bitwise `or` operator. Table 12.4 explains the meaning of these attributes.

TABLE 12.4 Section Attributes

Section Attribute	*Meaning*
`SEC_COMMIT`	Allocates physical storage in memory or in the paging file for all pages in a section. This is the default value.
`SEC_IMAGE`	File-mapping information and attributes are taken from the file image. This applies to executable image files only. (Note that this attribute is ignored under Windows 95/98.)

Section Attribute	*Meaning*
`SEC_NOCACHE`	No memory-mapped pages are cached. Therefore, the system applies all file writes directly to the file's data on disk. This mainly applies to device drivers and not to applications. (Note that this attribute is ignored under Windows 95/98.)
`SEC_RESERVE`	Reserves pages of a section without allocating physical storage.

The `dwMaximumSizeHigh` parameter specifies the high-order 32 bits of the file-mapping object's maximum size. Unless you're accessing files larger than 4GB, this value will always be zero.

The `dwMinimumSizeLow` parameter specifies the low-order 32 bits of the file-mapping object's maximum size. A value of zero for this parameter would indicate a maximum size for the file-mapping object equal to the size of the file being mapped.

The `lpName` parameter specifies the name of the file-mapping object. This value may contain any character except a backslash character (`\`). If this parameter matches the name of an existing file-mapping object, this function requests access to that same file-mapping object using the attributes specified by the `flProtect` parameter. It's valid to pass `nil` as this parameter, which creates a nameless file-mapping object.

If `CreateFileMapping()` is successful, it returns a valid handle to a file-mapping object. If this file-mapping object happens to refer to an already existing file-mapping object, the value `ERROR_ALREADY_EXISTS` will be returned from the `GetLastError()` function. If `CreateFileMapping()` fails, it returns a `nil` value. You must call the `GetLastError()` function to determine the reason for failure.

CAUTION

Under Windows 95/98, do not use file I/O functions on file handles that have been used to create file mappings. The data in such files may not be coherent. It is therefore recommended that you open the file with exclusive access. See the section "Memory-Mapped File Coherence."

After you've obtained a valid file-mapping object, you can map the file's data into the process's address space.

Mapping a View of the File into the Process's Address Space

The `MapViewOfFile()` function maps a view of the file into the process's address space. This function is defined as follows:

```
function MapViewOfFile(
hFileMappingObject: THandle;
dwDesiredAccess: DWORD;
dwFileOffsetHigh,
dwFileOffsetLow,
dwNumberOfBytesToMap: DWORD): Pointer;
```

`hFileMappingObject` is the handle to an open file-mapping object that was opened with a call to either the `CreateFileMapping()` or `OpenFileMapping()` function.

The `dwDesiredAccess` parameter indicates how the file data is to be accessed and may be one of the values specified in Table 12.5.

TABLE 12.5 Desired Access to File View

dwDesiredAccess Value	*Meaning*
`FILE_MAP_WRITE`	Allows read-write access to the file data. The `PAGE_READ_WRITE` attribute must have been used with the `CreateFileMapping()` function.
`FILE_MAP_READ`	Allows read-only access to the file data. The `PAGE_READ_WRITE` or `PAGE_READ` attribute must have been used with the `CreateFileMapping()` function.
`FILE_MAP_ALL_ACCESS`	Same access provided by using `FILE_MAP_WRITE`.
`FILE_MAP_COPY`	Enables copy-on-write access. When you write to the file, a private copy of the page written to is created. `CreateFileMapping()` must have been used with the `PAGE_READ_ONLY`, `PAGE_READ_WRITE`, or `PAGE_WRITE_COPY` attributes.

The `dwFileOffsetHigh` parameter specifies the high-order 32 bits of the file offset where the file mapping begins.

The `dwFileOffsetLow` parameter specifies the lower-order 32 bits of the file offset where mapping begins.

The `dwNumberOfBytesToMap` parameter indicates how many bytes of the file to map. A zero value indicates the entire file.

`MapViewOfFile()` returns the starting address of the mapped view. If it's unsuccessful, `nil` is returned and you must call the `GetLastError()` function to determine the cause of the error.

Unmapping the View of the File

The `UnmapViewOfFile()` function unmaps the view of the file from the calling process's address space. This function is defined as follows:

```
function UnmapViewOfFile(lpBaseAddress: Pointer): BOOL;
```

This function's single parameter `lpBaseAddress` must point to the base address of the mapped region. This is the same value returned from the `MapViewOfFile()` function.

You need to call `UnmapViewOfFile()` when you've finished working with the file; otherwise, the mapped region of memory will not get released by the system until your process terminates.

Closing the File-Mapping and File Kernel Objects

The calls to `FileOpen()` and `CreateFileMapping()` are both open kernel objects that you're responsible for closing. This is done by using the `CloseHandle()` function. `CloseHandle()` is defined as follows:

```
function CloseHandle(hObject: THandle): BOOL;
```

If the call to `CloseHandle()` is successful, it will return `True`. Otherwise, it will return `False`, and you'll have to examine the result of `GetLastError()` to determine the cause of the error.

A Simple Memory-Mapped File Example

To illustrate the use of the memory-mapped file functions, examine Listing 12.10. You can find this project on the CD as `TextUpper.dpr`.

LISTING 12.10 A Simple Memory-Mapped File Example

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

const
  FName = 'test.txt';

type

  TMainForm = class(TForm)
    btnUpperCase: TButton;
    memTextContents: TMemo;
    lblContents: TLabel;
    btnLowerCase: TButton;
    procedure btnUpperCaseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnLowerCaseClick(Sender: TObject);
```

continues

LISTING 12.10 Continued

```
  public
    UCase: Boolean;
    procedure ChangeFileCase;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnUpperCaseClick(Sender: TObject);
begin
  UCase := True;
  ChangeFileCase;
end;

procedure TMainForm.btnLowerCaseClick(Sender: TObject);
begin
  UCase := False;
  ChangeFileCase;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  memTextContents.Lines.LoadFromFile(FName);
  // Change to upper case by default.
  UCase := True;
end;

procedure TMainForm.ChangeFileCase;
var
  FFileHandle: THandle; // Handle to the open file.
  FMapHandle: THandle;  // Handle to a file-mapping object
  FFileSize: Integer;   // Variable to hold the file size.
  FData: PByte;         // Pointer to the file's data when mapped.
  PData: PChar;         // Pointer used to reference the file data.
begin

  { First obtain a file handle to the file to be mapped. This code
    assumes the existence of the file. Otherwise, you can use the
    FileCreate() function to create a new file. }

  if not FileExists(FName) then
```

```
    raise Exception.Create('File does not exist.')
  else
    FFileHandle := FileOpen(FName, fmOpenReadWrite);

  // If CreateFile() was not successful, raise an exception
  if FFileHandle = INVALID_HANDLE_VALUE then
    raise Exception.Create('Failed to open or create file');


  try
    { Now obtain the file size which we will pass to the other file-
      mapping functions. We'll make this size one byte larger as we
      need to append a null-terminating character to the end of the
      mapped-file's data.}
    FFileSize := GetFileSize(FFileHandle, Nil);

   { Obtain a file-mapping object handle. If this function is not
      successful, then raise an exception. }
    FMapHandle := CreateFileMapping(FFileHandle, nil,
       PAGE_READWRITE, 0, FFileSize, nil);

    if FMapHandle = 0 then
      raise Exception.Create('Failed to create file mapping');
  finally
    // Release the file handle
    CloseHandle(FFileHandle);
  end;

  try
    { Map the file-mapping object to a view. This will return a pointer
      to the file data. If this function is not successful, then raise
      an exception. }
    FData := MapViewOfFile(FMapHandle, FILE_MAP_ALL_ACCESS, 0, 0, FFileSize);

    if FData = Nil then
      raise Exception.Create('Failed to map view of file');

  finally
    // Release the file-mapping object handle
    CloseHandle(FMapHandle);
  end;

  try
    { !!! Here is where you would place the functions to work with
    the mapped file's data. For example, the following line forces
    all characters in the file to uppercase }
    PData := PChar(FData);
```

continues

LISTING 12.10 Continued

```
    // Position the pointer to the end of the file's data
    inc(PData, FFileSize);

    // Append a null-terminating character to the end of the file's data
    PData^ := #0;

    // Now set all characters in the file to upper-case
    if UCase then
      StrUpper(PChar(FData))
    else
      StrLower(PChar(FData));

  finally
    // Release the file mapping.
    UnmapViewOfFile(FData);
  end;
  memTextContents.Lines.Clear;
  memTextContents.Lines.LoadFromFile(FName);
end;

end.
```

You'll see in Listing 12.10 that the first step is to obtain a handle to the file to be mapped to the process's region of memory. This is done by calling the `FileOpen()` function. You pass the `fmOpenReadWrite` file-access mode to this function to give you read/write access to the file's contents.

Next, you obtain the size of the file and change the last character to a null terminator. This should actually be the end-of-file marker, which is the same byte value as the null-terminator. You do it here for clarity. The point is that because you're manipulating the file's data as a null-terminating string, you need to ensure that a null-terminator is present.

The following step obtains the memory-mapping file object by calling `CreateFileMapping()`. If this function fails, you raise an exception. Otherwise, you go on to the next step to map the file-mapping object to a view. Again, you raise an exception if this function fails.

You then change the case of the data in the file. If you were to view the file in a text editor after executing this routine, you would see that the file's characters have all been converted to the selected case. Lastly, you unmap the view of the file by calling the `UnMapViewOfFile()` function.

You might have noticed that in this code, you release both the file handle and the file-mapping object's handle before you even manipulate the file's data after it has been mapped to a view.

This is possible because the system keeps a usage count on the file handle and file-mapping object when the call to `MapViewOfFile()` is made. Therefore, you can close the object up front by calling `CloseHandle()`, thus reducing the chances of causing a resource leak. Later, you'll see a more elaborate use of memory-mapped files as you build a `TMemoryMapFile` class and use it to perform text searches through text files.

Memory-Mapped File Coherence

The Win32 system ensures that multiple views of a file remain coherent as long as they're mapped using the same file-mapping object. This means that if one view modifies the contents of a file, a second view will realize those modifications. Keep in mind, however, that this is only true when using the same file-mapping objects. When you're using different file-mapping objects, multiple views are not guaranteed to be coherent. This particular problem exists only with files that are mapped for write access. Read-only files are always coherent. Also, files shared over a network are not kept coherent in write-file mappings in different machines.

The Text-File Search Utility

To illustrate a practical use of memory-mapped files, we've created a project that performs a text search on text files in the current directory. The filenames, along with the number of times a string is found in the file, are added to a list box on the main form. The main form for this project is shown in Figure 12.2. You can find this project on the CD as `FileSrch.dpr`.

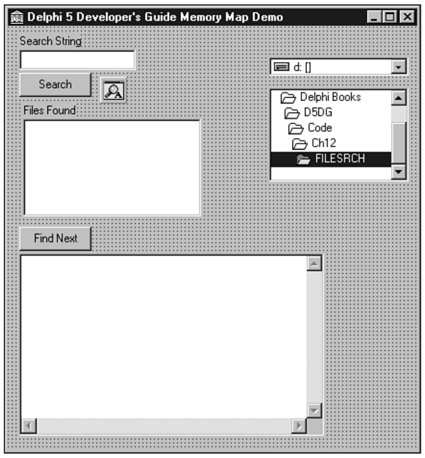


FIGURE 12.2
The main form for the text search project.

This project also illustrates how to encapsulate the functionality of memory-mapped files into an object. To show this, we've created the `TMemMapFile` class.

The TMemMapFile Class

The unit containing the `TMemMapFile` class is shown in Listing 12.11.

LISTING 12.11 The Source Code for `MemMap.pas`, the Unit Defining the `TMemMapFile` Class

```
unit MemMap;

interface

uses Windows, SysUtils, Classes;

type
  EMMFError = class(Exception);

  TMemMapFile = class
  private
    FFileName: String;    // File name of the mapped file.
    FSize: Longint;       // Size of the mapped view
    FFileSize: Longint;   // Actual File Size
    FFileMode: Integer;   // File access mode
    FFileHandle: Integer; // File handle
    FMapHandle: Integer;  // Handle to the file mapping object.
    FData: PByte;         // Pointer to the file's data
    FMapNow: Boolean;     // Determines whether or
                          //   not to map view of immediately.
    procedure AllocFileHandle;
    { Retrieves a handle to the disk file. }
    procedure AllocFileMapping;
    { Retrieves a file-mapping object handle }
    procedure AllocFileView;
    { Maps a view to the file }
    function GetSize: Longint;
    { Returns the size of the mapped view }
  public
    constructor Create(FileName: String; FileMode: integer;
                       Size: integer; MapNow: Boolean); virtual;
    destructor Destroy; override;
    procedure FreeMapping;
    property Data: PByte read FData;
    property Size: Longint read GetSize;
    property FileName: String read FFileName;
    property FileHandle: Integer read FFileHandle;
    property MapHandle: Integer read FMapHandle;
  end;

implementation
```

```
constructor TMemMapFile.Create(FileName: String; FileMode: integer;
                               Size: integer; MapNow: Boolean);
{ Creates Memory Mapped view of FileName file.
  FileName: Full pathname of file.
  FileMode: Use fmXXX constants.
  Size: size of memory map.  Pass zero as the size to use the
        file's own size.
}
begin

  { Initialize private fields }
  FMapNow := MapNow;
  FFileName := FileName;
  FFileMode := FileMode;

  AllocFileHandle;  // Obtain a file handle of the disk file.
  { Assume file is < 2 gig  }

  FFileSize := GetFileSize(FFileHandle, Nil);
  FSize := Size;

  try
    AllocFileMapping; // Get the file mapping object handle.
  except
    on EMMFError do
    begin
      CloseHandle(FFileHandle);  // close file handle on error
      FFileHandle := 0;          // set handle back to 0 for clean up
      raise;                     // re-raise exception
    end;
  end;
  if FMapNow then
    AllocFileView;  // Map the view of the file
end;

destructor TMemMapFile.Destroy;
begin

  if FFileHandle <> 0 then
    CloseHandle(FFileHandle); // Release file handle.

  { Release file mapping object handle }
  if FMapHandle <> 0 then
    CloseHandle(FMapHandle);

  FreeMapping; { Unmap the file mapping view . }
```

continues

LISTING 12.11 Continued

```
  inherited Destroy;
end;

procedure TMemMapFile.FreeMapping;
{ This method unmaps the view of the file from this process's address
  space. }
begin
  if FData <> Nil then
  begin
    UnmapViewOfFile(FData);
    FData := Nil;
  end;
end;

function TMemMapFile.GetSize: Longint;
begin
  if FSize <> 0 then
    Result := FSize
  else
    Result := FFileSize;
end;

procedure TMemMapFile.AllocFileHandle;
{ creates or opens disk file before creating memory mapped file }
begin
  if FFileMode = fmCreate then
    FFileHandle := FileCreate(FFileName)
  else
    FFileHandle := FileOpen(FFileName, FFileMode);

  if FFileHandle < 0 then
    raise EMMFError.Create('Failed to open or create file');
end;

procedure TMemMapFile.AllocFileMapping;
var
  ProtAttr: DWORD;
begin
  if FFileMode = fmOpenRead then  // obtain correct protection attribute
    ProtAttr := Page_ReadOnly
  else
    ProtAttr := Page_ReadWrite;
  { attempt to create file mapping of disk file.
    Raise exception on error. }
  FMapHandle := CreateFileMapping(FFileHandle, Nil, ProtAttr,
      0, FSize, Nil);
  if FMapHandle = 0 then
    raise EMMFError.Create('Failed to create file mapping');
end;

procedure TMemMapFile.AllocFileView;
var
  Access: Longint;
begin
  if FFileMode = fmOpenRead then // obtain correct file mode
    Access := File_Map_Read
  else
    Access := File_Map_All_Access;
  FData := MapViewOfFile(FMapHandle, Access, 0, 0, FSize);
  if FData = Nil then
    raise EMMFError.Create('Failed to map view of file');
end;

end.
```

The commentary lists the purpose of the various fields and methods for the `TMemMapFile` class.

The class contains the methods `AllocFileHandle()`, `AllocFileMapping()`, and `AllocFileView()` to retrieve the file handle, file-mapping object handle, and a view to the specified file, respectively.

The `Create()` constructor is where the fields get initialized and the methods to allocate the various handles get called. Failure of any of those methods results in an exception being raised. The `Destroy()` destructor ensures that the view gets unmapped by calling the `UnMapViewOfFile()` method.

Using the TMemMapFile Class

The main form for the file-search project is shown in Listing 12.12.

LISTING 12.12 The Source Code for the Main Form for the File-Search Project

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons, FileCtrl;

type
```

continues

LISTING 12.12 Continued

```
  TMainForm = class(TForm)
    btnSearch: TButton;
    lbFilesFound: TListBox;
    edtSearchString: TEdit;
    lblSearchString: TLabel;
    lblFilesFound: TLabel;
    memFileText: TMemo;
    btnFindNext: TButton;
    FindDialog: TFindDialog;
    dcbDrives: TDriveComboBox;
    dlbDirectories: TDirectoryListBox;
    procedure btnSearchClick(Sender: TObject);
    procedure lbFilesFoundClick(Sender: TObject);
    procedure btnFindNextClick(Sender: TObject);
    procedure FindDialogFind(Sender: TObject);
    procedure edtSearchStringChange(Sender: TObject);
    procedure memFileTextChange(Sender: TObject);
  public
  end;

var
  MainForm: TMainForm;

implementation
uses MemMap, Search;

{$R *.DFM}

procedure TMainForm.btnSearchClick(Sender: TObject);
var
  MemMapFile: TMemMapFile;
  SearchRec: TSearchRec;
  RetVal: Integer;
  FoundStr: PChar;
  FName: String;
  FindString: String;
  WordCount: Integer;
begin
  memFileText.Lines.Clear;
  btnFindNext.Enabled := False;
  lbFilesFound.Items.Clear;

  { Retrieve each text file on which the text search is to be
    performed. Use the FindFirst/FindeNext sequence on this search. }
  RetVal := FindFirst(dlbDirectories.Directory+'\*.txt', faAnyFile, SearchRec);
  try
    while RetVal = 0 do
    begin
      FName := SearchRec.Name;

      // Open the memory mapped file for read-only access.
      MemMapFile := TMemMapFile.Create(FName, fmOpenRead, 0, True);
      try

        { Use a temporary storage for the search string }
        FindString := edtSearchString.Text;

        WordCount := 0; // Initialize the WordCount to zero
        { Get the first occurrence of the search string  }
        FoundStr := StrPos(PChar(MemMapFile.Data), PChar(FindString));

        if FoundStr <> nil then
        begin
          { Continue to search through the remaining text of the file
            for occurrences of the search string. On each find,
            increment the WordCount variable }
          repeat
            inc(WordCount);
            inc(FoundStr, Length(FoundStr));

            { Retrieve the next occurrence of the search string. }
            FoundStr := StrPos(PChar(FoundStr), PChar(FindString));
          until FoundStr = nil;
          { Add the file's name to the list box }
          lbFilesFound.Items.Add(SearchRec.Name +
             ' - '+IntToStr(WordCount));
        end;
        { Retrieve the next file on which to perform the search }
        RetVal := FindNext(SearchRec);
      finally
        MemMapFile.Free; { Free the memory mapped file instance }
      end;
    end;
  finally
    FindClose(SearchRec);
  end;
end;

procedure TMainForm.lbFilesFoundClick(Sender: TObject);
var
  FName: String;
```

continues

LISTING 12.12 Continued

```
  B: Byte;
begin
  with lbFilesFound do
    if ItemIndex <> -1 then
    begin
      B := Pos(' ', Items[ItemIndex]);
      FName := Copy(Items[ItemIndex], 1, B);
      memFileText.Clear;
      memFileText.Lines.LoadFromFile(FName);
    end;
end;

procedure TMainForm.btnFindNextClick(Sender: TObject);
begin
  FindDialog.FindText := edtSearchString.Text;
  FindDialog.Execute;
  FindDialog.Top := Top+Height;
  FindDialogFind(FindDialog);
end;

procedure TMainForm.FindDialogFind(Sender: TObject);
begin
  with Sender as TFindDialog do
    if not SearchMemo(memFileText, FindText, Options) then
      ShowMessage('Cannot find "' + FindText + '".');
end;

procedure TMainForm.edtSearchStringChange(Sender: TObject);
begin
  btnSearch.Enabled := edtSearchString.Text <> EmptyStr;
end;

procedure TMainForm.memFileTextChange(Sender: TObject);
begin
  btnFindNext.Enabled := memFileText.Lines.Count > 0;
end;

end.
```

This project performs a case-sensitive search on text files in the current directory.

`btnSearchClick()` contains the code that performs the actual search, determines the number of times the specified string is found in each file, and adds the files containing the search string to `lbFilesFound`.

It first uses a FindFirst()/FindNext() sequence of calls to find each file with a .txt extension in the current directory. Both these functions are discussed later in this chapter. The method then uses a TMemMapFile class on the temporary file to get access to the file's data. This file is opened with read-only access because you won't be modifying it. The following lines of code perform the logic required to obtain a count of the number of times the search string occurs in the file:

```
if FoundStr <> nil then
begin
  repeat
    inc(WordCount);
    inc(FoundStr, length(FoundStr));
    FoundStr := StrPos(PChar(FoundStr), PChar(FindString))
  until FoundStr = nil;
```

Both the filename and number of occurrences of the search string in the file are added to lbFilesFound.

When the user double-clicks a TListBox item, the file is loaded into the TMemo control, where the user can locate each occurrence of the search string by clicking the Find Next button.

The btnFindNext() event handler initializes the FindDialog.FindText property to the string in edtSearchString. It then invokes FindDialog.

When the user clicks the Find Next button on FindDialog, its OnFind event handler gets invoked. This event handler is FindDialogFind(). FindDialogFind() uses the function SearchMemo(), which is defined in the unit Search.pas.

SearchMemo() scans the text of any TCustomEdit descendant and selects that text, which brings it into view.

NOTE

The Search.pas unit is a file that ships with Borland Delphi 1.0 as one of its demos. We obtained permission to include this file on the CD-ROM accompanying this book from Borland. This unit does not make use of various string-handling features because it was designed for Delphi 1.0. We did, however, make a minor change to allow a TMemo control to bring the caret into view, which was done automatically in Windows 3.1. In Win32, you must pass an EM_SCROLLCARET message to the TMemo control after setting its SelStart property. Read the comments in Search.pas for further information.

Directories and Drives

You can perform several tasks that you might find useful in your applications with the drives installed on a system and the directories on those drives. The next several sections cover some of these tasks.

Obtaining a List of Available Drives and Drive Types

To obtain a list of available drives on your system, you use the `GetDriveType()` Win32 API function. This function takes a `PChar` parameter and returns an integer value representing one of the drive types specified in Table 12.6.

TABLE 12.6 `GetDriveType()` Return Values

Return Value	*Meaning*
`0`	Cannot determine the drive type.
`1`	Root directory does not exist.
`DRIVE_REMOVABLE`	Drive is removable.
`DRIVE_FIXED`	Drive is not removable.
`DRIVE_REMOTE`	Drive is a remote (network) drive.
`DRIVE_CDROM`	Drive is a CD-ROM drive.
`DRIVE_RAMDISK`	Drive is a RAM disk.

Listing 12.13 illustrates how you would use the `GetDriveType()` function.

LISTING 12.13 Use of the `GetDriveType()` Function

```
procedure TMainForm.btnGetDriveTypesClick(Sender: TObject);
var
  i: Integer;
  C: String;
  DType: Integer;
  DriveString: String;
begin
  { Loop from A..Z to determine available drives }
  for i := 65 to 90 do
  begin
    C := chr(i)+':\'; // Format a string to represent the root directory.
    { Call the GetDriveType() function which returns an integer
      value representing one of the types shown in the case statement
      below }
    DType := GetDriveType(PChar(C));
    { Based on the drive type returned, format a string to add to
      the listbox displaying the various drive types. }
    case DType of
      0: DriveString := C+' The drive type cannot be determined.';
      1: DriveString := C+' The root directory does not exist.';
      DRIVE_REMOVABLE: DriveString :=
         C+' The drive can be removed from the drive.';
      DRIVE_FIXED: DriveString :=
         C+' The disk cannot be removed from the drive.';
      DRIVE_REMOTE: DriveString :=
         C+' The drive is a remote (network) drive.';
      DRIVE_CDROM: DriveString := C+' The drive is a CD-ROM drive.';
      DRIVE_RAMDISK: DriveString := C+' The drive is a RAM disk.';
    end;
    { Only add drive types that can be determined. }
    if not ((DType = 0) or (DType = 1)) then
      lbDrives.Items.AddObject(DriveString, Pointer(i));
  end;

end;
```

Listing 12.13 is a simple routine that loops through all characters in the alphabet and passes them to the `GetDriveType()` function as root directories to determine whether they are valid drive types. If so, `GetDriveType()` will return which type of drive they are, which is determined by the `case` statement. A descriptive string is created and added to a list box along with the number representing the drive letter in the list box's `Objects` array. Only those drives that are valid are added to the list box. By the way, Delphi 5 does come with a `TDriveComboBox` component that enables you to select a drive. You'll find this on the Win 3.1 page of the Component Palette.

Obtaining Drive Information

In addition to determining the available drives and their types, you can obtain useful information on a particular drive. This information includes the following:

- Sectors per cluster
- Bytes per sector
- Number of free clusters
- Total number of clusters
- Total bytes of free disk space
- Total bytes of disk size

The first four items can be obtained by calling the `GetDiskFreeSpace()` Win32 API function. The last two items can be calculated from the information provided by `GetDiskFreeSpace()`. Listing 12.14 illustrates how you would use `GetDiskFreeSpace()`.

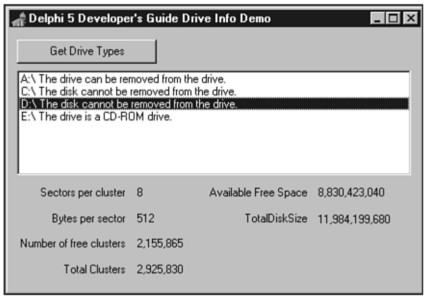
LISTING 12.14 Use of the `GetDiskFreeSpace()` Function

```
procedure TMainForm.lbDrivesClick(Sender: TObject);
var
  RootPath: String;          // Holds the drive root path
  SectorsPerCluster: DWord;  // Sectors per cluster
  BytesPerSector: DWord;     // Bytes per sector
  NumFreeClusters: DWord;    // Number of free clusters
  TotalClusters: DWord;      // Total clusters
  DriveByte: Byte;           // Drive byte value
  FreeSpace: Int64;          // Free space on drive
  TotalSpace: Int64;         // Total drive space.
  DriveNum:  Integer;        // Drive number 1 = A, 2 = B, etc.

begin
  with lbDrives do
  begin
    { Convert the ascii value for the drive letter to a valid drive number:
        1 = A, 2 = B, etc. by subtracting 64 from the ascii value. }
    DriveByte := Integer(Items.Objects[ItemIndex])-64;
    { First create the root path string }
    RootPath := chr(Integer(Items.Objects[ItemIndex]))+':\';
    { Call GetDiskFreeSpace to obtain the drive information }
    if GetDiskFreeSpace(PChar(RootPath), SectorsPerCluster,
      BytesPerSector, NumFreeClusters, TotalClusters) then
    begin
      { If this function is successful, then update the labels to
        display the disk information. }
      lblSectPerCluster.Caption := Format('%.0n', [SectorsPerCluster*1.0]);
      lblBytesPerSector.Caption := Format('%.0n', [BytesPerSector*1.0]);
      lblNumFreeClust.Caption := Format('%.0n', [NumFreeClusters*1.0]);
      lblTotalClusters.Caption := Format('%.0n', [TotalClusters*1.0]);
      // Obtain the available disk space
      FreeSpace  := DiskFree(DriveByte);
      TotalSpace := DiskSize(DriveByte);
      lblFreeSpace.Caption := Format('%.0n', [FreeSpace*1.0]);
      { Calculate the total disk space }
      lblTotalDiskSpace.Caption := Format('%.0n', [TotalSpace*1.0]);
    end
    else begin
      { Set labels to display nothing }
      lblSectPerCluster.Caption := 'X';
      lblBytesPerSector.Caption := 'X';
      lblNumFreeClust.Caption := 'X';
      lblTotalClusters.Caption := 'X';
      lblFreeSpace.Caption := 'X';
      lblTotalDiskSpace.Caption := 'X';
      ShowMessage('Cannot get disk info');
    end;
  end;

end;
```

Listing 12.14 is the `OnClick` event handler for a list box. In fact, a sample project illustrating both the `GetDriveType()` and `GetDiskFreeSpace()` functions exists on the CD as `DrvInfo.dpr`.

In Listing 12.14, when the user clicks one of the available items in `lbDrives`, a string representing the root directory for that drive is created and passed to the `GetDiskFreeSpace()` function. If the function is successful in determining the drive information, various labels on the form are updated to reflect that information. An example of the form for the sample project just mentioned is shown in Figure 12.3.

Note that you don't use the values returned from `GetDiskFreeSpace()` to determine the drive's size or its free space. Instead, you use the `DiskFree()` and `DiskSize()` functions that are defined in `SysUtils.pas`. The reason for this is that `GetDiskFreeSpace()` is flawed in Windows 95 in that it does not report drive sizes larger then 2GB, and it reports altered sector sizes for drives larger then 1GB. The `DiskSize()` and `DiskFree()` functions use a new Win32 API to obtain the information if it's available from the operating system.



```
A:\ The drive can be removed from the drive.
C:\ The disk cannot be removed from the drive.
D:\ The disk cannot be removed from the drive.
E:\ The drive is a CD-ROM drive.

       Sectors per cluster  8            Available Free Space  8,830,423,040
          Bytes per sector  512                  TotalDiskSize  11,984,199,680
Number of free clusters  2,155,865
            Total Clusters  2,925,830
```

FIGURE 12.3
The main form showing drive information for available drives.

Obtaining the Location of the Windows Directory

To obtain the location of the Windows directory, you must use the `GetWindowsDirectory()` Win32 API function. This function is defined as follows:

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
```

The first parameter is a null-terminated string buffer that will hold the Windows directory location. The second parameter indicates the size of the buffer. The following code fragment illustrates how you would use this function:

```
var
  WDir: String;
begin
  SetLength(WDir, 144);
  if GetWindowsDirectory(PChar(WDir), 144) <> 0 then
  begin
    SetLength(WDir, StrLen(PChar(WDir)));
    ShowMessage(WDir);
  end
  else
    RaiseLastWin32Error;
end;
```

Notice that because we used a long-string variable, we were able to typecast it as a `PChar` type. The `GetWindowsDirectory()` function returns an integer value representing the length of the directory path. Otherwise, it returns zero, indicating that an error occurred, in which case you must call `RaiseLastWin32Error` to determine the cause.

NOTE

You'll notice in the preceding code that we added the following line after the call to `GetWindowsDirectory()`:

```
SetLength(WDir, StrLen(PChar(WDir)));
```

Whenever you pass a long string to a function by first typecasting it as a `PChar`, Delphi doesn't know that the string has been manipulated and therefore cannot update its length information. You must explicitly do this by using the technique shown, which uses `StrLen()` to search for the null-terminator to determine the string's length. It then resizes the string through `SetLength()`.

Obtaining the Location of the System Directory

You can also obtain the location of the system directory by calling the `GetSystemDirectory()` Win32 API function. `GetSystemDirectory()` works just like `GetWindowsDirectory()` except that it returns the full path to the Windows system directory as opposed to the Windows directory. The following code fragment illustrates how you would use this function:

```
var
  SDir: String;
begin
  SetLength(SDir, 144);
  if GetSystemDirectory(PChar(SDir), 144) <> 0 then
  begin
    SetLength(SDir, StrLen(PChar(SDir)));
    ShowMessage(SDir);
  end
  else
    RaiseLastWin32Error;
end;
```

The return value of this function represents the same values from the `GetWindowsDirectory()` function.

Obtaining the Name of the Current Directory

Often, you need to obtain the current directory (that is, the directory from which your application was executed). To do this, you call the `GetCurrentDirectory()` Win32 API function. If you guess that the `GetCurrentDirectory()` operates just like the last two functions mentioned, you're absolutely right (well, sort of). There's one slight catch—the parameters are reversed. The following code fragment illustrates the use of this function:

```
var
  CDir: String;
begin
  SetLength(CDir, 144);
  if GetCurrentDirectory(144, PChar(CDir)) <> 0 then
  begin
    SetLength(CDir, StrLen(PChar(CDir)));
    ShowMessage(CDir);
  end
  else
    RaiseLastWin32Error;
end;
```

NOTE

Delphi provides the functions `CurDir()` and `ChDir()` in the `System` unit as well as the `GetCurrentDir()` and `SetCurrentDir()` functions in `SysUtils.pas`.

NOTE

Delphi comes with its own set of routines to obtain directory information on a given file. For example, the `TApplication.ExeName` property holds the full path and filename for the running process. Assuming that this parameter holds the value `"C:\Delphi\Bin\Project.exe"`, Table 12.7 shows the values returned from the various Delphi functions when passing the `TApplication.ExeName` property.

TABLE 12.7 Delphi File/Directory Information Function

Function	*Result of Passing* `"C:\Delphi\Bin\Project.exe"`
`ExtractFileDir()`	`C:\Delphi\Bin`
`ExtractFileDrive()`	`C:`
`ExtractFileExt()`	`.exe`
`ExtractFileName()`	`Project1.exe`
`ExtractFilePath()`	`C:\Delphi\Bin\`

Searching for a File Across Directories

You might at some time need to search for or perform some process on files, given a file mask across a directory and its subdirectories. Listing 12.15 illustrates how you can do this using a procedure that gets called recursively so that the subdirectories can be searched as well as the current directory. This demo exists on the CD as `DirSrch.dpr`.

NOTE

You can use the Win32 API function `SearchPath()` to search across a specified directory, the system directories, directories in the environment variable `PATH`, or a semicolon-delimited list of directories. This function doesn't search across subdirectories of a given directory, however.

LISTING 12.15 Example of Searching Across Directories to Perform a File Search

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, FileCtrl, Grids, Outline, DirOutln;

type
  TMainForm = class(TForm)
    dcbDrives: TDriveComboBox;
    edtFileMask: TEdit;
    lblFileMask: TLabel;
    btnSearchForFiles: TButton;
    lbFiles: TListBox;
    dolDirectories: TDirectoryOutline;
    procedure btnSearchForFilesClick(Sender: TObject);
    procedure dcbDrivesChange(Sender: TObject);
  private
    FFileName: String;
    function GetDirectoryName(Dir: String): String;
    procedure FindFiles(APath: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

function TMainForm.GetDirectoryName(Dir: String): String;
{ This function formats the directory name so that it is a valid
  directory containing the back-slash (\) as the last character. }
begin
  if Dir[Length(Dir)]<> '\' then
    Result := Dir+'\'
  else
    Result := Dir;
end;

procedure TMainForm.FindFiles(APath: String);
{ This is a procedure which is called recursively so that it finds the
  file with a specified mask through the current directory and its
  sub-directories. }
var
  FSearchRec,
  DSearchRec: TSearchRec;
  FindResult: integer;

  function IsDirNotation(ADirName: String): Boolean;
  begin
```

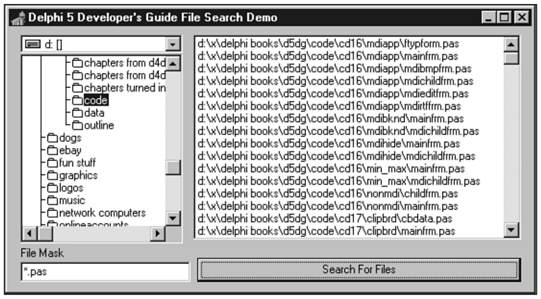
continues

LISTING 12.15 Continued

```
    Result := (ADirName = '.') or (ADirName = '..');
  end;

begin
  APath := GetDirectoryName(APath); // Obtain a valid directory name
  { Find the first occurrence of the specified file name }
  FindResult := FindFirst(APath+FFileName,faAnyFile+faHidden+
                          faSysFile+faReadOnly,FSearchRec);
  try
    { Continue to search for the files according to the specified
      mask. If found, add the files and their paths to the listbox.}
    while FindResult = 0 do
    begin
      lbFiles.Items.Add(LowerCase(APath+FSearchRec.Name));
      FindResult := FindNext(FSearchRec);
    end;

    { Now search the sub-directories of this current directory. Do this
      by using FindFirst to loop through each subdirectory, then call
      FindFiles (this function) again. This recursive process will
      continue until all sub-directories have been searched. }
    FindResult := FindFirst(APath+'*.*', faDirectory, DSearchRec);

    while FindResult = 0 do
    begin
      if ((DSearchRec.Attr and faDirectory) = faDirectory) and not
        IsDirNotation(DSearchRec.Name) then
        FindFiles(APath+DSearchRec.Name); // Recursion here
      FindResult := FindNext(DSearchRec);
    end;
  finally
    FindClose(FSearchRec);
  end;
end;

procedure TMainForm.btnSearchForFilesClick(Sender: TObject);
{ This method starts the searching process. It first changes the cursor
  to an hourglass since the process may take awhile. It then clears the
  listbox and calls the FindFiles() function which will be called
  recursively to search through sub-directories }
begin
  Screen.Cursor := crHourGlass;
  try
    lbFiles.Items.Clear;
    FFileName := edtFileMask.Text;
    FindFiles(dolDirectories.Directory);
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TMainForm.dcbDrivesChange(Sender: TObject);
begin
  dolDirectories.Drive := dcbDrives.Drive;
end;

end.
```

In the `FindFiles()` method, the first `while..do` construct searches for files in the current directory specified by the `APath` parameter and then adds the files and their paths to `lbFiles`. The second `while..do` construct finds the subdirectories in the current directory and appends them to the `APath` variable. The `FindFiles()` method then passes `APath`, now with a subdirectory name, to itself, resulting in a recursive call. This process continues until all subdirectories have been searched through.

Figure 12.4 shows the results of a file search for all PAS files in the Delphi 5 Code directory.



```
d:\x\delphi books\d5dg\code\cd16\mdiapp\ftypform.pas
d:\x\delphi books\d5dg\code\cd16\mdiapp\mainfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdiapp\mdibmpfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdiapp\mdichildfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdiapp\mdieditfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdiapp\mdirtffrm.pas
d:\x\delphi books\d5dg\code\cd16\mdibknd\mainfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdibknd\mdichildfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdihide\mainfrm.pas
d:\x\delphi books\d5dg\code\cd16\mdihide\mdichildfrm.pas
d:\x\delphi books\d5dg\code\cd16\min_max\mainfrm.pas
d:\x\delphi books\d5dg\code\cd16\min_max\mdichildfrm.pas
d:\x\delphi books\d5dg\code\cd16\nonmdi\childfrm.pas
d:\x\delphi books\d5dg\code\cd16\nonmdi\mainfrm.pas
d:\x\delphi books\d5dg\code\cd17\clipbrd\cbdata.pas
d:\x\delphi books\d5dg\code\cd17\clipbrd\mainfrm.pas
```

File Mask: *.pas

FIGURE 12.4
The result of a file search across directories.

Two Object Pascal structures and two functions merit mention here. First, we'll talk about the `TSearchRec` structure and the `FindFirst()` and `FindNext()` functions. Then, we'll discuss the `TWin32FindData` structure.

Copying and Deleting a Directory Tree

Before Win32, you were required to parse a directory tree and use the `FindFirst()`/`FindNext()` pairs to copy a directory to another location. Now you can use the

ShFileOperation() Win32 function, which greatly simplifies the process. The following code illustrates a function that uses the ShFileOperation() API to perform a directory copy operation. This function is well documented in the Win32 online help, so we won't repeat that information here. Instead, we suggest that you give it a readthrough. Note the inclusion of the ShellAPI unit in the uses clause. Here's the code:

```
uses
  ShellAPI;

procedure CopyDirectoryTree(AHandle: THandle; AFromDir, AToDir: String);
var
  SHFileOpStruct: TSHFileOpStruct;
Begin
  with SHFileOpStruct do
  begin
    Wnd     := AHandle;
    wFunc   := FO_COPY;
    pFrom   := PChar(AFromDir);
    pTo     := PChar(AToDir);
    fFlags  := FOF_NOCONFIRMATION or FOF_RENAMEONCOLLISION;
    fAnyOperationsAborted   := False;
    hNameMappings           := nil;
    lpszProgressTitle       := nil;
  end;
  ShFileOperation(SHFileOpStruct);
end;
```

The ShFileOperation() function can also be used to move a directory to the Recycle Bin, as illustrated here:

```
uses ShellAPI;

procedure ToRecycle(AHandle: THandle; AFileName: STring);
var
  SHFileOpStruct: TSHFileOpStruct;
begin
  with SHFileOpStruct do
  begin
    Wnd    := AHandle;
    wFunc  := FO_DELETE;
    pFrom  := PChar(AFileName);
    fFlags := FOF_ALLOWUNDO;
  end;
  SHFileOperation(SHFileOpStruct);
end;
```

We will discuss the SHFileOperation() in greater detail later in this chapter.

The TSearchRec Record

The TSearchRec record defines data returned by the FindFirst() and FindNext() functions. Object Pascal defines this record as the following:

```
TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
end;
```

TSearchRec's fields are modified by the aforementioned functions when the file is found.

The Time field contains the file time of creation or modification. The Size field contains the size of the file in bytes. The Name field holds the name of the file. The Attr field contains one or more of the file attributes shown in Table 12.8.

TABLE 12.8 File Attributes

Attribute	*Value*	*Description*
faReadOnly	$01	Read-only file
faHidden	$02	Hidden file
faSysFile	$04	System file
faVolumeID	$08	Volume ID file
faDirectory	$10	Directory
faArchive	$20	Archive file
faAnyFile	$3F	Any file

The FindHandle and ExcludeAttr fields are used internally by FindFirst() and FindNext(). You need not concern yourself with these fields.

Both FindFirst() and FindNext() take a path as a parameter that can contain wildcard characters—for example, C:\DELPHI 5\BIN\*.EXE means all files with an .EXE extension in the C:\DELPHI 5\BIN\ directory. The Attr parameter specifies the file attributes on which to search. Suppose that you want to search on system files only; you would invoke FindFirst() and/or FindNext(), as in this code:

```
FindFirst(Path, faSysFile, SearchRec);
```

The TWin32FindData Record

The `TWin32FindData` record contains information about the found file or subdirectory. This record is defined as follows:

```
TWin32FindData = record
    dwFileAttributes: DWORD;
    ftCreationTime: TFileTime;
    ftLastAccessTime: TFileTime;
    ftLastWriteTime: TFileTime;
    nFileSizeHigh: DWORD;
    nFileSizeLow: DWORD;
    dwReserved0: DWORD;
    dwReserved1: DWORD;
    cFileName: array[0..MAX_PATH - 1] of AnsiChar;
    cAlternateFileName: array[0..13] of AnsiChar;
  end;
```

Table 12.9 shows the meaning of `TWin32FindData`'s fields.

TABLE 12.9 `TWin32FindData` Field Meanings

Field	*Meaning*
`dwFileAttributes`	The file attributes for the found file. See the online help under `WIN32_FIND_DATA` for more information.
`FtCreationTime`	The time the file was created.
`FtLastAccessTime`	The time the file was last accessed.
`FtLastWriteTime`	The time the file was last modified.
`NFileSizeHigh`	The high-order `DWORD` of the file size in bytes. This value is zero unless the file is larger than `MAXDWORD`.
`NFileSizeLow`	The low-order `DWORD` of the file size in bytes.
`DwReserved0`	Not currently used (reserved).
`DwReserved1`	Not currently used (reserved).
`CFileName`	Null-terminated filename.
`CAlternateFileName`	An 8.3 formatted name, a truncation of the long filename.

Getting File Version Information

It's possible to extract version information from EXE and DLL files that contain the version information resource. In the following sections, you create a class that encapsulates the functionality to extract the version information resource, and you use that class in a sample project.

Defining the TVerInfoRes Class

The `TVerInfoRes` class encapsulates three Win32 API functions for extracting version information from files that contain version information. These functions are `GetFileVersionInfoSize()`, `GetFileVersionInfo()` and `VerQueryValue()`. Version information on a file may include data such as company name, file description, version, and comments, just to name a few. The data that `TVerInfoRes` retrieves is as follows:

- *Company name*. The name of the company that created the file
- *Comments*. Any additional comments that may be attached to the file
- *File description*. A description of the file
- *File version*. A version number
- *Internal name*. An internal name as defined by the company generating the file
- *Legal copyright*. All copyright notices that apply to the file
- *Legal trademarks*. Legal trademarks that apply to the file
- *Original filename*. The original filename (if any)

The unit that defines the `TVerInfoRes` class, `VERINFO.PAS`, is shown in Listing 12.16.

LISTING 12.16 The Source Code for `VERINFO.PAS`, the `TVerInfoRes` Class Definition

```
unit VerInfo;

interface

uses SysUtils, WinTypes, Dialogs, Classes;

type
  { define a generic exception class for version info, and an exception
    to indicate that no version info is available. }
  EVerInfoError   = class(Exception);
  ENoVerInfoError = class(Exception);
  eNoFixeVerInfo  = class(Exception);

  // define enum type representing different types of version info
  TVerInfoType =
    (viCompanyName,
     viFileDescription,
     viFileVersion,
     viInternalName,
     viLegalCopyright,
     viLegalTrademarks,
     viOriginalFilename,
```

continues

LISTING 12.16 Continued

```
     viProductName,
     viProductVersion,
     viComments);

const

  // define an array constant of strings representing the pre-defined
  // version information keys.
  VerNameArray: array[viCompanyName..viComments] of String[20] =
  ('CompanyName',
   'FileDescription',
   'FileVersion',
   'InternalName',
   'LegalCopyright',
   'LegalTrademarks',
   'OriginalFilename',
   'ProductName',
   'ProductVersion',
   'Comments');

type

  // Define the version info class
  TVerInfoRes = class
  private
    Handle            : DWord;
    Size              : Integer;
    RezBuffer         : String;
    TransTable        : PLongint;
    FixedFileInfoBuf  : PVSFixedFileInfo;
    FFileFlags        : TStringList;
    FFileName         : String;
    procedure FillFixedFileInfoBuf;
    procedure FillFileVersionInfo;
    procedure FillFileMaskInfo;
  protected
    function GetFileVersion   : String;
    function GetProductVersion: String;
    function GetFileOS        : String;
  public
    constructor Create(AFileName: String);
    destructor Destroy; override;
    function GetPreDefKeyString(AVerKind: TVerInfoType): String;
    function GetUserDefKeyString(AKey: String): String;
    property FileVersion    : String read GetFileVersion;
    property ProductVersion : String read GetProductVersion;
    property FileFlags      : TStringList read FFileFlags;
    property FileOS         : String read GetFileOS;
  end;

implementation

uses Windows;

const
  // strings that must be fed to VerQueryValue() function
  SFInfo                = '\StringFileInfo\';
  VerTranslation: PChar = '\VarFileInfo\Translation';
  FormatStr             = '%s%.4x%.4x\%s%s';


constructor TVerInfoRes.Create(AFileName: String);
begin
  FFileName := aFileName;
  FFileFlags := TStringList.Create;
  // Get the file version information
  FillFileVersionInfo;
  //Get the fixed file info
  FillFixedFileInfoBuf;
  // Get the file mask values
  FillFileMaskInfo;
end;


destructor TVerInfoRes.Destroy;
begin
  FFileFlags.Free;
end;

procedure TVerInfoRes.FillFileVersionInfo;
var
  SBSize: UInt;
begin
  // Determine size of version information
  Size := GetFileVersionInfoSize(PChar(FFileName), Handle);
  if Size <= 0 then         { raise exception if size <= 0 }
    raise ENoVerInfoError.Create('No Version Info Available.');

  // Set the length accordingly
  SetLength(RezBuffer, Size);
  // Fill the buffer with version information, raise exception on error
```

continues

LISTING 12.16 Continued

```
  if not GetFileVersionInfo(PChar(FFileName), Handle, Size,
➥PChar(RezBuffer)) then
    raise EVerInfoError.Create('Cannot obtain version info.');

  // Get translation info, raise exception on error
  if not VerQueryValue(PChar(RezBuffer), VerTranslation,  pointer(TransTable),
  SBSize) then
    raise EVerInfoError.Create('No language info.');
end;

procedure TVerInfoRes.FillFixedFileInfoBuf;
var
  Size: Longint;
begin
  if VerQueryValue(PChar(RezBuffer), '\', pointer(FixedFileInfoBuf),
➥Size) then begin
     if Size < SizeOf(TVSFixedFileInfo) then
        raise eNoFixeVerInfo.Create('No fixed file info');
  end
  else
    raise eNoFixeVerInfo.Create('No fixed file info')
end;

procedure TVerInfoRes.FillFileMaskInfo;
begin
  with FixedFileInfoBuf^ do begin
    if (dwFileFlagsMask and dwFileFlags and VS_FF_PRERELEASE) <> 0then
      FFileFlags.Add('Pre-release');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_PRIVATEBUILD) <> 0 then
      FFileFlags.Add('Private build');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_SPECIALBUILD) <> 0 then
      FFileFlags.Add('Special build');
    if (dwFileFlagsMask and dwFileFlags and VS_FF_DEBUG) <> 0 then
      FFileFlags.Add('Debug');
  end;
end;

function TVerInfoRes.GetPreDefKeyString(AVerKind: TVerInfoType): String;
var
  P: PChar;
  S: UInt;
begin
  Result := Format(FormatStr, [SfInfo, LoWord(TransTable^),HiWord(TransTable^),
    VerNameArray[aVerKind], #0]);
  // get and return version query info, return empty string on error
  if VerQueryValue(PChar(RezBuffer), @Result[1], Pointer(P), S) then
    Result := StrPas(P)
  else
    Result := '';
end;

function TVerInfoRes.GetUserDefKeyString(AKey: String): String;
var
  P: Pchar;
  S: UInt;
begin
  Result := Format(FormatStr, [SfInfo, LoWord(TransTable^),HiWord(TransTable^),
    aKey, #0]);
  // get and return version query info, return empty string on error
  if VerQueryValue(PChar(RezBuffer), @Result[1], Pointer(P), S) then
    Result := StrPas(P)
  else
    Result := '';
end;


function VersionString(Ms, Ls: Longint): String;
begin
  Result := Format('%d.%d.%d.%d', [HIWORD(Ms), LOWORD(Ms),
     HIWORD(Ls), LOWORD(Ls)]);
end;

function TVerInfoRes.GetFileVersion: String;
begin
  with FixedFileInfoBuf^ do
    Result := VersionString(dwFileVersionMS, dwFileVersionLS);
end;

function TVerInfoRes.GetProductVersion: String;
begin
  with FixedFileInfoBuf^ do
    Result := VersionString(dwProductVersionMS, dwProductVersionLS);
end;

function TVerInfoRes.GetFileOS: String;
begin
  with FixedFileInfoBuf^ do
    case dwFileOS of
      VOS_UNKNOWN:  // Same as VOS__BASE
        Result := 'Unknown';
      VOS_DOS:
```

continues

LISTING 12.16 Continued

```
        Result := 'Designed for MS-DOS';
      VOS_OS216:
        Result := 'Designed for 16-bit OS/2';
      VOS_OS232:
        Result := 'Designed for 32-bit OS/2';
      VOS_NT:
        Result := 'Designed for Windows NT';


      VOS__WINDOWS16:
        Result := 'Designed for 16-bit Windows';
      VOS__PM16:
        Result := 'Designed for 16-bit PM';
      VOS__PM32:
        Result := 'Designed for 32-bit PM';
      VOS__WINDOWS32:
        Result := 'Designed for 32-bit Windows';

      VOS_DOS_WINDOWS16:
        Result := 'Designed for 16-bit Windows, running on MS-DOS';
      VOS_DOS_WINDOWS32:
        Result := 'Designed for Win32 API, running on MS-DOS';
      VOS_OS216_PM16:
        Result := 'Designed for 16-bit PM, running on 16-bit OS/2';
      VOS_OS232_PM32:
        Result := 'Designed for 32-bit PM, running on 32-bit OS/2';
      VOS_NT_WINDOWS32:
        Result := 'Designed for Win32 API, running on Windows/NT';
    else
      Result := 'Unknown';
    end;
end;

end.
```

`TVerInfoRes` contains the required fields and encapsulates the appropriate Win32 API routines to obtain version information from any file. The file from which the version information is to be obtained is specified by passing the filename as `AFileName` to the `TVerInfoRes.Create()` constructor. This filename is assigned to the field `FFileName`, which is used in another routine to actually extract the version information. The constructor then calls three methods, `FillFileVersionInfo()`, `FillFixedFileInfoBuf()`, and `FillFileMaskInfo()`.

The FillFileVersionInfo() Method

The `FillFileVersionInfo()` method performs the initial work of loading the version information before you can start to examine the version information specifics. This method first determines whether the file even has version information and, if so, its size. The size is necessary to determine how much memory to allocate to hold this information when it's retrieved. The Win32 API function `GetFileVersionInfoSize()` determines the size of the version information contained in a file. This function is declared as follows:

```
function GetFileVersionInfoSize(lptstrFilename: PChar;
var lpdwHandle: DWORD): DWORD; stdcall;
```

The `lptstrFileName` parameter refers to the file from which the version information is to be obtained. The `lpdwHandle` parameter is a `DWORD` variable that's set to zero when the function is called. As far as we can determine, this variable serves no other purpose.

`FillFileVersionInfo()` passes `FFileName` to `GetFileVersionInfoSize()`; if the return value, stored in the `Size` variable, is greater than zero, a buffer, `RezBuffer`, is allocated to store `Size` bytes.

After memory for `RezBuffer` has been allocated, it's passed to the function `GetFileVersionInfo()`, which actually fills `RezBuffer` with the version information. `GetFileVersionInfo()` is declared as follows:

```
function GetFileVersionInfo(lptstrFilename: PChar; dwHandle,
dwLen: DWORD; lpData: Pointer): BOOL; stdcall;
```

The `lptstrFileName` parameter takes the filename of the file, `FFileName`. `DwHandle` is ignored. `DwLen` is the return value from `GetFileVersionInfoSize()`, which was stored in the variable `Size`. `LpData` is a pointer to the buffer that holds the version information. If `GetFileVersionInfo()` does not succeed in retrieving the version information, it returns `False`; otherwise, `True` is returned.

Finally, the `FillFileVersionInfo()` method calls the API function `VerQueryValue()`, which is used to return selected version information from the version information resource. In this instance, `VerQueryValue()` is called to retrieve a pointer to the language and character set identifier array. This array is used in subsequent calls to `VerQueryValue()` to access version information in the language-specific `StringTable` in the version information resource.

`VerQueryValue()` is declared as follows:

```
function VerQueryValue(pBlock: Pointer; lpSubBlock: PChar;
var lplpBuffer: Pointer; var puLen: UINT): BOOL; stdcall;
```

The parameter `pBlock` refers to the `lpData` parameter, which was passed to `GetFileVersionInfo()`. `LpSubBlock` is a null-terminated string that specifies which version information value to retrieve. You might take a look at the online help for `VerQueryValue()`, which describes the various strings that can be passed to `VerQueryValue()`. In the case of the preceding example, the string `"\VarFileInfo\Translation"` is passed as the `lpSubBlock` parameter to retrieve the language and character set translation information. The `lplpBuffer` parameter points to the buffer that holds the version information value. The `puLen` parameter contains the length of the data retrieved.

The FillFixedFileInfoBuf() Method

The `FillFixedFileInfoBuf()` method illustrates how to use `VerQueryValue()` to obtain a pointer to the `VS_FIXEDFILEINFO` structure, which contains the version information about the file. This is done by passing the string `"\"` as the `lpSubBlock` parameter to the `VerQueryValue()` function. This pointer is stored in the `TVerInfoRes.FixedFileInfoBuf` field.

The FillFileMaskInfo() Method

The `FillFileMaskInfo()` method illustrates how to obtain module attributes. This is handled by performing the appropriate bitmask operation on the `dwFileFlagsMask` and `dwFileFlags` fields of `FixedFileInfoBuf` as well as on the specific flag being evaluated. We won't get into the specifics as to the meaning of these flags. If you're interested, the online help for the Version Info page of the Project Options dialog box explains this in detail.

The GetPreDefKeyString() and GetUserDefKeyString() Methods

The `GetPreDefKeyString()` and `GetUserDefKeyString()` methods illustrate how to use the `VerQueryValue()` function to retrieve the version information strings that are entered into the `Key` table on the Version Info page of the Project Options dialog box. By default, the Win32 API provides 10 predefined strings that we've placed into the `VerNameArray` constant. To retrieve a specific string, you must pass as the `lpSubBlock` parameter of `VerQueryValue()` the string `"\StringFileInfo\`*`lang-charset`*`\`*`string-name`*`"`. The *`lang-charset`* string refers to the language and character set identifier previously retrieved in the `FillFileVersionInfo()` method and referred to by the `TransTable` field. The *`string-name`* string refers to one of the predefined string constants in `VerNameArray`. `GetPreDefKeyString()` handles retrieving the predefined version information strings.

`GetUserDefKeyString()` is similar in functionality to `GetPreDefKeyString()` except that the key string must be passed in as a parameter. The value of the `lpSubBlock` string is constructed in this method, using the `AKey` parameter as the key.

Getting the Version Numbers

The `GetFileVersion()` and `GetProductVersion()` methods illustrate how to obtain the file and product version numbers for a file.

The FixedFileInfoBuf structure contains fields that refer to the version number of the file itself as well as the version number of the product with which the file may be distributed. These version numbers are stored in a 64-bit number. The most significant and least significant 32 bits are retrieved separately by using different fields.

The file's binary version number is stored in the fields dwFileVersionMS and dwFileVersionLS. The version number for the product with which a file is distributed is stored in the dwProductVersionMS and dwProductVersionLS fields.

The GetFileVersion() and GetProductVersion() methods return a string representing the version number for a given file. They both use a helper function, VersionString(), to properly format the string.

Getting the Operating System Information

The GetFileOS() method illustrates how to determine for which operating system the file was designed. This is accomplished by examining the dwFileOS field of the FixedFileInfoBuf structure. For more information on the meaning of the various values that can be assigned to dwFileOS, examine the online API help for VS_FIXEDFILEINFO.

Using the TVerInfoRes Class

We created the project VerInfo.dpr to illustrate the use of the TVerInfoRes class. Listing 12.17 shows the source for this project's main form.

LISTING 12.17 The Source Code for the Version Information Demo Main Form

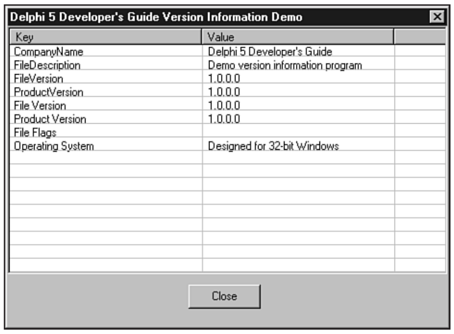
```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, FileCtrl, StdCtrls, verinfo, Grids, Outline, DirOutln,
  ComCtrls;

type
  TMainForm = class(TForm)
    lvVersionInfo: TListView;
    btnClose: TButton;
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
```

continues

LISTING 12.17 Continued

```
  private
    VerInfoRes: TVerInfoRes;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure AddListViewItem(const aCaption, aValue: String; aData: Pointer;
  aLV: TListView);
// This method is used to add a TListItem to the TListView, aLV
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.Data := aData;
  NewItem.SubItems.Add(aValue);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  VerInfoRes := TVerInfoRes.Create(Application.ExeName);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  VerInfoRes.Free;
end;

procedure TMainForm.FormShow(Sender: TObject);
var
  VerString: String;
  i: integer;
  sFFlags: String;


begin
  for i := ord(viCompanyName) to ord(viComments) do begin
    VerString := VerInfoRes.GetPreDefKeyString(TVerInfoType(i));
    if VerString <> '' then
      AddListViewItem(VerNameArray[TVerInfoType(i)], VerString, nil,
        lvVersionInfo);
  end;
  VerString := VerInfoRes.GetUserDefKeyString('Author');
  if VerString <> EmptyStr then
      AddListViewItem('Author', VerString, nil, lvVersionInfo);


  AddListViewItem('File Version', VerInfoRes.FileVersion, nil,
    lvVersionInfo);
  AddListViewItem('Product Version', VerInfoRes.ProductVersion, nil,
    lvVersionInfo);
  for i := 0 to VerInfoRes.FileFlags.Count - 1 do begin
    if i <> 0 then
      sFFlags := SFFlags+', ';
    sFFlags := SFFlags+VerInfoRes.FileFlags[i];
  end;
  AddListViewItem('File Flags',SFFlags, nil, lvVersionInfo);
  AddListViewItem('Operating System', VerINfoRes.FileOS, nil, lvVersionInfo);

end;

procedure TMainForm.btnCloseClick(Sender: TObject);
begin
  Close;
end;

end.
```

The version information demo is straightforward. It simply displays the version information for itself. Figure 12.5 shows the project running and displaying this information.



Key	Value
CompanyName	Delphi 5 Developer's Guide
FileDescription	Demo version information program
FileVersion	1.0.0.0
ProductVersion	1.0.0.0
File Version	1.0.0.0
Product Version	1.0.0.0
File Flags	
Operating System	Designed for 32-bit Windows

FIGURE 12.5
Version information for demo application.

Using the SHFileOperation() Function

A very useful Windows API function is `SHFileOperation()`. This function uses a `SHFILEOP-STRUCT` structure to perform copy, move, rename, or delete operations on any file system object, such as files and directories. The Win32 API help file documents this structure well, so we won't repeat that information here. We will, however, show a few useful and frequently requested techniques on using this function to copy an entire directory to another location and to delete a file so that it's placed into the Windows Recycle Bin.

Copying a Directory

Listing 12.18 is a procedure we wrote to copy a directory tree from one location to another.

LISTING 12.18 The `CopyDirectoryTree()` Procedure

```
procedure CopyDirectoryTree(AHandle: THandle;
  const AFromDirectory, AToDirectory: String);
var
  SHFileOpStruct: TSHFileOpStruct;
  FromDir: PChar;
  ToDir: PChar;
begin

  GetMem(FromDir, Length(AFromDirectory)+2);
  try
    GetMem(ToDir, Length(AToDirectory)+2);
    try

      FillChar(FromDir^, Length(AFromDirectory)+2, 0);
      FillChar(ToDir^, Length(AToDirectory)+2, 0);

      StrCopy(FromDir, PChar(AFromDirectory));
      StrCopy(ToDir, PChar(AToDirectory));

      with SHFileOpStruct do
      begin
        Wnd    := AHandle;   // Assign the window handle
        wFunc  := FO_COPY;  // Specify a file copy
        pFrom  := FromDir;
        pTo    := ToDir;
        fFlags := FOF_NOCONFIRMATION or FOF_RENAMEONCOLLISION;
        fAnyOperationsAborted := False;
        hNameMappings := nil;
        lpszProgressTitle := nil;
        if SHFileOperation(SHFileOpStruct) <> 0 then
          RaiseLastWin32Error;
      end;
    finally
      FreeMem(ToDir, Length(AToDirectory)+2);
    end;
  finally
    FreeMem(FromDir, Length(AFromDirectory)+2);
  end;
end;
```

The `CopyDirectoryTree()` procedure takes three parameters. The first, `AHandle`, is the handle of a dialog box owner that would display any status information about the file operation. The remaining two parameters are the source and destination directory locations. Since Windows API functions work with `PChars`, we simply copy these two locations into `PChar` variables after we allocate memory for the `PChars`. Then, we assign these values to the `pFrom` and `pTo` members of the `SHFileOpStruct` structure. Note the assignment to the `wFunc` member as `FO_COPY`. This is what instructs `SHFileOperation` of the type of operation to perform. The remaining members are explained in the online help. On the call to `SHFileOperation()`, the source directory would be moved to the destination specified by the `AToDirectory` parameter.

Moving Files and Directories to the Recycle Bin.

Listing 12.19 shows a similar technique to that preceding listing, except that this shows how you might move a file to the Windows Recycle Bin.

Listing 12.19 The `ToRecycle()` Procedure

```
procedure ToRecycle(AHandle: THandle; const ADirName: String);
var
  SHFileOpStruct: TSHFileOpStruct;
  DirName: PChar;
  BufferSize: Cardinal;
begin
  BufferSize := Length(ADirName) +1 +1;
  GetMem(DirName, BufferSize);
  try
    FillChar(DirName^, BufferSize, 0);
    StrCopy(DirName, PChar(ADirName));

    with SHFileOpStruct do
    begin
      Wnd := AHandle;
      wFunc := FO_DELETE;
      pFrom := DirName;
      pTo := nil;
```

continues

LISTING 12.19 Continued

```
      fFlags := FOF_ALLOWUNDO;

      fAnyOperationsAborted := False;
      hNameMappings := nil;
      lpszProgressTitle := nil;
    end;

    if SHFileOperation(SHFileOpStruct) <> 0 then
      RaiseLastWin32Error;
  finally
    FreeMem(DirName, BufferSize);
  end;
end;
```

You'll notice that there's not much of a difference between this procedure and the previous except that the `wFunc` member is assigned `FO_DELETE` and the `pTo` member is set to `nil`. The `pTo` member is ignored by the `SHFileOperation()` function on a delete operation. Also, because the `FOF_ALLOWUNDO` flag is added to the `fFlags` member, the function will move the file to the Recycle Bin to allow for undoing the operation.

Examples of both of these operations are included on the CD in the `SHFileOp.dpr` project.

Summary

This chapter gave you a substantial amount of information on working with files, directories, and drives. You learned how to manipulate different file types. The chapter illustrated the technique of descending from Delphi's `TFileStream` class to encapsulate record-file I/O. It even showed you how to use Win32's memory-mapped files. You created a `TMemMapFile` class to encapsulate the memory-mapped functionality. Finally, the chapter demonstrated how to retrieve version information from a file containing such information.