Глава 12

Работа с файлами

В этой главе…

Работа с файлами, каталогами и устройствами является обычной задачей программирования, с которой рано или поздно вам обязательно придется столкнуться. В этой главе рассматриваются способы работы с различными типами файлов: текстовыми, типизированными и нетипизированными. Кроме того, вы узнаете, как используется для инкапсуляции процессов ввода-вывода файлов класс `TFileStream` и как воспользоваться преимуществами *файлов, отображенных в память*, — одного из самых мощных средств Win32. Вы также узнаете, как создать класс `TMemoryMappedFile`, который инкапсулирует некоторые возможности отображения в память, и научитесь использовать этот класс для выполнения поиска заданного текста в текстовых файлах. В этой главе также демонстрируются эффективные методы определения доступных устройств, исследования деревьев каталогов для обнаружения нужных файлов и получения информации о версиях файлов. К концу главы у вас появится уверенность в себе, поскольку полученных знаний будет вполне достаточно для успешной работы с файлами, каталогами и устройствами памяти.

Файловые операции ввода-вывода

Как программисту, вам придется иметь дело с тремя типами файлов: текстовыми, типизированными и двоичными. В следующих нескольких разделах рассматривается ввод-вывод файлов этих типов. *Текстовые файлы*, как следует из их названия, содержат текст ASCII, который может прочитать любой текстовый редактор. *Типизированные файлы* включают данные, тип которых определяется программистом. Наконец, *двоичные файлы* охватывают множество остальных файлов. Под этим обтекаемым названием может скрываться любой файл, содержащий данные, представленные в любом формате или вовсе неформатированные.

Работа с текстовыми файлами

В этом разделе показаны примеры манипуляции текстовыми файлами с помощью процедур и функций, построенных на основе библиотеки времени исполнения Object Pascal. Прежде чем что-либо делать с текстовым файлом, его необходимо открыть, но сначала нужно объявить переменную типа `TextFile`:

```
var MyTextFile: TextFile;
```

Теперь эту переменную можно использовать для ссылки на любой текстовый файл.

Чтобы открыть текстовый файл, нужно знать о существовании двух процедур. Первая называется `AssignFile()` и связывает имя файла с файловой переменной:

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

После того как вы свяжете файловую переменную с именем файла, можно открыть сам файл. В случае текстового файла это можно сделать тремя способами. Во-первых, создать и открыть файл можно с помощью процедуры `Rewrite()`. Если применить эту процедуру к существующему файлу, он будет перезаписан, т.е. с тем же именем будет создан новый файл. Во-вторых, можно открыть файл с доступом только для чтения — для этого потребуется вызвать процедуру `Reset()`. И в-третьих, для добавления информации в конец существующего файла вам не обойтись без процедуры `Append()`.

На заметку Процедура `Reset()` открывает как типизированные, так и нетипизированные файлы с доступом только для чтения.

Чтобы закрыть файл после его открытия используйте процедуру `CloseFile()`. Рассмотрим пример, в котором иллюстрируется каждая процедура.

Для открытия файла с доступом только для чтения используйте последовательность инструкций:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Reset(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Для создания нового файла выполните следующее:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Rewrite(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

Для добавления данных в конец существующего файла используйте такую последовательность инструкций:

```
var
  MyTextFile: TextFile;
begin
  AssignFile(MyTextFile, 'MyTextFile.txt');
  Append(MyTextFile);
  try
    { Работа с файлом }
  finally
    CloseFile(MyTextFile);
  end;
end;
```

В листинге 12.1 продемонстрирована возможность использования процедуры `Rewrite()` для создания текстового файла и добавления в него пяти строк текста.

Листинг 12.1. Создание текстового файла

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

После выполнения этой программы файл `MyTextFile.txt` будет содержать следующий текст:

```
Это строка # 1
Это строка # 2
Это строка # 3
Это строка # 4
Это строка # 5
```

Листинг 12.2 иллюстрирует один из способов добавления еще пяти строк в тот же файл.

Листинг 12.2. Добавление строк в конец текстового файла

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

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

Теперь содержимое этого файла выглядит следующим образом:

```
Это строка # 1
Это строка # 2
Это строка # 3
Это строка # 4
Это строка # 5
Это строка # 6
Это строка # 7
Это строка # 8
Это строка # 9
Это строка # 10
```

Обратите внимание, что в обоих листингах реализована возможность записи в файл как строки, так и целого. То же справедливо и для всех числовых типов, определенных в Object Pascal. Чтение информации, хранящейся в текстовом файле, можно выполнить, как показано в листинге 12.3.

Листинг 12.3. Чтение из текстового файла

```
var
  MyTextFile: TextFile;
  S: String[12];
  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;
```

Обратите внимание, что в листинге 12.3 строковая переменная `S` объявлена как `String[12]`. Такое объявление предотвращает чтение целой строки из файла в переменную `S` — в противном случае возникла бы ошибка при попытке прочитать значение в целочисленную переменную `j`. Кстати говоря, этот момент иллюстрирует еще одну важную особенность ввода-вывода из текстовых файлов — возможность записи в текстовые файлы данных, разбитых по столбцам с последующим чтением этих столбцов в строки заданной длины.

Важно иметь в виду, что ширина каждого столбца устанавливается равной определенному значению, даже если реальные строки имеют различную длину. Обратите также внимание на использование функции `Eof()`, которая определяет, находится ли указатель в конце файла. Если да, то необходимо прервать выполнение цикла, поскольку больше не осталось текста, который можно считывать.

Чтобы проиллюстрировать чтение текстового файла, отформатированного в виде столбцов, был создан файл `USCaps.txt`, включающий названия столиц штатов США. Часть этого файла выглядит следующим образом:

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

Столбец названия штата имеет ширину, равную 20 символам, поэтому названия столиц при распечатке выравниваются вертикально. Мы создали проект, в котором выполняется чтение этого файла и хранение списка штатов в Paradox-таблице. Его исходный код представлен в листинге 12.4.

Листинг 12.4. Исходный код проекта Capitals.dpr

```
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;
  // Назначаем файловую переменную текстовому файлу,
  // отформатированному в виде столбцов
  AssignFile(F, 'USCAPS.TXT');
  // Открываем файл с доступом только для чтения.
  Reset(F);
  try
    while not Eof(F) do
    begin
      {  Читаем строку файла в две строки, причем длина каждой
         из строк приемника совпадает с числом символов, составляющих
         ширину каждого столбца. }
      Readln(F, StateName, CapitalName);
      //Теперь сохраняем обе строки в отдельных столбцах Paradox-таблицы
      tblCapitals.Insert;
      tblCapitals['State_Name']    := StateName;
      tblCapitals['State_Capital'] := CapitalName;
      tblCapitals.Post;
    end;
  finally
    CloseFile(F); // По окончании чтения закрываем файл
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Очистка таблицы при запуске проекта
  tblCapitals.EmptyTable;
end;

end.
```

Текст приведенного выше проекта чрезвычайно прост, хотя программирование баз данных еще не рассматривалось. На этот пример стоит обратить внимание, поскольку соответствующая обработка текстовых файлов часто имеет большое прикладное значение. Этот текстовый файл с таким же успехом мог бы содержать информацию, например о банковских счетах, загруженную через компьютерную сеть.

Работа с типизированными файлами (файлами записей)

Структуры данных Object Pascal можно хранить в дисковых файлах, а затем считывать файловую информацию прямо в структуры данных. Это позволяет использовать типизированные файлы для хранения и чтения информации, подобно записям в таблице. Файлы, в которых хранятся структуры данных Object Pascal, называют *файлами записей.* Проиллюстрировать использование таких файлов можно следующей структурой записи:

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

На заметку Записи, содержащие такие типы данных, как `AnsiString` и `variant`, а также экземпляры классов, интерфейсы или динамические массивы, не могут быть записаны в файл.

Теперь предположим, что вы хотели бы сохранить одну или несколько таких записей в файле. В предыдущих разделах было показано, как это делается с помощью текстового файла. Теперь рассмотрим использование для этого файла записей, определенного следующим образом:

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

Чтобы прочитать одну запись типа `TPersonRec`, выполните код

```
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;
```

Следующий фрагмент иллюстрирует, как добавить в этот файл одну запись:

```
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;
```

Обратите внимание на использование процедуры `Seek()` для перемещения в конец файла перед внесением записи в файл. Эта функция достаточно хорошо описана в интерактивной справочной системе Delphi, поэтому здесь мы не будем углубляться в подробности ее применения.

Для иллюстрации использования типизированных файлов мы создали маленькое приложение, которое хранит информацию о персонале в формате Object Pascal. Это приложение позволяет просматривать, добавлять и редактировать записи. Мы также проиллюстрируем использование потомка класса `TFileStream` для инкапсуляции файловых операций ввода-вывода таких записей.

Определение потомка класса TFileStream для операций ввода-вывода с типизированными файлами

Класс `TFileStream` представляет собой класс потоков данных, который можно использовать для хранения элементов, не являющихся объектами. Структуры записей не обладают методами, позволяющими им сохраняться на диске или в памяти. Одно из решений состоит в том, чтобы сделать запись объектом, а затем к этом объекту присоединить функции хранения. Другое решение лежит в использовании для запоминания записей функций класса `TFileStream`. В листинге 12.5 представлен текст модуля, в котором определяется запись `TPersonRec`, и класс `TRecordStream`, который является потомком класса `TFileStream` и выполняет файловые операции ввода-вывода для запоминания и чтения записей.

На заметку

Более подробно работа с потоками данных (streaming) рассматривается в главе 22, “Сложные методики работы с компонентами”.

Листинг 12.5. Исходный код модуля PersRec.PAS

```
unit persrec;

interface
uses Classes, dialogs, sysutils;

type

  // Определяем запись для хранения информации о персонале
  TPersonRec = packed record
    FirstName: String[20];
    LastName: String[20];
    MI: String[1];
    BirthDay: TDateTime;
    Age: Integer;
  end;

  // Создаем потомок класса TFileStream для работы с записью 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;
    // Свойство NumRecs отображает число записей в потоке данных
    property NumRecs: Longint read GetNumRecs;
    // Свойство CurRec отражает текущую запись в потоке данных
    property CurRec: Longint read GetCurRec write SetCurRec;
  end;

implementation

function TRecordStream.GetRecSize:Longint;
begin
  {  Эта функция возвращает размер записи, с которой работает этот поток
     данных, т.е. записи TPersonRec. }
  Result := SizeOf(TPersonRec);
end;

function TRecordStream.GetNumRecs: Longint;
begin
```

```
  // Эта функция возвращает число записей в потоке данных
  Result := Size div GetRecSize;
end;

function TRecordStream.GetCurRec: Longint;
begin
 {  Эта функция возвращает позицию текущей записи. К этому
    значению нужно добавлять единицу, поскольку указатель
    файла всегда находится в начале записи, не отраженной
    в выражении Position div GetRecSize }
  Result := (Position div GetRecSize) + 1;
end;

procedure TRecordStream.SetCurRec(RecNo: Longint);
begin
  {  Эта процедура устанавливает позицию для записи в потоке,
     заданную значением переменной RecNo. }
  if RecNo > 0 then
    Position := (RecNo - 1) * GetRecSize
  else
    Raise Exception.Create('Cannot go beyond beginning of file.');
      // Нельзя выйти за начало файла
end;

function TRecordStream.SeekRec(RecNo: Longint; Origin: Word): Longint;
begin
  {  Эта функция перемещает указатель файла в позицию, заданную переменной RecNo. }

  {  ЗАМЕЧАНИЕ: Этот метод не содержит обработки ошибки, возникающей
     при попытке выхода за пределы начала/конца файла потоков данных. }
  Result := Seek(RecNo * GetRecSize, Origin);
end;

function TRecordStream.WriteRec(Const Rec): Longint;
begin
  // Эта функция записывает запись Rec в поток
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.AppendRec(Const Rec): Longint;
begin
  // Эта функция записывает запись Rec в поток
  Seek(0, 2);
  Result := Write(Rec, GetRecSize);
end;

function TRecordStream.ReadRec(var Rec): Longint;
begin
  {  Эта функция читает запись Rec из потока и перемещает указатель назад
     в начало этой записи. }
  Result := Read(Rec, GetRecSize);
  Seek(-GetRecSize, 1);
end;

procedure TRecordStream.First;
begin
  {  Эта функция перемещает указатель файла в начало потока. }
  Seek(0, 0);
end;

procedure TRecordStream.Last;
begin
  // Эта функция перемещает указатель файла в конец потока
  Seek(0, 2);
  Seek(-GetRecSize, 1);
end;

procedure TRecordStream.NextRec;
begin
  { Эта процедура перемещает указатель файла на следующую запись. }

  {  Переходим к следующей записи, если при этом не
```

```
     происходит выход за конец файла. }
  if ((Position + GetRecSize) div GetRecSize) = GetNumRecs then
    raise Exception.Create('Cannot read beyond end of file')
      // Нельзя читать за концом файла
  else
    Seek(GetRecSize, 1);
end;

procedure TRecordStream.PreviousRec;
begin
  {  Эта процедура перемещает указатель файла к
     предыдущей записи в потоке данных. }

  {  Вызываем эту функцию, если не происходит выхода за начало файла. }
  if (Position - GetRecSize >= 0) then
    Seek(-GetRecSize, 1)
  else
    Raise Exception.Create('Cannot read beyond beginning of the  file.');
      // Нельзя читать за началом файла
end;

end.
```

В этом модуле сначала объявляется запись `TPersonRec`, которая подлежит сохранению. Класс `TRecordStream` является потомком класса `TFileStream` и используется для выполнения файловых операций ввода-вывода для записи `TPersonRec`. Класс `TRecordStream` имеет два свойства: `NumRecs`, которое означает число записей в потоке данных, и `CurRec`, которое указывает на текущую запись в этом потоке.

С помощью метода `GetNumRecs()`, который является методом доступа к свойству `NumRecs`, определяется количество записей, существующих в потоке данных. Значение, возвращаемое функцией `GetNumRecs()`, образуется путем деления общего размера потока в байтах, полученного из свойства `TStream.Size`, на размер записи `TPersonRec`. Заметьте, однако, что точный размер записи достигается только в случае, если она упакована. Оказывается, такие структурированные типы, как записи и массивы, в целях более быстрого к ним доступа выравниваются по границам слов или двойных слов. Это значит, что запись занимает больше памяти, чем ей нужно на самом деле. При использовании перед объявлением записи зарезервированного слова `packed` применяется точный способ хранения данных, без выравнивания. В отсутствие ключевого слова `packed` результаты функции `GetNumRecs()` могут быть неточными.

С помощью метода `GetCurRec()` определяется, какая запись является текущей. Это делается путем деления свойства `TStream.Position` на размер свойства `TPersonRec` и прибавления к частному значения 1. Метод `SetCurRec()` размещает указатель файла в такой позиции потока данных, которая соответствует началу записи, заданной свойством `RecNo`.

Метод `SeekRec()` позволяет источнику вызова разместить указатель файла в позиции, определяемой параметрами `RecNo` и `Origin`. В зависимости от значения свойства `Origin` указатель файла можно переместить в потоке вперед или назад, отталкиваясь от начальной, конечной или текущей позиции указателя. Эти перемещения совершаются благодаря использованию метода `Seek()` объекта `TStream`. Использование метода `TStream.Seek()` описано в электронном справочном файле “Component Writers Guide”.

Метод `WriteRec()` записывает содержимое параметра `TPersonRec` в файл, начиная с текущей позиции, которая будет принадлежать существующей записи, поэтому последняя будет перезаписана содержимым параметра `TPersonRec`.

Метод `AppendRec()` добавляет новую запись в конец файла.

Метод `ReadRec()` читает данные из потока в параметр `TPersonRec`, а затем перемещает указатель файла в начало этой записи с помощью метода `Seek()`. Это делается с целью использования класса `TRecordStream` в характерном для баз данных стиле, в соответствии с которым указатель файла должен всегда находиться в начале текущей записи, и тогда эта запись будет в поле зрения системы.

Методы `First()` и `Last()` размещают указатель в начале и в конце файла соответственно.

Метод `NextRec()` размещает указатель файла в начале следующей записи при условии, если он не находится уже в позиции, принадлежащей последней записи файла.

Метод `PreviousRec()` размещает указатель файла в начале предыдущей записи при условии, если он не находится в позиции, принадлежащей первой записи файла.

Использование потомка класса TFileStream для файловых операций ввода-вывода

В листинге 12.6 содержится исходный код, обеспечивающий функционирование главной формы приложения, в котором используется объект `TRecordStream`.

Листинг 12.6. Исходный код главной формы проекта FileOfRec.dpr

```
unit MainFrm;

interface

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

const
  // Объявляем имя файла как константу
  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);
begin
  {  Если файл не существует, создаем его. В противном случае
     открываем его для чтения и записи. Это реализуется путем
     создания экземпляра класса 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; // Освобождаем экземпляр класса TRecordStream
end;

procedure TMainForm.ShowCurrentRecord;
begin
  // Читаем текущую запись
  RecordStream.ReadRec(PersonRec);
  // Копируем данные из параметра PersonRec в элементы управления формы
  with PersonRec do
  begin
    edtFirstName.Text := FirstName;
    edtLastName.Text  := LastName;
    edtMI.Text        := MI;
    dtpBirthDay.Date  := BirthDay;
    meAge.Text        := IntToStr(Age);
  end;
  // Отображаем на главной форме номер записи и общее число записей
  lblRecNo.Caption  := IntToStr(RecordStream.CurRec);
  lblNoRecs.Caption := IntToStr(RecordStream.NumRecs);
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Отображаем текущую запись, если она существует
  if RecordStream.NumRecs <> 0 then
     ShowCurrentRecord;
end;

procedure TMainForm.btnAppendClick (Sender: TObject);
begin
  // Копируем содержимое элементов управления формы в запись PersonRec
  with PersonRec do
  begin
    FirstName := edtFirstName.Text;
    LastName  := edtLastName.Text;
    MI        := edtMI.Text;
    BirthDay  := dtpBirthDay.Date;
    Age       := StrToInt(meAge.Text);
  end;
  // Записываем новую запись в поток
  RecordStream.AppendRec(PersonRec);
  // Отображаем текущую запись
  ShowCurrentRecord;
end;

procedure TMainForm.btnUpdateClick(Sender: TObject);
begin
  {  Копируем содержимое элементов управления формы в запись
     PersonRec и записываем ее в поток данных. }
  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
  {  Переходим на первую запись в потоке и отображаем ее,
     если в потоке существуют какие-либо записи }
```

```
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.First;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnNextClick(Sender: TObject);
begin
  // Переходим на следующую запись в потоке, если в нем вообще есть записи
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.NextRec;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnLastClick(Sender: TObject);
begin
  { Переходим на последнюю запись в потоке, если в нем вообще есть записи }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.Last;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnPrevClick(Sender: TObject);
begin
  { Переходим на предыдущую запись в потоке, если в нем вообще есть записи }
  if RecordStream.NumRecs <> 0 then
  begin
    RecordStream.PreviousRec;
    ShowCurrentRecord;
  end;
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  // Очищаем все элементы управления на форме
  edtFirstName.Text := '';
  edtLastName.Text  := '';
  edtMI.Text        := '';
  meAge.Text        := '';
end;

end.
```

На рис. 12.1 показана главная форма для этого проекта.

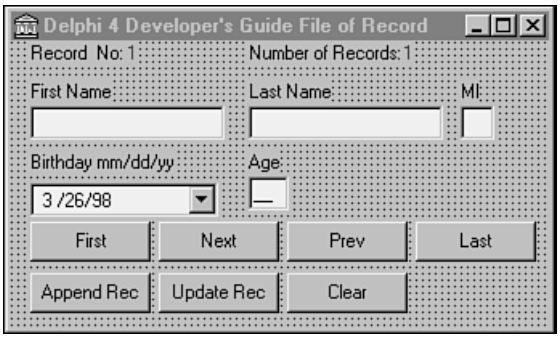

Рис. 12.1. Главная форма, демонстрирующая работу экземпляра класса `TRecordStream`

Главная форма содержит как поле `TPersonRec`, так и класс `TRecordStream`. В поле `TPersonRec` хранится содержимое текущей записи. Экземпляр класса `TRecordStream` создается в обработчике события формы `OnCreate`. Если заданный файл еще не существует, то выполняется его создание. В противном случае он просто открывается.

Метод `ShowCurrentRecord()` используется для извлечения текущей записи из потока данных путем вызова метода `RecordStream.ReadRec()`, который сначала читает запись, а затем перемещает указатель файла в начало этой записи.

Работа большинства функций этого приложения описана в комментариях исходного кода. Здесь же обратим ваше внимание лишь на самые важные моменты.

Процедура `btnAppendClick()` добавляет в файл новую запись.

Метод `btnUpdateClick()` записывает содержимое элементов управления формы в файл, начиная с позиции текущей записи; таким образом модифицируется содержимое этого участка файла.

Остальные методы перемещают указатель файла на следующую, предыдущую, первую или последнюю запись в файле, что позволяет легко просматривать любые записи.

Данный пример демонстрирует, как с помощью файловых операций ввода-вывода использовать типизированные файлы для выполнения простых манипуляций, необходимых для работы с базами данных. Этот пример также иллюстрирует возможность использования объекта `TFileStream` в качестве упаковки функций ввода-вывода записей применительно к файлу.

Работа с нетипизированными файлами

До сих пор рассматривались способы манипуляции с текстовыми и типизированными файлами. Текстовые файлы используются для хранения последовательностей ASCII-символов, а типизированные — для хранения данных, в которых каждый элемент отвечает определенному формату структуры записи Object Pascal. В обоих случаях каждый файл хранит некоторое количество байтов, которые могут быть интерпретированы приложениями соответствующим образом.

Однако многие файлы не утруждают себя наличием строгого формата. Например, RTF-файлы, не только содержат текст, но и включают информацию о различных текстовых атрибутах. Для просмотра содержимого эти файлы нельзя загрузить в обычный текстовый редактор, для этого нужно применить специальную программу, которая в состоянии интерпретировать данные, использующие формат RTF.

Ниже иллюстрируются возможности работы с нетипизированными файлами.

Следующая строка объявляет нетипизированный файл:

```
var UntypedFile: File;
```

Этой инструкцией объявляется файл, состоящий из последовательности блоков размером 128 байт.

Для чтения данных из нетипизированного файла используется процедура `BlockRead()`, а для записи — процедура `BlockWrite()`. Эти процедуры объявляются следующим образом:

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

Обе процедуры принимают три обязательных параметра. Параметр `F` представляет собой переменную нетипизированного файла. Параметр `Buf` является переменной буфера, который предназначен для хранения данных, считанных или записываемых в файл. Параметр `Count` содержит количество записей, которые нужно прочитать из файла. Необязательный параметр `Result` содержит количество записей, действительно считанных из файла во время операции чтения. Аналогично, параметр `Result` в операции записи содержит количество записей, полностью записанных в файл. Если это значение не равно значению `Count`, возможно, на диске не хватило места.

Постараемся объяснить смысл утверждения о том, что эти процедуры читают или записывают число записей, равное `Count`. Если объявить нетипизированный файл, как показано ниже, то по умолчанию этой строкой определяется файл, каждая запись которого состоит из 128 байт данных:

```
UntypedFile: File;
```

Это объявление не имеет ничего общего с любой отдельно взятой структурой записи. Оно просто задает размер блока данных, в который считывается одна запись. В листинге 12.7 демонстрируется, как прочитать из файла одну запись (размером 128 байт).

Листинг 12.7. Чтение из нетипизированного файла

```
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;
```

В этой небольшой программе вы открываете файл с именем `SOMEFILE.DAT` и читаете 128 байт данных (что соответствует одной записи или блоку) в буфер с подходящим именем `Buffer`. Чтобы записать в файл 128 байт данных, воспользуйтесь вариантом программы, предложенным в листинге 12.8.

Листинг 12.8. Запись данных в нетипизированный файл

```
var
  UnTypedFile: File;
  Buffer: array[0..128] of byte;
  NumRecsWritten: Integer;
begin
  AssignFile(UnTypedFile, 'SOMEFILE.DAT');
  // Если файл не существует, создаем его. В противном случае
  // просто открываем его для чтения и записи
  if FileExists('SOMEFILE.DAT') then
    Reset(UnTypedFile)
  else
    Rewrite(UnTypedFile);
  try
    // Перемещаем указатель файла в конец файла
    Seek(UnTypedFile, FileSize(UnTypedFile));
    FillChar(Buffer, SizeOf(Buffer), 'Y');
    BlockWrite(UnTypedFile, Buffer, 1, NumRecsWritten);
  finally
    CloseFile(UnTypedFile);
  end;
end;
```

Если используется стандартный размер блока (128 байт), то при чтении из нетипизированного файла может возникнуть проблема, связанная с тем, что размер этого файла должен быть кратным 128. В противном случае будет предпринята попытка чтения за пределами конца файла. Эту проблему можно легко обойти, задав с помощью процедуры `Reset()` размер записи равным одному байту. Если передать размер записи, равный одному байту, то при чтении блоков любого размера общий размер файла будет всегда кратным одному байту. В качестве примера листинг 12.9 демонстрирует простую программу копирования файлов с помощью процедур `BlockRead()` и `BlockWrite()`.

Листинг 12.9. Демонстрационная программа копирования файлов

```
unit MainFrm;

interface

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

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
  {  Назначим файлам источника и приемника соответствующие
     файловые переменные. }
  AssignFile(SrcFile, 'srcfile.tst');
  AssignFile(DestFile, 'destfile.tst');
  // Откроем файл источника с доступом для чтения
  Reset(SrcFile, 1);
  try
    // Откроем файл приемника с доступом для записи
    Rewrite(DestFile, 1);
    try
      {  Поместим процесс копирования в блок try..except,
         чтобы в случае ошибки можно было удалить этот файл. }
      try
        // Сначала установим общее число прочитанных байтов равным нулю
        TotalRead := 0;
        // Получаем размер файла источника
        FSize := FileSize(SrcFile);
        {  Читаем SizeOf(Buffer) байтов из файла источника
          и добавляем эти байты в файл приемника. Повторяем этот
          процесс до тех пор, пока все байты не будут прочитаны
          из файла источника. В окне процесса отображается динамика
          выполнения операции копирования. }
        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
        {  В случае возникновения исключительной ситуации удаляем
           файл приемника, поскольку он может быть с искажениями.
           Затем повторно генерируем исключительную ситуацию. }
        Erase(DestFile);
        raise;
      end;
    finally
      CloseFile(DestFile); // Закрываем файл приемника
    end;
  finally
    CloseFile(SrcFile);   // Закрываем файл источника
  end;
end;

end.
```

На заметку

Одна из демонстрационных программ, поставляемых вместе с Delphi 4, содержит несколько полезных функций, среди которых есть и функция копирования файла. Эта демонстрационная программа находится в каталоге `\DEMOS\DOS\FILMANEX\`. Ниже перечислены функции, содержащиеся в файле `FmxUtils.PAS`.

```
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;
```

Прежде всего в демонстрационной программе открывается файл источника и создается файл приемника, в который будут скопировано содержимое файла источника. Переменные `TotalRead` и `FSize` используются для обновления окна динамики процесса `TProgressBar`, чтобы периодически отображать состояние операции копирования. Сама операция копирования выполняется внутри цикла `repeat`. Сначала из файла источника читается `SizeOf(Buffer)` байтов, а переменная `BytesRead` определяет реальное количество прочитанных байтов. Затем делается попытка скопировать `BytesRead` байтов в файл приемника, в то время как количество реально записанных байтов сохраняется в переменной `BytesWritten`. Если во время операции копирования не возникло никакой ошибки, то в этот момент переменные `BytesRead` и `BytesWritten` должны иметь одинаковые значения. Описанный процесс продолжается до тех пор, пока не будут скопированы все байты файла. При возникновении ошибки возбуждается исключительная ситуация и файл приемника удаляется с диска.

Структуры записей TTextRec и TFileRec

Большинство функций управления файлами на самом деле являются функциями или прерываниями операционной системы, взятыми на вооружение процедурами Object Pascal. Например, функция `Reset()` представляет собой Pascal-оболочку для функции Win32 `CreateFileA()` из библиотеки KERNEL32. Имея дело с функциями Object Pascal, а не с самими функциями Win32, вы можете не беспокоиться насчет деталей реализации файловых операций. Однако остается неясным, как в случае необходимости получить доступ к определенным файловым атрибутам (например, к дескриптору файла), поскольку они скрыты для пользователей Object Pascal.

При использовании “неродных” для Object Pascal функций, которые требуют дескриптор файла (например, функция `LZCopy()`), последний можно получить с помощью операций приведения типа, выполняемых над переменными текстового или двоичного файлов: `TTextRec` или `TFileRec` соответственно. Эти типы записей содержат дескриптор файла и другие файловые атрибуты. Но кроме дескриптора файла, вам вряд ли придется когда-либо получать доступ к другим полям данных. Процедура получения дескриптора файла выглядит следующим образом:

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

Определение записи `TTextRec` имеет вид:

```
PTextBuf = ^TTextBuf;
TTextBuf = array[0..127] of Char; // Определение буфера для первых
                                  // 127 символов в файле
TTextRec = record
    Handle: Integer;              // Дескриптор файла
    Mode: Integer;                // Режим файла
    BufSize: Cardinal;            // Следующие 4 параметра
    BufPos: Cardinal;             // используются для буферизации памяти
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;             // Поля XXXXFunc являются указателями на
    InOutFunc: Pointer;            // функции доступа к файлу. Их можно
    FlushFunc: Pointer;            // модифицировать при написании
    CloseFunc: Pointer;            // файловых драйверов устройств
    UserData: array[1..32] of Byte;// Не используется
    Name: array[0..259] of Char;   // Полное имя файла
    Buffer: TTextBuf;              // Буфер, содержащий
                                   // первые 127 символов файла
  end;
```

Определение записи `TFileRec` выглядит следующим образом:

```
TFileRec = record
  Handle: Integer;                  // Дескриптор файла
  Mode: Integer;                    // Режим файла
  RecSize: Cardinal;                // Размер каждой записи файла
  Private: array[1..28] of Byte;    // Используется для внутренних целей
                                    // Object Pascal
  UserData: array[1..32] of Byte;   // Не используется
  Name: array[0..259] of Char;      // Полное имя файла
end;
```

Работа с файлами, отображенными в память

Очевидно, одним из наиболее удобных средств Win32 является возможность получать доступ к файлам на диске подобно доступу к их содержимому в памяти. Это средство доступа обеспечивается благодаря файлам, отображенным в память.

С помощью файлов, отображенных в память, можно избежать необходимости выполнения всех файловых операций ввода-вывода. Вместо этого резервируется некоторая область виртуального адресного пространства, а затем физическая память файла на диске связывается с адресом этого зарезервированного пространства памяти. После этого можно ссылаться на содержимое файла с помощью указателя, действующего в этой зарезервированной области. Немного позже мы покажем, как использовать эту возможность для создания эффективной утилиты поиска текста для текстовых файлов, которая своей простотой обязана использованию файлов, отображенных в память.

Применение файлов, отображенных в память

Для файлов, отображенных в память, находят три области применения.

- Система Win32 загружает и выполняет EXE- и DLL-файлы, используя файлы, отображенные в память. При этом экономится пространство файла подкачки (paging file) и, как следствие, для таких файлов уменьшается время загрузки.
- Доступ к содержимому файлов, отображенных в память, осуществляется с помощью указателя на область отображенной памяти. При этом не только упрощается процесс доступа к файлам, но разработчики освобождаются от необходимости применять различные схемы буферизации файлов.
- Файлы, отображенные в память, предоставляют возможность разделения данных среди различных процессов, выполняющихся на одном и том же компьютере.

В этой главе мы не будем касаться первой области применения файлов, отображенных в память, поскольку это затрагивает поведение системы в целом. Рассмотрим лишь вторую область применения файлов, отображенных в память, так как, вероятно, именно она может заинтересовать вас как разработчика. Что касается третьей области применения, то в главе 9, “Динамически компонуемые библиотеки”, было показано, как разделять данные между процессами с помощью файлов, отображенных в память. Прочитав этот раздел, вы можете вернуться к указанной главе, чтобы лучше разобраться в приведенных примерах.

Доступ к содержимому файлов, отображенных в память

При создании файла, отображенного в память, он, по сути, связывается с некоторой областью адресного пространства виртуальной памяти процесса. Для получения этой связи необходимо создать *объект отображения файла* (file-mapping object). Для просмотра или редактирования содержимого файла нужно иметь специальное окно просмотра (file view), которое позволит получить доступ к содержимому файла с помощью указателя, подобно доступу к некоторой области памяти.

При записи в это окно просмотра система выполняет кэширование, буферизацию и загрузку данных файла, а также управляет выделением и освобождением памяти, и вам остается только редактировать данные, расположенные в некоторой области памяти (причем файловые операции ввода-вывода полностью лежат на плечах системы). В этом и заключается вся прелесть использования файлов, отображенных в память, — ведь задача управления файлами значительно упрощается по сравнению со стандартными методами ввода-вывода, рассмотренными ранее. Кроме того, есть выигрыш и в скорости выполнения этих операций.

В следующих разделах описана последовательность действий, необходимая для создания и открытия файла, отображенного в память.

Создание или открытие файла

Первый шаг на пути к созданию или открытию файла, отображенного в память, состоит в получении дескриптора для файла, подлежащего отображению. Для этого можно воспользоваться одной из функций:

FileCreate() или FileOpen(). Определение функции FileCreate() содержится в модуле SysUtils.pas и выглядит следующим образом:

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

Эта функция создает новый файл с именем, заданным строковым параметром FileName. При успешном выполнении этой функции возвращается действительное значение дескриптора файла. В противном случае возвращается значение, определенное константой INVALID_HANDLE_VALUE.

Функция FileOpen() открывает существующий файл, используя заданный режим доступа. При успешном завершении эта функция возвратит действительный дескриптор файла. В противном случае будет возвращено значение, определенное константой INVALID_HANDLE_VALUE. Определение функции FileOpen() находится в модуле SysUtils.pas и выглядит следующим образом:

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

Первый параметр представляет собой полный путь к файлу, к которому применяется операция отображения в память. В качестве второго параметра можно передать один из возможных режимов доступа, описанных в табл. 12.1.

Таблица 12.1. Режимы доступа к файлу (fmOpenXXXX)

Режим доступа	*Значение*
fmOpenRead	Позволяет выполнять только чтение из файла
fmOpenWrite	Позволяет выполнять только запись в файл
fmOpenReadWrite	Позволяет выполнять и чтение и запись в файл

Если в качестве параметра Mode передать значение 0, то вы не сможете ни прочитать, ни записать данные в файл. Нулевое значение можно использовать только для получения различных атрибутов файла. Характер совместного использования заданного файла различными приложениями можно определить путем применения поразрядной операции OR (ИЛИ), используя режимы доступа, перечисленные в табл. 12.1, вместе с одним из режимов fmShareXXXX, описанных в табл. 12.2.

Таблица 12.2. Режимы доступа fmShareXXXX

Режим разделения	*Значение*
fmShareCompat	Механизм разделения файла совместим с блоками управления DOS 1.x и 2.x. Это значение используется совместно с другими режимами fmShareXXXX
fmShareExclusive	Разделение не разрешается
fmShareDenyWrite	Попытки других программ открыть файл с доступом fmOpenWrite будут неуспешными
fmShareDenyRead	Попытки других программ открыть файл с доступом fmOpenRead будут неуспешными
fmShareDenyNone	Попытки других программ открыть файл с любым доступом будут успешными

Возвратив действительный дескриптор файла, можно получить объект отображения в память.

Создание объекта отображения в память

Для создания *именованных* или *неименованных* объектов отображения файла используйте функцию CreateFileMapping(), которая определяется следующим образом:

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

С помощью параметров, передаваемых функции `CreateFileMapping()`, системе предоставляется информация, необходимая для создания объекта отображения файла. Первый параметр `hFile` является дескриптором файла, полученным от предыдущего обращения к функции `FileOpen()` или `FileCreate()`. Важно отметить, что этот файл открывается с признаками защиты, совместимыми с параметром `flProtect`, который будет рассмотрен чуть ниже. В главе 9, "Динамически компонуемые библиотеки", описан метод создания условий для разделения данных между отдельными процессами: в этом случае также используется функция `CreateFileMapping()` для создания объекта отображения файла, поддерживаемого системным файлом подкачки.

Параметр `lpFileMappingAttributes` является указателем типа `PSecurityAttributes`, который ссылается на атрибуты защиты для объекта отображения файла. Этот параметр практически всегда равен нулю.

Параметр `flProtect` задает тип защиты, применяемый к окну просмотра файла. Как упоминалось ранее, это значение должно быть совместимо с атрибутами, с которыми файл был открыт для получения его дескриптора. В табл. 12.3 перечислены различные атрибуты, которые могут быть присвоены параметру `flProtect`.

Таблица 12.3. Атрибуты защиты

Атрибут защиты	*Значение*
`PAGE_READONLY`	Содержимое этого файла можно читать. Файл должен быть создан с помощью функции `FileCreate()` или открыт путем вызова функции `FileOpen()` с режимом доступа `fmOpenRead`
`PAGE_READWRITE`	Для этого файла разрешено чтение и запись. Файл должен быть открыт с режимом доступа `fmOpenReadWrite`
`PAGE_WRITECOPY`	Для этого файла разрешено чтение и запись, однако при записи в файл создается закрытая копия модифицированной страницы. Смысл этой копии состоит в том, что файлы, отображенные в память, которые разделяются между процессами, не потребляют двойной объем ресурсов в системной памяти или в памяти файла подкачки. Дублированию подлежит только память, необходимая для хранения модифицированных страниц. Файл должен быть открыт с режимом доступа `fmOpenWrite` или `fmOpenReadWrite`

К параметру `flProtect` можно также применить атрибуты раздела с использованием поразрядного логического оператора `or`. Значения атрибутов описаны в табл. 12.4.

Таблица 12.4. Атрибуты раздела

Атрибут раздела	*Значение*
`SEC_COMMIT`	Выделяет для всех страниц раздела физическую память в области системной памяти или в файле подкачки. Это значение устанавливается по умолчанию
`SEC_IMAGE`	Информация об отображении файла и атрибутах берется из образа файла. Этот атрибут применяется только к исполняемым файлам изображений. (Примечание: данный атрибут игнорируется в среде Windows 95.)
`SEC_NOCACHE`	Страницы, отображенные в память, не кэшируются, следовательно все, что записывается в этот файл, применяется системой непосредственно к данным файла на диске. (Примечание: данный атрибут игнорируется в среде Windows 95.)
`SEC_RESERVE`	Резервирует страницы раздела без выделения физической памяти

Параметр `dwMaximumSizeHigh` задает старшие 32 разряда максимального размера объекта отображения файла. Если вы не собираетесь получать доступ к файлам, размер которых превышает 4 Гбайт, это значение всегда будет равно `0`.

Параметр dwMinimumSizeLow задает младшие 32 разряда максимального размера объекта отображения файла. Нулевое значение этого параметра будет означать, что максимальный размер объекта отображения файла равен размеру отображаемого файла.

Параметр lpName определяет имя объекта отображения файла. Это значение может содержать любой символ, за исключением обратной косой черты (\). Если значение этого параметра совпадает с именем существующего объекта отображения файла, значит, эта функция требует доступа к тому же объекту отображения с помощью атрибутов, заданных параметром flProtect. В качестве параметра lpName допускается передача значения nil, что приводит к созданию неименованного объекта отображения файла.

При успешном выполнении функции CreateFileMapping() возвращается дескриптор, связанный с объектом отображения файла. Если функции CreateFileMapping() не удалось создать заказанный объект, она вернет значение nil. Чтобы установить причину неудачи, следует вызвать функцию GetLastError(). Если окажется, что создаваемый объект отображения файла указывает на уже существующий объект подобного типа, то функция GetLastError() вернет значение ERROR_ALREADY_EXISTS.

Внимание

В Windows 95 не применяются функции, выполняющие операции ввода-вывода на основе дескрипторов файлов, использовавшихся для создания отображений, в результате чего в таких файлах данные могут оказаться несогласованными. Поэтому рекомендуется открывать файл с исключительными правами доступа (подробнее это описано ниже, в разделе “Согласованность файлов, отображенных в память”).

Получив объект отображения файла, можно отображать данные файла в адресное пространство процесса.

Отображение данных файла в адресное пространство процесса

Для отображения данных файла в адресное пространство процесса предназначена функция MapViewOfFile(), которая определяется следующим образом:

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

Параметр hFileMappingObject представляет собой дескриптор, связанный с открытым объектом отображения, который был создан с помощью обращения к функции CreateFileMapping() либо к функции OpenFileMapping().

Параметр dwDesiredAccess определяет способ доступа к данным файла и может иметь одно из значений, перечисленных в табл. 12.5.

Таблица 12.5. Доступ к данным файла

Значение параметра dwDesiredAccess	*Описание*
FILE_MAP_WRITE	Позволяет выполнять чтение и запись данных файла. При этом в функции CreateFileMapping() должен использоваться атрибут PAGE_READ_WRITE
FILE_MAP_READ	Позволяет выполнять чтение и запись данных файла. При этом в функции CreateFileMapping() должны использоваться атрибуты PAGE_READ_WRITE или PAGE_READ
FILE_MAP_ALL_ACCESS	Предоставляет тот же доступ, который действует при использовании атрибута FILE_MAP_WRITE
FILE_MAP_COPY	Предоставляет доступ к записи с копированием. При записи в файл создается закрытая (private) копия записываемой страницы. В этом случае функция CreateFileMapping() должна использоваться с атрибутами PAGE_READ_ONLY, PAGE_READ_WRITE или PAGE_WRITE_COPY

Параметр dwFileOffsetHigh задает старшие 32 разряда смещения файла, откуда начинается его отображение.

Параметр dwFileOffsetLow задает младшие 32 разряда смещения файла, откуда начинается его отображение.

Параметр dwNumberOfBytesToMap определяет количество байтов файла, предназначенных для отображения. Нулевое значение указывает на отображение целого файла.

При успешном выполнении функция MapViewOfFile() возвращает начальный адрес отображения, а в случае неудачи — значение nil, и тогда для определения причины ошибки необходимо вызвать функцию GetLastError().

Освобождение окна просмотра файла

С помощью функции UnmapViewOfFile(), образно говоря, "заметаются следы" отображения файла в адресном пространстве вызывающего процесса. Эта функция определяется следующим образом:

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

Единственный параметр lpBaseAddress этой функции должен указывать на базовый адрес отображаемой области, который совпадает со значением, возвращаемым функцией MapViewOfFile().

По окончании работы с файлом следует непременно вызвать функцию UnmapViewOfFile(); в противном случае отображаемая область памяти не будет освобождена системой до тех пор, пока не закончится ваш процесс.

Закрытие объекта отображения

Обращения к функциям FileOpen() и CreateFileMapping() связаны с открытием объектов ядра операционной системы, и поэтому вы сами несете ответственность за их закрытие. Это можно сделать с помощью функции CloseHandle(), которая определяется следующим образом:

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

При успешном выполнении функция CloseHandle() возвращает значение True, в противном случае — значение False, и тогда для определения причины ошибки нужно проверить результат функции GetLastError().

Пример использования файла, отображенного в память

Для иллюстрации использования функций работы с файлом, отображенным в память, рассмотрим листинг 12.10.

Листинг 12.10. Простой пример использования файла, отображенного в память

```
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);
  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);
  // По умолчанию преобразуем в прописные символы
  UCase := True;
end;

procedure TMainForm.ChangeFileCase;
var
  FFileHandle: THandle; // Дескриптор открытого файла
  FMapHandle: THandle;  // Дескриптор объекта отображения файла
  FFileSize: Integer;   // Переменная для хранения размера файла
  FData: PByte;         // Указатель на данные файла при отображении
  PData: PChar;         // Указатель для ссылки на данные файла
begin

  {  Сначала получаем дескриптор файла, который подлежит
     отображению. Этот код предполагает существование такого
     файла. В противном случае для создания нового файла
     можно использовать функцию FileCreate(). }

  if not FileExists(FName) then
    raise Exception.Create('File does not exist.') // Файл не существует
  else
    FFileHandle := FileOpen(FName, fmOpenReadWrite);

  // В случае неудачного выполнения функции CreateFile()
  // генерируется исключительная ситуация
  if FFileHandle = INVALID_HANDLE_VALUE then
    raise Exception.Create('Failed to open or create file');
        // Файл не был открыт или не создан
    try
    {  Теперь получим размер файла, который будет передан другим функциям
       отображения файла. Сделаем этот размер на один байт больше, чтобы
       добавить завершающий нуль-символ в конец файла, отображенного в память.}
    FFileSize := GetFileSize(FFileHandle, Nil);

   {  Получаем дескриптор объекта отображения файла. Если работа этой функции
      не увенчалась успехом, генерируем исключительную ситуацию. }
    FMapHandle := CreateFileMapping(FFileHandle, nil,
       PAGE_READWRITE, 0, FFileSize, nil);

    if FMapHandle = 0 then
      raise Exception.Create('Failed to create file mapping');
        // Не удалось создать отображение файла
  finally
        // Освобождаем дескриптор файла
    CloseHandle(FFileHandle);
  end;

  try
    {  Представляем объект отображения файла в окне просмотра. Эта функция
       вернет указатель на данные файла. В случае неуспешной работы
```

```
        генерируется исключительная ситуация. }
      FData := MapViewOfFile(FMapHandle, FILE_MAP_ALL_ACCESS, 0, 0, FFileSize);

      if FData = Nil then
        raise Exception.Create('Failed to map view of file');
          // Не удалось отобразить окно просмотра файла
    finally
          // Освобождаем дескриптор объекта отображения файла
      CloseHandle(FMapHandle);
    end;

    try
      {  !!! Сюда следует поместить функции для работы с данными файла,
         отображенного в память. Например, следующая строка преобразует все
         символы файла в прописные буквы. }
      PData := PChar(FData);
      // Устанавливаем указатель в конец данных файла
      inc(PData, FFileSize);

      // Добавляем завершающий нуль-символ в конец данных файла
      PData^ := #0;

      // Теперь переводим все символы файла в прописные буквы
      if UCase then
        StrUpper(PChar(FData))
      else
        StrLower(PChar(FData));

    finally
      // Освобождаем отображение файла
      UnmapViewOfFile(FData);
    end;
    memTextContents.Lines.Clear;
    memTextContents.Lines.LoadFromFile(FName);
  end;

end.
```

Как видно из листинга 12.10, первый этап состоит в получении дескриптора файла, подлежащего отображению в область памяти процесса. Он реализуется путем вызова функции `FileOpen()`. Этой функции передается режим доступа к файлу, равный значению `fmOpenReadWrite`, чтобы получить возможность чтения содержимого файла и записи в него.

Затем вы получаете размер файла и устанавливаете последний символ равным завершающему нулю. На самом деле это должен быть маркер конца файла, байтовое значение которого совпадает с ограничивающим нулем. Все это делается ради ясности и простоты. Поскольку вы обращаетесь с данными файла как со строками, ограниченными завершающими нулями, наличие такого завершающего нуль-символа просто необходимо.

Следующий этап — получение объекта файла отображения в память путем вызова функции `CreateFileMapping()`. При неуспешном выполнении этой функции генерируется исключительная ситуация. В противном случае вы переходите к следующему этапу — представлению объекта отображения файла в окне просмотра. И вновь в случае сбоя в работе этой функции генерируется исключение.

Затем выполняется смена регистра представления данных (перевод символов в прописные буквы). Если бы после выполнения этой процедуры вы просмотрели файл в текстовом редакторе, то увидели бы, что все символы этого файла преобразованы в прописные буквы. И наконец, отмена отображения окна просмотра файла выполняется путем вызова функции `UnmapViewOfFile()`.

Согласованность файлов, отображенных в память

Система Win32 гарантирует, что несколько окон просмотра файла остаются согласованными, если они отображены с помощью одного и того же объекта отображения файла. Это значит, что если содержимое файла модифицируется в одном окне, то второе окно будет “в курсе” этих модификаций. Но имейте в виду, что это справедливо только при использовании одних и тех же объектов отображения файлов. Использование же различных объектов отображения не гарантирует согласованности нескольких окон просмотра. Следует отметить, что эта проблема существует только для файлов, которые отображаются с доступом для записи. Окна просмотра для файлов, открываемых только для чтения, всегда согласованы. Следует также отметить, что

файлы, разделяемые в сети, не сохраняют согласованности в отображениях файлов с доступом для записи на различных машинах.

Утилита поиска в текстовых файлах

Для иллюстрации эффективности использования файлов, отображенных в память, мы создали проект, предназначенный для поиска текста в текстовых файлах текущего каталога. Главная форма этого проекта показана на рис. 12.2. В окно списка, расположенного на главной форме, добавляются имена файлов вместе с количеством вхождений искомой строки, обнаруженных в соответствующем файле.

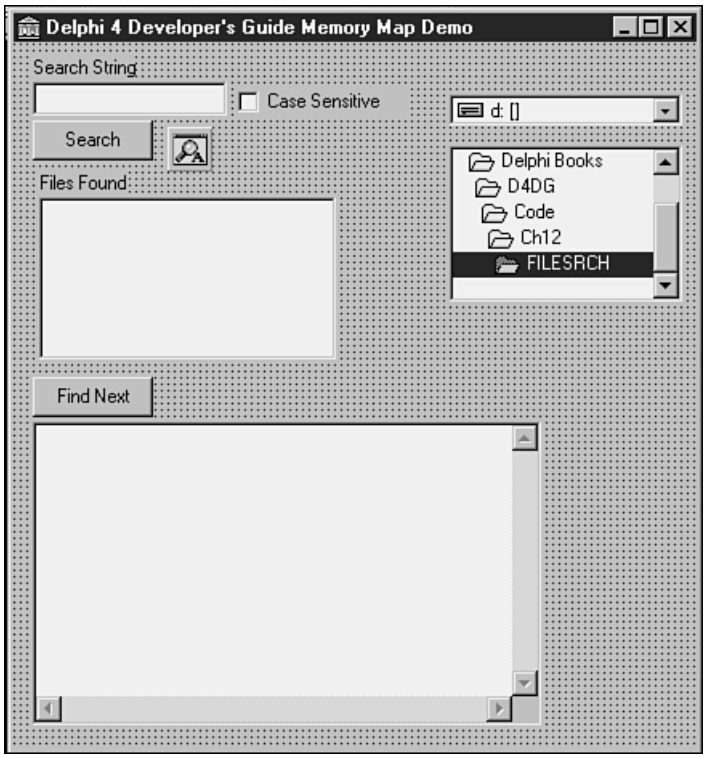

Рис. 12.2. Главная форма проекта поиска текста

Проект служит также иллюстрацией того, как в объекте инкапсулируются функции файлов, отображенных в память. Для демонстрации был создан класс `TMemMapFile`.

Класс TMemMapFile

Исходный текст модуля, содержащего класс `TMemMapFile`, представлен в листинге 12.11.

Листинг 12.11. Исходный код модуля MemMap.pas, в котором определяется класс TMemMapFile

```
unit MemMap;

interface

uses Windows, SysUtils, Classes;

type
  EMMFError = class(Exception);

  TMemMapFile = class
  private
    FFileName: String;     // Имя файла, отображенного в память
    FSize: Longint;        // Размер окна отображения
    FFileSize: Longint;    // Действительный размер файла
    FFileMode: Integer;    // Режим доступа к файлу
    FFileHandle: Integer; // Дескриптор файла
    FMapHandle: Integer;  // Дескриптор объекта отображения файла
```

```
    FData: PByte;         // Указатель на данные файла
    FMapNow: Boolean;     // Определяет, немедленно ли
                          // отображать окно просмотра
    procedure AllocFileHandle;
    {  Считывает дескриптор дискового файла }
    procedure AllocFileMapping;
    { Считывает дескриптор объекта отображения файла }
    procedure AllocFileView;
    {  Отображает окно просмотра в файл }
    function GetSize: Longint;
    {  Возвращает размер отображенного окна }
  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);
{  Создает окно просмотра файла FileName.
  FileName: Полный путь файла.
  FileMode: Используются константы fmXXX.
  Size: размер отображения памяти. При
  использовании собственного размера файла
  передайте нулевое значение.
}
begin

  {  Инициализируем закрытые поля }
  FMapNow := MapNow;
  FFileName := FileName;
  FFileMode := FileMode;

  AllocFileHandle;  // Получаем дескриптор дискового файла
  {  Предполагаем, что файл < 2 гигабайт }

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

  try
    AllocFileMapping; // Получаем дескриптор объекта отображения файла
  except
    on EMMFError do
    begin
      CloseHandle(FFileHandle);
        // Закрываем дескриптор файла в случае ошибки
      FFileHandle := 0;
        // Снова устанавливаем дескриптор равным
      raise;
        // Передаем исключительную ситуацию
    end;
  end;
  if FMapNow then
    AllocFileView;  // Отображаем окно просмотра файла
end;

destructor TMemMapFile.Destroy;
begin

  if FFileHandle <> 0 then
    CloseHandle(FFileHandle); // Освобождаем дескриптор файла

  { Освобождаем дескриптор объекта отображения файла }
  if FMapHandle <> 0 then
```

```
    CloseHandle(FMapHandle);

  FreeMapping; {  Отменяем отображение окна просмотра файла }
  inherited Destroy;
end;

procedure TMemMapFile.FreeMapping;
{  В этом методе отменяется отображение окна просмотра файла
   в адресное пространство данного процесса }
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;
{  Создает или открывает дисковый файл перед созданием файла,
   отображенного в память }
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');
        // Не удалось открыть или создать файл
end;

procedure TMemMapFile.AllocFileMapping;
var
  ProtAttr: DWORD;
begin
  if FFileMode = fmOpenRead then
    // Получаем атрибут корректной защиты
    ProtAttr := Page_ReadOnly
  else
    ProtAttr := Page_ReadWrite;
  {  Пытаемся создать отображение дискового файла.
     В случае ошибки возбуждаем исключительную ситуацию. }
  FMapHandle := CreateFileMapping(FFileHandle, Nil, ProtAttr,
      0, FSize, Nil);
  if FMapHandle = 0 then
    raise EMMFError.Create('Failed to create file mapping');
        // Не удалось создать отображение файла
end;

procedure TMemMapFile.AllocFileView;
var
  Access: Longint;
begin
  if FFileMode = fmOpenRead then
    // Получаем корректный режим доступа к файлу
    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');
        // Не удалось отобразить окно просмотра файла
end;
```

```
end.
```

В комментариях описано назначение различных полей и методов класса `TMemMapFile`.

Этот класс содержит методы `AllocFileHandle()`, `AllocFileMapping()` и `AllocFileView()`, которые используются для считывания дескрипторов дискового файла, объекта отображения и окна просмотра заданного файла соответственно.

Именно в конструкторе `Create()` выполняется инициализация полей и вызываются методы для назначения различных дескрипторов. При неудачном выполнении любого из этих методов возбуждается исключительная ситуация. Деструктор `Destroy()` гарантирует, что при вызове метода `UnMapViewOfFile()` отменяется окно просмотра файла.

Использование класса TMemMapFile

Функционирование главной формы, предназначенной для проекта поиска текста, реализовано с помощью кода, представленного в листинге 12.12.

Листинг 12.12. Исходный код главной формы проекта поиска текста

```
unit MainFrm;

interface

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

type

  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;
```

```
  {  Считываем каждый текстовый файл, в котором предполагается
     выполнить поиск текста. Используем при поиске последовательность
     вызовов функций FindFirst/FindeNext. }
  RetVal := FindFirst(dlbDirectories.Directory+'\ *.txt', faAnyFile, SearchRec);
  try
    while RetVal = 0 do
    begin
      FName := SearchRec.Name;

      { Открываем файл, отображенный в память, с доступом только для чтения.}
      MemMapFile := TMemMapFile.Create(FName, fmOpenRead, 0, True);
      try

        {  Используем для искомой строки временную область памяти }
        FindString := edtSearchString.Text;

        WordCount := 0;
            // Начальная установка счетчика WordCount
        {  Получаем первое вхождение искомой строки }
        FoundStr := StrPos(PChar(MemMapFile.Data), PChar(FindString));

        if FoundStr <> nil then
        begin
          {  Продолжаем поиск вхождений заданной строки в остальной части
             текста файла. При каждом обнаружении искомой строки увеличиваем
             переменную WordCount. }
          repeat
            inc(WordCount);
            inc(FoundStr, Length(FoundStr));

           {  Считываем следующее вхождение искомой строки. }
            FoundStr := StrPos(PChar(FoundStr), PChar(FindString));
          until FoundStr = nil;
          {  Добавляем имя файла в окно списка. }
          lbFilesFound.Items.Add(SearchRec.Name +
            ' - '+IntToStr(WordCount));
        end;
        {  Считываем следующий файл для выполнения поиска. }
        RetVal := FindNext(SearchRec);
      finally
        MemMapFile.Free;
        {  Освобождаем экземпляр файла, отображенного в память. }
      end;
    end;
  finally
    FindClose(SearchRec);
  end;
end;

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

В этом проекте выполняется поиск (с учетом регистра) заданной строки в текстовых файлах текущего каталога.

Процедура `btnSearchClick()` содержит код, предназначенный для выполнения реального поиска, определения числа вхождений заданной строки в каждом файле и добавления имен файлов, в которых была обнаружена искомая строка, в список `lbFilesFound`.

Сначала в этой процедуре с помощью последовательности вызовов функций `FindFirst()/FindNext()` в текущем каталоге выполняется поиск файлов с расширением `.txt`. Обе функции рассматриваются ниже в этой главе. Затем для получения доступа к данным очередного текстового файла используется класс `TMemMapFile`, причем файл открывается с доступом только для чтения, поскольку в этом проекте никакой модификации данных не предусмотрено. В следующих строках кода реализована логика, требуемая для получения числа вхождений искомой строки в рассматриваемом файле:

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

Как имя файла, так и число вхождений искомой строки в файле добавляются в список `lbFilesFound` для отображения на главной форме проекта.

Когда пользователь дважды щелкает на элементе `TListBox`, в окно примечания (элемент управления `TMemo`) загружается соответствующий файл, в котором можно найти очередное вхождение искомой строки, щелкнув на кнопке Find Next (Найти следующее).

Обработчик событий `btnFindNext()` инициализирует свойство `FindDialog.FindText`, устанавливая его равным тексту, содержащемуся в строке редактирования `edtSearchString`. Затем активизируется диалоговое окно FindDialog.

Когда пользователь щелкает на кнопке Find Next в диалоговом окне FindDialog, вызывается обработчик события `OnFind`, реализованный в виде процедуры `FindDialogFind()`. В ней используется функция `SearchMemo()`, которая определена в модуле `Search.pas`.

На заметку

Модуль `Search.pas` представляет собой файл, который поставляется с Borland Delphi 1.0 в качестве одного из демонстрационных проектов. В этом модуле не используются различные средства обработки строк, поскольку он предназначен для Delphi 1.0, но мы внесли минимальные изменения, благодаря которым курсор сможет оказаться в поле зрения элемента управления `TMemo`, что автоматически делалось в Windows 3.1. А в Win32 в элемент управления `TMemo` (после установки его свойства `SelStart`) необходимо передавать сообщение Windows `EM_SCROLLCARET`. Для получения дополнительной информации читайте комментарии, приведенные в тексте модуля `Search.pas`.

Функция `SearchMemo()` просматривает текст любого потомка класса `TCustomEdit` и выбирает нужный фрагмент текста для отображения в поле примечания.

Каталоги и устройства

В приложениях можно эффективно выполнять разнообразные задачи, связанные с устройствами, инсталлированными в системе, и каталогами этих устройств. Некоторые из этих задач рассматриваются в следующих разделах.

Получение списка доступных устройств и их типов

Для получения списка доступных устройств в системе используйте функцию Win32 API `GetDriveType()`. Она принимает параметр `PChar` и возвращает целое значение, представляющее один из типов устройств, перечисленных в табл. 12.6.

Таблица 12.6. Значения, возвращаемые функцией GetDriveType()

Возвращаемое значение	*Описание*
0	Тип устройства определить невозможно
1	Корневой каталог не существует
DRIVE_REMOVABLE	Съемное устройство
DRIVE_FIXED	Несъемное устройство
DRIVE_REMOTE	Удаленное (сетевое) устройство
DRIVE_CDROM	Компакт-диск
DRIVE_RAMDISK	Диск ОЗУ

В листинге 12.13 демонстрируется вариант использования функции `GetDriveType()`.

Листинг 12.13. Использование функции GetDriveType()

```
procedure TMainForm.btnGetDriveTypesClick(Sender: TObject);
var
  i: Integer;
  C: String;
  DType: Integer;
  DriveString: String;
begin
  {  Цикл по всем буквам A..Z для определения доступных устройств }
  for i := 65 to 90 do
  begin
    C := chr(i)+':\ ';
    // Форматируем строку для представления корневого каталога
    {  Вызываем функцию GetDriveType(), которая возвращает целое
       значение, представляющее один из типов, приведенных в
       инструкции case.  }
    DType := GetDriveType(PChar(C));
    {  На основе полученного типа устройства форматируем строку
       для добавления в список различных типов устройств. }
    case DType of
      0: DriveString := C+' The drive type cannot be determined.';
        // Нельзя определить тип устройства
      1: DriveString := C+' The root directory does not exist.';
        // Корневой каталог не существует
      DRIVE_REMOVABLE: DriveString :=
         C+' The drive can be removed from the drive.';
        // Съемное устройство
      DRIVE_FIXED: DriveString :=
         C+' The disk cannot be removed from the drive.';
        // Несъемное устройство
      DRIVE_REMOTE: DriveString :=
         C+' The drive is a remote (network) drive.';
        // Удаленное (сетевое) устройство
      DRIVE_CDROM: DriveString := C+' The drive is a CD-ROM drive.';
        // Компакт-диск
```

```
      DRIVE_RAMDISK: DriveString := C+' The drive is a RAM disk.';
        // Диск ОЗУ
    end;
    {  Добавляем только те типы, которые поддаются определению. }
    if not ((DType = 0) or (DType = 1)) then
      lbDrives.Items.AddObject(DriveString, Pointer(i));
  end;

end;
```

В листинге 12.13 представлен текст простой процедуры, в которой выполняется цикл по всем символам алфавита. Эти символы (в виде выражения для корневого каталога) по очереди передаются функции `GetDriveType()`, чтобы определить, относятся ли они к допустимым типам устройств. Если тип поддается определению, функция `GetDriveType()` возвращает соответствующий тип устройства, распознаваемый с помощью инструкции `case`. При этом создается описывающая строка, которая добавляется в окно списка вместе с символом, представляющим букву устройства в массиве списка `Objects`. Следует отметить, что в список добавляются только те устройства, которые существуют в системе и тип которых поддается определению. Кстати, Delphi 4 поставляется с компонентом `TDriveComboBox`, позволяющим выбирать дисковое устройство. Его можно найти на вкладке Win3.1 палитры компонентов.

Получение информации об устройстве

Помимо определения доступных устройств и их типов, можно также получить полезную информацию о конкретном устройстве, которая включает следующие элементы:

- число секторов на кластер;
- число байтов в секторе;
- количество свободных кластеров;
- общее число кластеров;
- общее число байтов свободного пространства на диске;
- общее число байтов на диске (размер диска).

Четыре первых элемента можно получить путем вызова функции Win32 API `GetDiskFreeSpace()` и на их основе вычислить два последних. Как использовать функцию `GetDiskFreeSpace()`, показано на примере листинга 12.14.

Листинг 12.14. Использование функции GetDiskFreeSpace()

```
procedure TMainForm.lbDrivesClick(Sender: TObject);
var
  RootPath: String;          // Хранит путь корневого каталога устройства
  SectorsPerCluster: DWord;  // Число секторов на кластер
  BytesPerSector: DWord;     // Число байтов в секторе
  NumFreeClusters: DWord;    // Число свободных кластеров
  TotalClusters: DWord;      // Общее число кластеров
  DriveByte: Byte;           // Значение байта устройства
  FreeSpace: Int64;          // Свободное пространство на диске
  TotalSpace: Int64;         // Размер диска
  DriveNum:  Integer;        // Номер диска: 1 = A, 2 = B и т.д.

begin
  with lbDrives do
  begin
    {  Преобразуем значение ASCII, используемое для буквы устройства,
       в допустимый номер устройства (1 = A, 2 = B и т.д.) путем
       вычитания числа 64 из значения ASCII. }
    DriveByte := Integer(Items.Objects[ItemIndex])-64;
    {  Сначала создаем строку пути корневого каталога. }
    RootPath := chr(Integer(Items.Objects[ItemIndex]))+':\ ';
    {  Вызываем функцию GetDiskFreeSpace для получения информации об устройстве }
    if GetDiskFreeSpace(PChar(RootPath), SectorsPerCluster,
      BytesPerSector, NumFreeClusters, TotalClusters) then
    begin
      {  При успешном выполнении этой функции обновляем метки для
         отображения информации о диске. }
      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]);
      // Получаем информацию о свободном и максимальном дисковом пространстве
      FreeSpace  := DiskFree(DriveByte);
      TotalSpace := DiskSize(DriveByte);
      lblFreeSpace.Caption := Format('%.0n', [FreeSpace*1.0]);
      {  Вычисляем общий объем дискового пространства. }
      lblTotalDiskSpace.Caption := Format('%.0n', [TotalSpace*1.0]);
    end
    else begin
      {  Заполняем метки для "пустого" отображения. }
      lblSectPerCluster.Caption := 'X';
      lblBytesPerSector.Caption := 'X';
      lblNumFreeClust.Caption := 'X';
      lblTotalClusters.Caption := 'X';
      lblFreeSpace.Caption := 'X';
      lblTotalDiskSpace.Caption := 'X';
      ShowMessage('Cannot get disk info');
        // Информацию о диске получить не удалось
    end;
  end;

end;
```

В листинге 12.14 представлен код обработчика события `OnClick` для элемента управления списком.

Как видно из листинга 12.14, когда пользователь щелкает на одном из доступных элементов окна списка `lbDrives`, для выбранного диска создается строковое представление корневого каталога, которое передается функции `GetDiskFreeSpace()`. Если функция успешно справится с задачей получения информации об устройстве, для ее отображения будут обновлены различные метки главной формы. Пример формы, разработанной для данного проекта, показан на рис. 12.3.

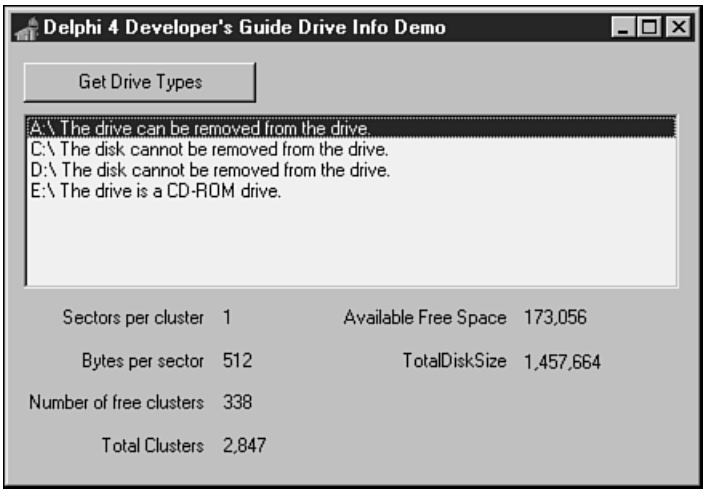

Рис. 12.3. Главная форма, отображающая информацию о диске, выбранном из списка доступных устройств

Обратите внимание, что в этом обработчике событий для определения размера диска и объема свободного пространства не используются значения, возвращаемые функцией `GetDiskFreeSpace()`. Вместо этого применяются функции `DiskFree()` и `DiskSize()`, которые определены в модуле `SysUtils.pas`. Дело в том, что функция `GetDiskFreeSpace()` не доработана в Windows 95 — она не в состоянии правильно оценить объем дисков, размер которых превышает 2 Гбайт, а для дисков размером больше 1 Гбайт сообщаются усеченные размеры секторов. Функции `DiskSize()` и `DiskFree()` используют для получения информации новые возможности Win32 API, если они доступны в операционной системе.

Получение информации о размещении каталога Windows

Для получения информации о расположении каталога Windows необходимо использовать функцию Win32 API `GetWindowsDirectory()`, которая определена следующим образом:

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

Первый параметр представляет собой буфер, в котором помещается каталог Windows в виде строки с ограничивающим нуль-символом. Второй параметр определяет размер буфера. Далее показано, как следует использовать эту функцию:

```
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;
```

На заметку

Нетрудно заметить, что в приведенном фрагменте кода после обращения к функции `GetWindowsDirectory()` добавлена следующая строка:

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

При передаче длинной строки в функцию с помощью такой операции приведения типа, как `PChar`, Delphi не в состоянии узнать, что эта строка была уже обработана, а следовательно, не может обновить информацию о ее длине. Вы должны сделать это в явном виде, использовав функцию `StrLen()` для поиска ограничивающего нуль-символа, который позволит определить длину строки, и изменив размер строки с помощью функции `SetLength()`.

Следует отметить, что использование переменной длинной строки позволило выполнить операцию приведения к типу `PChar`. Функция `GetWindowsDirectory()` возвращает целое значение, представляющее длину пути искомого каталога. В случае возникновения ошибки возвращается нулевое значение, и тогда для определения причины ошибочной ситуации нужно вызвать функцию `RaiseLastWin32Error`.

Получение информации о размещении системного каталога

Вызвав функцию Win32 API `GetSystemDirectory()`, можно также получить информацию о расположении системного каталога. Эта функция работает аналогично функции `GetWindowsDirectory()`, за исключением того, что она возвращает полный путь системного каталога, а не самого каталога Windows. Следующий фрагмент кода показывает, как использовать эту функцию:

```
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;
```

Значения, возвращаемые этой функцией, совпадают со значениями возврата функции `GetWindowsDirectory()`.

Получение имени текущего каталога

Часто возникает необходимость в получении имени текущего каталога, из которого было запущено ваше приложение. Для этого достаточно вызвать функцию Win32 API `GetCurrentDirectory()`. Если вы думаете, что эта функция действует подобно двум предыдущим, то вы совершенно правы (или почти правы). Разница лишь в том, что у нее другой порядок следования параметров. Приведенный ниже фрагмент кода иллюстрирует использование этой функции.

```
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;
```

На заметку Для аналогичных целей в Delphi предусмотрены функции `CurDir()` и `ChDir()` в модуле `System`, а также функции `GetCurrentDir()` и `SetCurrentDir()` в модуле `SysUtils.pas`.

На заметку Delphi поставляется с собственным набором процедур, предназначенных для получения частичной информации о данном файле. Например, свойство `TApplication.ExeName` содержит полный путь и имя файла выполняемого процесса. Если, к примеру, этот параметр будет иметь значение `C:\Delphi\Bin\Project.exe`, то некоторые функции при передаче им свойства `TApplication.ExeName` возвратят значения, перечисленные в табл. 12.7.

Таблица 12.7. Функции, возвращающие данные о файле или каталоге

Функция	*Результат при передаче параметра `C:\Delphi\Bin\Project.exe`*
`ExtractFileDir()`	`C:\Delphi\Bin`
`ExtractFileDrive()`	`C:`
`ExtractFileExt()`	`.exe`
`ExtractFileName()`	`Project1.exe`
`ExtractFilePath()`	`C:\Delphi\Bin\`

На заметку Для поиска заданного каталога, системных каталогов, каталогов, заданных с помощью переменной окружения `PATH`, или списка каталогов, разделенных точкой с запятой, можно использовать функцию Win32 API `SearchPath()`. К сожалению, эта функция не выполняет поиск файла среди подкаталогов данного каталога.

Поиск файла

Возможно, в один прекрасный день у вас возникнет необходимость найти в каком-то каталоге и его подкаталогах некоторые файлы (заданные маской) или выполнить с ними некоторые операции. В листинге 12.15 иллюстрируется, как это можно сделать с помощью процедуры, которая использует рекурсию для того, чтобы подкаталоги просматривались так же, как и текущий каталог.

Листинг 12.15. Пример просмотра каталогов в поисках файла

```
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;
{  Эта функция форматирует имя каталога таким образом, чтобы оно в
   качестве последнего символа содержало обратную косую черту (\). }
begin
  if Dir[Length(Dir)]<> '\ ' then
    Result := Dir+'\ '
  else
    Result := Dir;
end;

procedure TMainForm.FindFiles(APath: String);
{  Эта процедура вызывается рекурсивно для выполнения поиска
   заданного маской файла в текущем каталоге и его подкаталогах.}
var
  FSearchRec,
  DSearchRec: TSearchRec;
  FindResult: integer;

  function IsDirNotation(ADirName: String): Boolean;
  begin
    Result := (ADirName = '.') or (ADirName = '..');
  end;

begin
  APath := GetDirectoryName(APath);
    // Получаем имя каталога
    {  Находим первое вхождение заданного имени файла. }
  FindResult := FindFirst(APath+FFileName,faAnyFile+faHidden+
                          faSysFile+faReadOnly,FSearchRec);
  try
    {  Продолжаем искать файлы в соответствии с заданной маской. При
       обнаружении искомого файла добавляем в список его имя и путь.}
    while FindResult = 0 do
    begin
      lbFiles.Items.Add(LowerCase(APath+FSearchRec.Name));
      FindResult := FindNext(FSearchRec);
    end;

    {  Теперь просмотрим подкаталоги текущего каталога. Для этого
       воспользуемся функцией FindFirst для циклического просмотра
       каждого каталога, а затем снова вызовем функцию FindFiles
       (т.е. себя же). Этот рекурсивный процесс будет продолжаться
       до тех пор, пока не будут просмотрены все подкаталоги. }
```

```
    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); // Рекурсия
      FindResult := FindNext(DSearchRec);
    end;
  finally
    FindClose(FSearchRec);
  end;
end;

procedure TMainForm.btnSearchForFilesClick(Sender: TObject);
{  Этот метод запускает процесс поиска. Сначала курсор принимает
   вид песочных часов, означающих, что для выполнения поиска потребуется
   некоторое время. Затем очищается список и рекурсивно вызывается функция
   FindFiles() для просмотра подкаталогов. }
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.
```

В методе `FindFiles()` первая конструкция `while..do` выполняет поиск файлов в текущем каталоге, заданном параметром `APath`, а затем добавляет найденные файлы и их пути в список `lbFiles`. Вторая конструкция `while..do` предназначена для отыскания подкаталогов в текущем каталоге и добавления их в конец переменной `APath`. Затем метод `FindFiles()` передает самому себе параметр `APath`, содержащий уже и имя подкаталога, образуя, таким образом, рекурсивный вызов. Этот процесс продолжается до тех пор, пока не будут просмотрены все подкаталоги.

На рис. 12.4 показаны результаты поиска `.PAS`-файлов в каталоге Delphi 4.

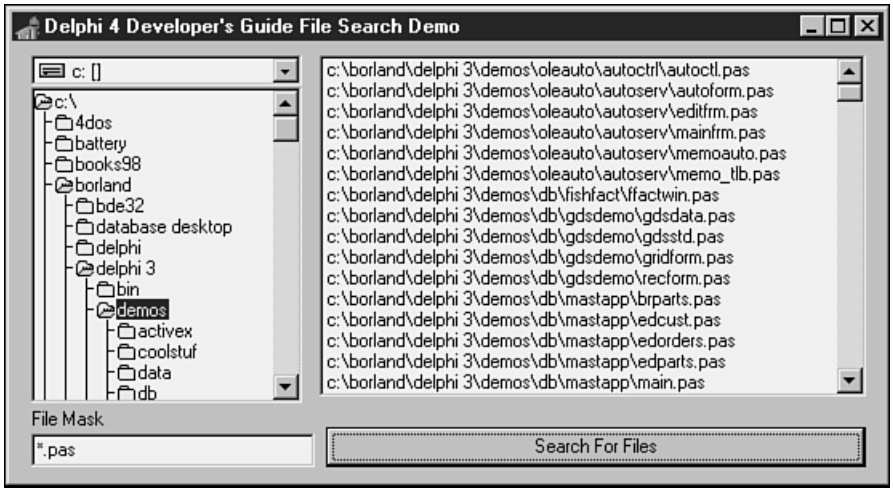

Рис. 12.4. Отображение результатов поиска файлов в каталогах

В этом проекте особого внимания заслуживают две структуры Object Pascal. Прежде всего рассмотрим структуру `TSearchRec` и функции `FindFirst()` и `FindNext()`, затем — структуру `TWin32FindData`.

Копирование и удаление дерева каталогов

До существования Win32 для копирования каталога в другое место диска нам приходилось анализировать дерево каталогов и использовать пары функций FindFirst()/FindNext(). Теперь можно прибегнуть к помощи функции Win32 ShFileOperation(), которая значительно упрощает этот процесс. Ниже приводится код, иллюстрирующий работу процедуры, которая использует функцию Win32 API ShFileOperation() для выполнения операции копирования каталога. Эта функция хорошо описана в электронной справочной системе Win32, и мы не будем дублировать эту информацию. Обратите также внимание на включение модуля ShellAPI в блок uses.

```
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;
```

Функцию ShFileOperation() также можно использовать для перемещения каталога в корзину:

```
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;
```

Запись TSearchRec

Запись TSearchRec определяет данные, необходимые для передачи функциям FindFirst() и FindNext(). Object Pascal определяет эту запись следующим образом:

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

При обнаружении искомого файла поля записи TSearchRec модифицируются вышеупомянутыми функциями.

Поле Time содержит время создания или модификации файла, а поле Size — размер файла в байтах. В поле Name хранится имя файла, а поле Attr содержит один или несколько атрибутов файла, перечисленных в табл. 12.8.

Таблица 12.8. Атрибуты файла

Атрибут	*Значение*	*Описание*
faReadOnly	$01	Файл, предназначенный только для чтения
faHidden	$02	Скрытый файл
faSysFile	$04	Системный файл
faVolumeID	$08	Метка тома
faDirectory	$10	Каталог
faArchive	$20	Архивный файл
faAnyFile	$3F	Любой файл

Поля FindHandle и ExcludeAttr используются функциями FindFirst() и FindNext() для внутренних нужд, поэтому нет необходимости вникать в их назначение.

Обе функции, FindFirst() и FindNext(), принимают путь в качестве параметра, который содержит символы шаблона (например, выражение C:\DELPHI 4\BIN*.EXE означает все файлы с расширением EXE в каталоге C:\DELPHI 4\BIN\). Параметр Attr определяет атрибуты файла, по которым следует проводить поиск. Если, например, вы хотите найти только системные файлы, следует вызывать функции FindFirst() и/или FindNext():

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

Запись TWin32FindData

Запись TWin32FindData содержит информацию о найденном файле или подкаталоге и определяется следующим образом:

```
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;
```

В табл. 12.9 описаны значения полей записи TWin32FindData.

Таблица 12.9. Значения полей записи TWin32FindData

Поле	*Значение*
dwFileAttributes	Атрибуты найденного файла. За дополнительной информацией обращайтесь к электронной справочной системе (параграф WIN32_FIND_DATA)
FtCreationTime	Время создания файла
FtLastAccessTime	Время последнего доступа к файлу
FtLastWriteTime	Время последней модификации файла
NFileSizeHigh	Старшие разряды (старшее двойное слово DWORD) размера файла в байтах. Если размер файла не превышает MAXDWORD, то это значение равно нулю

NFileSizeLow	Младшие разряды (младшее двойное слово DWORD) размера файла в байтах
DwReserved0	В данный момент не используется — зарезервировано
DwReserved1	В данный момент не используется — зарезервировано
CFileName	Имя файла в виде строки с ограничивающим нуль-символом
CAlternateFileName	Имя файла в формате 8.3, усечение длинного имени файла

Получение информации о версии файла

Из файлов EXE или DLL можно выделить информацию об их версиях. В следующих разделах описано создание класса, который инкапсулирует функции, предназначенные для выделения информации о версии, и приведен пример проекта с использованием этого класса.

Определение класса TVerInfoRes

Класс TVerInfoRes инкапсулирует три функции Win32 API для выделения информации из файлов, которые ее содержат. Речь идет о функциях GetFileVersionInfoSize(), GetFileVersionInfo() и VerQueryValue(). Информация о версии может включать такие данные, как имя компании, описание файла, номер версии и некоторые комментарии. Поскольку программирование не терпит приблизительности, ниже приводится полный список разделов, из которых состоит информация о версии файла.

Имя компании	Имя компании, создавшей файл
Комментарии	Любые дополнительные комментарии, которые характеризуют файл
Описание файла	Описание файла
Версия файла	Номер версии
Внутреннее имя	Внутреннее имя, присвоенное компанией — создателем файла
Допустимые авторские права	Все замечания об авторских правах, применимые к данному файлу
Допустимые торговые марки	Допустимые торговые марки, применимые к данному файлу
Оригинальное имя файла	Оригинальное имя файла, если таковое существует

Модуль VERINFO.PAS, в котором определяется класс TVerInfoRes, представлен в листинге 12.16.

Листинг 12.16. Исходный код модуля VERINFO.PAS, содержащего определение класса TVerInfoRes

```
unit VerInfo;

interface

uses SysUtils, WinTypes, Dialogs, Classes;

type
  {  Определяем общие классы для исключительных ситуаций,
     возникающих при получении информации о версии, а также
     при недоступности этой информации. }
  EVerInfoError   = class(Exception);
  ENoVerInfoError = class(Exception);
  eNoFixeVerInfo  = class(Exception);

  { Определяем перечислимый тип, представляющий различные
    виды информации о версии.}
  TVerInfoType =
    (viCompanyName,
     viFileDescription,
     viFileVersion,
     viInternalName,
     viLegalCopyright,
     viLegalTrademarks,
     viOriginalFilename,
```

```
     viProductName,
     viProductVersion,
     viComments);

const

  { Определяем массив строк-констант, представляющих заранее
    установленные ключевые элементы информации о версии файла.}
  VerNameArray: array[viCompanyName..viComments] of String[20] =
  ('CompanyName',
   'FileDescription',
   'FileVersion',
   'InternalName',
   'LegalCopyright',
   'LegalTrademarks',
   'OriginalFilename',
   'ProductName',
   'ProductVersion',
   'Comments');

type

  // Определям класс информации о версии
  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
  // Строки, которые должны быть переданы функции VerQueryValue()
  SFInfo                = '\ StringFileInfo\ ';
  VerTranslation: PChar = '\ VarFileInfo\ Translation';
  FormatStr             = '%s%.4x%.4x\ %s%s';


constructor TVerInfoRes.Create(AFileName: String);
begin
  FFileName := aFileName;
  FFileFlags := TStringList.Create;
  // Получаем информацию о версии файла
  FillFileVersionInfo;
  // Получаем фиксированную информацию о версии файла
  FillFixedFileInfoBuf;
  // Получаем значения масок файла
  FillFileMaskInfo;
end;
```

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

procedure TVerInfoRes.FillFileVersionInfo;
var
  SBSize: UInt;
begin
        // Определяем размер информации о версии
  Size := GetFileVersionInfoSize(PChar(FFileName), Handle);
  if Size <= 0 then
        // Если размер <= 0, генерируем исключительную ситуацию
    raise ENoVerInfoError.Create('No Version Info Available.');
        // Информация о версии недоступна
        // Устанавливаем соответствующую длину
  SetLength(RezBuffer, Size);
        // Заполняем буфер информацией о версии, а в случае
        // ошибки генерируем исключительную ситуацию
  if not GetFileVersionInfo(PChar(FFileName), Handle, Size, PChar(RezBuffer)) then
    raise EVerInfoError.Create('Cannot obtain version info.');
        // Невозможно получить информацию о версии
  if not VerQueryValue(PChar(RezBuffer), VerTranslation,  pointer(TransTable), SBSize)
then
    raise EVerInfoError.Create('No language info.');
end; // Нет информации о языке

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]);
  // Получаем и возвращаем информацию о версии,
  // в случае ошибки возвращаем пустую строку
  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]);
  // Получаем и возвращаем информацию о версии,
  // в случае ошибки возвращаем пустую строку
  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:  // То же самое, что и VOS__BASE
        Result := 'Unknown';
      VOS_DOS:
        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.
```

Класс TVerInfoRes содержит необходимые поля и инкапсулирует соответствующие методы Win32 API, требующиеся для получения информации о версии из любого файла. Файл, из которого добывается информация о версии, указывается путем передачи его имени конструктору TVerInfoRes.Create() в качестве параметра AFileName. Это имя файла присваивается полю FFileName, которое используется в другом методе для реального выделения информации о версии. Затем в этом конструкторе вызываются по очереди три метода: FillFileVersionInfo(), FillFixedFileInfoBuf() и FillFileMaskInfo().

Метод FillFileVersionInfo()

Этот метод предназначен для выполнения подготовительного этапа работы по загрузке информации о версии, после которого можно приступать к рассмотрению отдельных элементов этой информации. Прежде всего, в этом методе определяется факт наличия информации о версии, и в случае положительного результата — ее размер. Сведения о размере позволяют узнать, какой объем памяти нужно выделить для хранения этой информации. Непосредственным определением размера информации о версии, содержащейся в исследуемом файле, занимается функция Win32 API GetFileVersionInfoSize(), которая объявляется следующим образом:

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

Параметр lptstrFilename указывает на файл, из которого должна быть получена информация о версии. Параметр lpdwHandle представляет собой переменную типа DWORD, которая при вызове этой функции устанавливается равной нулю. Насколько мы могли понять, другого назначения эта переменная не имеет.

Метод FillFileVersionInfo() передает параметр FFileName функции GetFileVersionInfoSize(), и если возвращаемое ею значение, присвоенное переменной Size, оказывается больше нуля, то для хранения Size байт выделяется буфер RezBuffer.

После выделения памяти для буфера RezBuffer последний передается функции GetFileVersionInfo(), которая заполняет его информацией о версии. Функция GetFileVersionInfo() определяется следующим образом:

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

В качестве параметра lptstrFilename принимается имя файла FFileName. Параметр dwHandle игнорируется. Параметр dwLen — это не что иное, как значение, возвращаемое функцией GetFileVersionInfoSize(), которое хранится в переменной Size. Параметр lpData является указателем на буфер, выделенный для хранения информации о версии. Если считывание информации о версии не увенчается успехом, функция GetFileVersionInfo() вернет значение False; в противном случае — значение True.

И наконец, метод FillFileVersionInfo() вызывает функцию API VerQueryValue(), которая используется для возврата информации о версии, выбранной из информационного ресурса. В данном примере вызывается функция VerQueryValue(), чтобы считать указатель на массив языков и идентификаторов символьного набора. Этот массив используется в последующих обращениях к функции VerQueryValue() для получения доступа к определенной части информационного ресурса (с помощью поля TransTable), которая соответствует конкретному языку.

Функция VerQueryValue() определяется следующим образом:

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

Параметр pBlock указывает на параметр lpData, который передается функции GetFileVersionInfo(). Параметр LpSubBlock представляет собой строку с ограничивающим нулем, которая определяет, какое считывать значение информации о версии. Весьма полезно просмотреть электронную справку по функции VerQueryValue(), где описаны различные строки, допустимые для передачи функции VerQueryValue(). Для предыдущего примера, чтобы считать информацию о языке и соответствующем наборе символов, в качестве параметра lpSubBlock передается строка "\VarFileInfo\Translation". Параметр lplpBuffer указывает на буфер, в котором хранится значение информации о версии, а параметр puLen содержит длину считываемых данных.

Метод FillFixedFileInfoBuf()

В методе `FillFixedFileInfoBuf()` иллюстрируется использование функции `VerQueryValue()` для получения указателя на структуру `VS_FIXEDFILEINFO`, которая содержит информацию о файле. Это делается путем передачи функции `VerQueryValue()` в качестве параметра `lpSubBlock` строки `"\"`. Этот указатель сохраняется в поле `TVerInfoRes.FixedFileInfoBuf`.

Метод FillFileMaskInfo()

Этот метод служит иллюстрацией получения атрибутов модуля. Поставленная цель достигается здесь путем выполнения логических операций над соответствующими битовыми масками и полями `dwFileFlagsMask` и `dwFileFlags` поля `FixedFileInfoBuf` с последующей оценкой выделяемого признака. Мы не будем здесь останавливаться на значениях этих признаков, но, если вас это интересует, можно обратиться к электронной справке Delphi, которая открывает свое окно при щелчке на кнопке Help вкладки Version Info диалогового окна Project Options.

Методы GetPreDefKeyString() и GetUserDefKeyString()

В этих методах иллюстрируется использование функции `VerQueryValue()` для считывания строк информации о версии, которые являются элементами таблицы Key, расположенной во вкладке Version Info диалогового окна Project Options. По умолчанию Win32 API предоставляет 10 заранее определенных строк, которые мы поместили в массив констант `VerNameArray`. Чтобы прочитать определенную строку, нужно передать в качестве параметра `lpSubBlock` функции `VerQueryValue()` строку `"\StringFileInfo\lang-charset\string-name"`. Строка *`lang-charset`* указывает на идентификатор языка и символьного набора, который был ранее считан в методе `FillFileVersionInfo()` и к которому можно получить доступ с помощью поля `TransTable`. Строка *`string-name`* определяет одну из встроенных строковых констант в массиве `VerNameArray`. Функция `GetPreDefKeyString()` предназначена для считывания встроенных строк информации о версии.

Функция `GetUserDefKeyString()` действует аналогично функции `GetPreDefKeyString()` за исключением того, что в качестве параметра должна быть передана строка, содержащая ключ. В этом методе выполняется сборка значения строки `lpSubBlock` с использованием в качестве ключа параметра `AKey`.

Получение номеров версий

Методы `GetFileVersion()` и `GetProductVersion()` иллюстрируют способ получения номеров версий файла и продукта.

Структура `FixedFileInfoBuf` содержит поля, отведенные для номера версии как самого файла, так и номера версии продукта, с которым данный файл может распространяться. Эти номера версий хранятся в виде 64-разрядных чисел. Старшие и младшие 32-разрядные части этих чисел считываются по отдельности, и при этом используются различные поля.

Двоичный номер версии файла хранится в полях `dwFileVersionMS` и `dwFileVersionLS`, а номер версии продукта, с которым распространяется данный файл, — в полях `dwProductVersionMS` и `dwProductVersionLS`.

Строковое представление номера версии для данного файла возвращают методы `GetFileVersion()` и `GetProductVersion()`, использующие для соответствующего форматирования строки вспомогательную функцию `VersionString()`.

Получение информации об операционной системе

На примере метода `GetFileOS()` иллюстрируется способ определения того, для какой операционной системы разработан данный файл. Это выполняется путем анализа поля `dwFileOS` структуры `FixedFileInfoBuf`. Дополнительную информацию о различных значениях поля `dwFileOS` можно получить из электронной справки API для структуры `VS_FIXEDFILEINFO`.

Использование класса TVerInfoRes

Для иллюстрации использования класса `TVerInfoRes` был создан проект `VerInfo.dpr`. Исходный код главной формы этого проекта представлен в листинге 12.17.

Листинг 12.17. Исходный код главной формы проекта, демонстрирующего получение информации о версии

```
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);
  private
    VerInfoRes: TVerInfoRes;
  end;

var
  MainForm: TMainForm;

implementation

{ $R *.DFM}

procedure AddListViewItem(const aCaption, aValue: String; aData: Pointer; aLV:
TListView);
{ Этот метод используется для добавления элемента TListItem
  в список aLV типа TListView. }
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); // Автор
  AddListViewItem('File Version', VerInfoRes.FileVersion, nil,
                  lvVersionInfo); // Версия файла
  AddListViewItem('Product Version', VerInfoRes.ProductVersion, nil,
                  lvVersionInfo); // Версия продукта
  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); // Признаки файла
  AddListViewItem('Operating System', VerINfoRes.FileOS, nil, lvVersionInfo);
    // Операционная система
end;

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

end.
```

Текст этой программы не должен вызывать затруднений. Она просто выполняет отображение информации о версии для самой себя. На рис. 12.5 показан результат запуска этого проекта — отображение соответствующей информации.

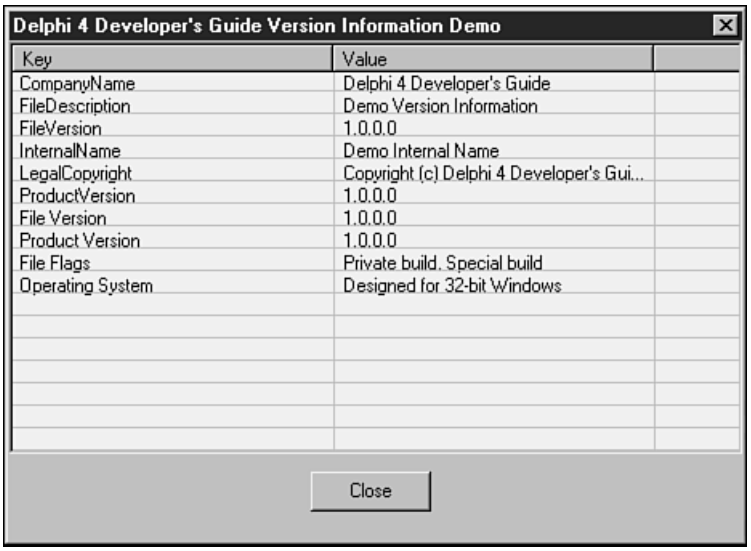

Рис. 12.5. Информация о версии демонстрационного приложения

Резюме

Эта глава содержит информацию, которой вполне достаточно для работы с файлами, каталогами и устройствами памяти (дисками). Во-первых, вы узнали о способах обработки различных типов файлов; это подготовило вас к самостоятельному созданию потомка класса Delphi `TFileStream`, в котором инкапсулированы операции ввода-вывода для работы с файлами, состоящими из записей. Вы научились использовать файлы, отображенные в память, которые являются мощным средством Win32. Для инкапсуляции функций отображения в память был создан класс `TMemMapFile`. И наконец, была продемонстрирована возможность извлечения информации о версии файлов.