Coding Standards Document

CHAPTER

6

IN THIS CHAPTER

Introduction

This document describes the coding standards for Delphi programming as used in *Delphi 5 Developer's Guide*. In general, this document follows the often "unspoken" formatting guidelines used by Borland International with a few minor exceptions. The purpose for including this document in *Delphi 5 Developer's Guide* is to present a method by which development teams can enforce a consistent style to the coding they do. The intent is to make it so that every programmer on a team can understand the code being written by other programmers. This is accomplished by making the code more readable by use of consistency.

This document by no means includes everything that might exist in a coding standard. However, it does contain enough detail to get you started. Feel free to use and modify these standards to fit your needs. We don't recommend, however, that you deviate too far from the standards used by Borland's development staff. We recommend this because as you bring new programmers to your team, the standards that they're most likely to be most familiar with are Borland's. Like most coding standards documents, this document will evolve as needed. Therefore, you'll find the most updated version online at `www.xapware.com/ddg`.

This document does not cover *user interface standards*. This is a separate but equally important topic. Enough third-party books and Microsoft documentation cover such guidelines that we decided not to replicate this information but rather refer you to the Microsoft Developers Network and other sources where that information is available.

General Source Code Formatting Rules

Indentation

Indenting shall be two spaces per level. Do not save tab characters to source files. The reason for this is because tab characters are expanded to different widths with different users' settings and by different source management utilities (print, archive, version control, and so on).

You can disable saving tab characters by turning off the Use Tab Character and Optimal Fill check boxes on the General page of the Editor Properties dialog box (accessed via Tools, Editor Options).

Margins

Margins will be set to 80 characters. In general, source shall not exceed this margin, with the exception to finish a word. However, this guideline is somewhat flexible. Wherever possible, statements that extend beyond one line shall be wrapped after a comma or an operator. When a statement is wrapped, it shall be indented two characters from the original statement line.

begin..end Pair

The `begin` statement appears on its own line. For example, the following first line is incorrect; the second line is correct:

```
for I := 0 to 10 do begin // Incorrect, begin on same line as for

for I := 0 to 10 do          // Correct, begin appears on a separate line
begin
```

An exception to this rule is when the `begin` statement appears as part of an `else` clause. Here's an example:

```
if some statement = then
begin
...
end
else begin
  SomeOtherStatement;
end;
```

The `end` statement always appears on its own line.

When the `begin` statement is not part of an `else` clause, the corresponding `end` statement is always indented to match its `begin` part.

Object Pascal

Parentheses

There shall never be white space between an open parenthesis and the next character. Likewise, there shall never be white space between a closed parenthesis and the previous character. The following example illustrates incorrect and correct spacing with regard to parentheses:

```
CallProc( AParameter );  // incorrect
CallProc(AParameter);    // correct
```

Never include extraneous parentheses in a statement. Parentheses shall only be used where required to achieve the intended meaning in source code. The following examples illustrate incorrect and correct usage:

```
if (I = 42) then         // incorrect - extraneous parentheses
if (I = 42) or (J = 42) then  // correct - parentheses required
```

Reserved Words and Key Words

Object Pascal language reserved words and key words shall always be completely lowercase.

Procedures and Functions (Routines)

Naming/Formatting

Routine names shall always begin with a capital letter and be camel-capped for readability. The following is an example of an incorrectly formatted procedure name:

```
procedure thisisapoorlyformattedroutinename;
```

This is an example of an appropriately capitalized routine name:

```
procedure ThisIsMuchMoreReadableRoutineName;
```

Routines shall be given names meaningful to their content. Routines that cause an action to occur will be prefixed with the action verb. Here's an example:

```
procedure FormatHardDrive;
```

Routines that set values of input parameters shall be prefixed with the word `set`:

```
procedure SetUserName;
```

Routines that retrieve a value shall be prefixed with the word `get`:

```
function GetUserName: string;
```

Formal Parameters

Formatting

Where possible, formal parameters of the same type shall be combined into one statement:

```
procedure Foo(Param1, Param2, Param3: Integer; Param4: string);
```

Naming

All formal parameter names shall be meaningful to their purpose and typically will be based off the name of the identifier that was passed to the routine. When appropriate, parameter names shall be prefixed with the character `A`:

```
procedure SomeProc(AUserName: string; AUserAge: integer);
```

The `A` prefix is a convention to disambiguate when the parameter name is the same as a property or field name in the class.

Ordering of Parameters

The following formal parameter ordering emphasizes taking advantage of register calling conventions calls.

Most frequently used (by the caller) parameters shall be in the first parameter slots. Less frequently used parameters shall be listed after that in left-to-right order.

Input lists shall exist before output lists in left-to-right order.

Place most generic parameters before most specific parameters in left-to-right order. For example: `SomeProc(APlanet, AContinent, ACountry, AState, ACity)`.

Exceptions to the ordering rule are possible, such as in the case of event handlers, where a parameter named `Sender` of type `TObject` is often passed as the first parameter.

Constant Parameters

When parameters of a record, array, `ShortString`, or interface type are unmodified by a routine, the formal parameters for that routine shall mark the parameter as `const`. This ensures that the compiler will generate code to pass these unmodified parameters in the most efficient manner.

Parameters of other types may optionally be marked as `const` if they're unmodified by a routine. Although this will have no effect on efficiency, it provides more information about parameter use to the caller of the routine.

Name Collisions

When using two units that each contain a routine of the same name, the routine residing in the unit appearing last in the `uses` clause will be invoked if you call that routine. To avoid these `uses` clause–dependent ambiguities, always prefix such method calls with the intended unit name. Here are two examples:

```
SysUtils.FindClose(SR);
```

and

```
Windows.FindClose(Handle);
```

Variables

Variable Naming and Formatting

Variables shall be given names meaningful to their purpose.

Loop control variables are generally given a single character name such as `I`, `J`, or `K`. It's acceptable to use a more meaningful name as well, such as `UserIndex`.

Boolean variable names must be descriptive enough so that the meanings of `True` and `False` values will be clear.

Local Variables

Local variables used within procedures follow the same usage and naming conventions for all other variables. Temporary variables shall be named appropriately.

When necessary, initialization of local variables will occur immediately upon entry into the routine. Local `AnsiString` variables are automatically initialized to an empty string, local interface and dispinterface type variables are automatically initialized to `nil`, and local `Variant` and `OleVariant` type variables are automatically initialized to `Unassigned`.

Use of Global Variables

Use of global variables is discouraged. However, they may be used when necessary. When this is the case, you're encouraged to keep global variables within the context in which they're used. For example, a global variable may be global only within the scope of a single unit's implementation section.

Global data that's intended to be used by a number of units shall be moved into a common unit used by all.

Global data may be initialized with a value directly in the `var` section. Bear in mind that all global data is automatically zero initialized; therefore, do not initialize global variables to "empty" values such as `0`, `nil`, `''`, `Unassigned`, and so on. One reason for this is because zero-initialized global data occupies no space in the EXE file. Zero-initialized data is stored in a virtual data segment that's allocated only in memory when the application starts up. Nonzero initialized global data occupies space in the EXE file on disk.

Types

Capitalization Convention

Type names that are reserved words shall be completely lowercase. Win32 API types are generally completely uppercase, and you shall follow the convention for a particular type name shown in the `Windows.pas` or other API unit. For other variable names, the first letter shall be uppercase, and the rest shall be camel-capped for clarity. Here are some examples:

```
var
  MyString: string;     // reserved word
  WindowHandle: HWND;  // Win32 API type
  I: Integer;          // type identifier introduced in System unit
```

Floating-Point Types

Use of the `Real` type is discouraged because it existed only for backward compatibility with older Pascal code. Although it's now the same as `Double`, this fact may be confusing to other developers. Use `Double` for general-purpose floating-point needs. Also, `Double` is what the processor instructions and busses are optimized for and is an IEEE-defined standard data format. Use `Extended` only when more range is required than that offered by `Double`. Extended is an Intel-specified type and is not supported in Java. Use `Single` only when the physical byte size of the floating-point variable is significant (such as when using other-language DLLs).

Enumerated Types

Names for enumerated types must be meaningful to the purpose of the enumeration. The type name must be prefixed with the `T` character to annotate it as a type declaration. The identifier list of the enumerated type must contain a lowercase two-to-three-character prefix that relates it to the original enumerated type name. Here's an example:

```
TSongType = (stRock, stClassical, stCountry, stAlternative, stHeavyMetal,
     stRB);
```

Variable instances of an enumerated type will be given the same name as the type without the `T` prefix (`SongType`) unless there's a reason to give the variable a more specific name, such as `FavoriteSongType1`, `FavoriteSongType2`, and so on.

Variant and OleVariant Types

The use of the `Variant` and `OleVariant` types is discouraged in general, but these types are necessary for programming when data types are known only at runtime, as is often the case in COM and database development. Use `OleVariant` for COM-based programming such as Automation and ActiveX controls, and use `Variant` for non-COM programming. The reason is that a `Variant` can store native Delphi strings efficiently (like a `string var`), but `OleVariant` converts all strings to OLE strings (`WideChar` strings) and are not reference counted; instead, they're always copied.

Structured Types

Array Types

Names for array types must be meaningful to the purpose for the array. The type name must be prefixed with a `T` character. If a pointer to the array type is declared, it must be prefixed with the character `P` and declared immediately prior to the type declaration. Here's an example:

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of integer;
```

When practical, variable instances of the array type shall be given the same name as the type name without the `T` prefix.

Record Types

A record type shall be given a name meaningful to its purpose. The type declaration must be prefixed with the character `T`. If a pointer to the record type is declared, it must be prefixed with the character `P` and declared immediately prior to the type declaration. The type declaration for each element may be optionally aligned in a column to the right. Here's an example:

```
type
```

```
PEmployee = ^TEmployee;

TEmployee = record
  EmployeeName: string
  EmployeeRate:  Double;
end;
```

Statements

if Statements

The most likely case to execute in an `if`/`then`/`else` statement shall be placed in the `then` clause, with less likely cases residing in the `else` clause(s).

Try to avoid chaining `if` statements and use `case` statements instead if at all possible.

Do not nest `if` statements more than five levels deep. Create a clearer approach to the code.

Do not use extraneous parentheses in an `if` statement.

If multiple conditions are being tested in an `if` statement, conditions shall be arranged from left to right in order of least to most computation intensive. This enables your code to take advantage of short-circuit Boolean evaluation logic built into the compiler. For example, if `Condition1` is faster than `Condition2`, and `Condition2` is faster than `Condition3`, then the `if` statement shall be constructed as follows:

```
if Condition1 and Condition2 and Condition3 then
```

case Statements

General Topics

The individual cases in a `case` statement shall be ordered by the case constant either numerically or alphabetically.

The actions statements of each case shall be kept simple and generally shall not exceed four to five lines of code. If the actions are more complex, the code shall be placed in a separate procedure or function.

The `else` clause of a `case` statement shall be used only for legitimate defaults or to detect errors.

Formatting

`case` statements follow the same formatting rules as other constructs in regards to indentation and naming conventions.

while Statements

The use of the `Exit` procedure to exit a `while` loop is discouraged; when possible, you shall exit the loop using only the loop condition.

All initialization code for a `while` loop shall occur directly before entering the `while` loop and shall not be separated by other nonrelated statements.

Any ending housekeeping shall be done immediately following the loop.

for Statements

`for` statements shall be used in place of `while` statements when the code must execute for a known number of increments.

repeat Statements

`repeat` statements are similar to `while` loops and shall follow the same general guidelines.

with Statements

General Topics

The `with` statement shall be used sparingly and with considerable caution. Avoid overuse of `with` statements and beware of using multiple objects, records, and so on in the `with` statement. For example,

```
with Record1, Record2 do
```

can confuse the programmer and can easily lead to difficult-to-detect bugs.

Formatting

`with` statements follow the same formatting rules in regard to naming conventions and indentation as described previously in this document.

Structured Exception Handling

General Topics

Exception handling shall be used abundantly for both error correction and resource protection. This means that in all cases where resources are allocated, a `try..finally` must be used to ensure proper deallocation of the resource. The exception to this involves cases where resources are allocated/freed in the initialization/finalization of a unit or the constructor/destructor of an object.

Use of try..finally

Where possible, each allocation shall be matched with a `try..finally` construct. For example, the following code could lead to possible bugs:

```
SomeClass1 := TSomeClass.Create;
SomeClass2 := TSomeClass.Create;
try
   { do some code }
finally
```

```
  SomeClass1.Free;
  SomeClass2.Free;
end;
```

A safer approach to the preceding allocation would be this:

```
SomeClass1 := TSomeClass.Create
try
  SomeClass2 := TSomeClass.Create;
  try
     { do some code }
  finally
    SomeClass2.Free;
  end;
finally
  SomeClass1.Free;
end;
```

Use of try..except

Use `try..except` only when you want to perform some task when an exception is raised. In general, you shall not use `try..except` to simply show an error message on the screen because that will be done automatically in the context of an application by the `Application` object. If you want to invoke the default exception handling after you've performed some task in the `except` clause, use `raise` to reraise the exception to the next handler.

Use of try..except..else

The use of the `else` clause with `try..except` is discouraged because it will block all exceptions, even those for which you may not be prepared.

Classes

Naming/Formatting

Type names for classes shall be meaningful to the purpose of the class. The type name must have the `T` prefix to annotate it as a type definition. Here's an example:

```
type
  TCustomer = class(TObject)
```

Instance names for classes will generally match the type name of the class without the `T` prefix:

```
var
  Customer: TCustomer;
```

NOTE

See the section "Component Type Naming Standards" for further information on naming components.

Fields

Naming/Formatting

Class field names follow the same naming conventions as variable identifiers, except they're prefixed with the `F` annotation to signify that they're field names.

Visibility

All fields shall be private. Fields that are accessible outside the class scope shall be made accessible through the use of a property.

Methods

Naming/Formatting

Method names follow the same naming conventions as described for procedures and functions in this document.

Use of Static Methods

Use static methods when you do not intend for a method to be overridden by descendant classes.

Use of Virtual/Dynamic Methods

Use virtual methods when you intend for a method to be overridden by descendant classes. Dynamic methods shall only be used on classes of which there will be many descendants (direct or indirect). For example, when working with a class that contains one infrequently overridden method and 100 descendent classes, you shall make the method dynamic to reduce the memory use by the 100 descendent classes.

Use of Abstract Methods

Do not use abstract methods on classes of which instances will be created. Use abstract methods only on base classes that will never be created.

Property-Access Methods

All access methods must appear in the `private` or `protected` sections of the class definition.

The naming conventions for property-access methods follow the same rules as for procedures and functions. The read accessor method (reader method) must be prefixed with the word `Get`.

The write accessor method (writer method) must be prefixed with the word `Set`. The parameter for the writer method shall have the name `Value`, and its type shall be that of the property it represents. Here's an example:

```
TSomeClass = class(TObject)
private
  FSomeField: Integer;
protected
  function GetSomeField: Integer;
  procedure SetSomeField( Value: Integer);
public
  property SomeField: Integer read GetSomeField write SetSomeField;
end;
```

Properties

Naming/Formatting

Properties that serve as accessors to private fields will be named the same as the fields they represent, without the `F` annotator.

Property names shall be nouns, not verbs. Properties represent data; methods represent actions.

Array property names shall be plural. Normal property names shall be singular.

Use of Access Methods

Although not required, it's encouraged that you use, at a minimum, a write access method for properties that represent a private field.

Files

Project Files

Naming

Project files shall be given descriptive names. For example, *The Delphi 5 Developer's Guide Bug Manager* is given the project name `DDGBugs.dpr`. A system information program shall be given a name such as `SysInfo.dpr`.

Form Files

Naming

A form file shall be given a name descriptive of the form's purpose, postfixed with the characters `Frm`. For example, an About form would have the filename `AboutFrm.dpr`, and a Main form would have the filename `MainFrm.dpr`.

Data Module Files

Naming

A data module shall be given a name that's descriptive of the data module's purpose. The name shall be postfixed with the characters `DM`. For example, a Customers data module will have the form filename `CustomersDM.dfm`.

Remote Data Module Files

Naming

A remote data module shall be given a name that's descriptive of the remote data module's purpose. The name shall be postfixed with the characters `RDM`. For example, a Customers remote data module would have the form filename `CustomersRDM.dfm`.

Unit Files

General Unit Structure

Unit Name

Unit files shall be given descriptive names. For example, the unit containing an application's main form might be called `MainFrm.pas`.

The uses Clause

A `uses` clause in the `interface` section shall only contain units required by code in the `interface` section. Remove any extraneous unit names that might have been automatically inserted by Delphi.

A `uses` clause in the `implementation` section shall only contain units required by code in the `implementation` section. Remove any extraneous unit names.

The interface Section

The `interface` section shall contain declarations for only those types, variables, procedure/function forward declarations, and so on that are to be accessible by external units. Otherwise, these declarations shall go into the `implementation` section.

The implementation Section

The `implementation` section shall contain any declarations for types, variables, procedures/functions, and so on that are private to the containing unit.

The initialization Section

Do not place time-intensive code in the `initialization` section of a unit. This will cause the application to seem sluggish upon startup.

The finalization Section

Make sure you deallocate any items you allocated in the `initialization` section.

Form Units

Naming

A unit file for a form shall be given the same name as its corresponding form file. For example, an About form would have the unit name `AboutFrm.pas`, and a Main form would have the unit filename `MainFrm.pas`.

Data Module Units

Naming

Unit files or data modules shall be given the same names as their corresponding form files. For example, a Customers data module unit would have the unit name `CustomersDM.pas`.

General-purpose Units

Naming

A general-purpose unit shall be given a name meaningful to the unit's purpose. For example, a utilities unit would be given the name `BugUtilities.pas`, and a unit containing global variables would be given the name `CustomerGlobals.pas`.

Keep in mind that unit names must be unique across all packages used by a project. Generic or common unit names are not recommended.

Component Units

Naming

Component units shall be placed in a separate directory to distinguish them as units defining components or sets of components. They shall never be placed in the same directory as the project. The unit name must be meaningful to its content.

NOTE

See the section "User-Defined Components" for further information on component-naming standards.

File Headers

Use of informational file headers is encouraged for all source files, project files, units, and so on. A proper file header must contain the following information:

```
{
Copyright © YEAR by AUTHORS
}
```

Forms and Data Modules

Forms

Form Type Naming Standard

Form types shall be given names descriptive of the form's purpose. The type definition shall be prefixed with a `T`, and a descriptive name shall follow the prefix. Finally, `Form` shall postfix the descriptive name. For example, the type name for an About form would be

```
TAboutForm = class(TForm)
```

A main form definition would be

```
TMainForm = class(TForm)
```

The customer entry form would have a name such as

```
TCustomerEntryForm = class(TForm)
```

Form Instance Naming Standard

Form instances shall be named the same as their corresponding types, without the `T` prefix. For example, for the preceding form types, the instance names are as follows:

Type Name	*Instance Name*
`TAboutForm`	`AboutForm`
`TMainForm`	`MainForm`
`TCustomerEntryForm`	`CustomerEntryForm`

Auto-creating Forms

Only the main form shall be autocreated unless there's a good reason to do otherwise. All other forms must be removed from the Autocreate list in the Project Options dialog box. See the following section for more information.

Modal Form Instantiation Functions

All form units shall contain a form-instantiation function that creates, sets up, and shows the form modally as well as frees the form. This function shall return the modal result returned by the form. Parameters passed to this function shall follow the parameter-passing standard specified in this document. This functionality is to be encapsulated in this way to facilitate code reuse and maintenance.

The form variable shall be removed from the unit and declared locally in the form-instantiation function. (Note that this requires that the form be removed from the Autocreate list in the Project Options dialog box. See "Autocreating Forms" earlier in this document.)

For example, the following unit illustrates such a function for a `GetUserData` form:

```
unit UserDataFrm;

interface

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

type
  TUserDataForm = class(TForm)
    edtUserName: TEdit;
    edtUserID: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

function GetUserData(var aUserName: String; var aUserID: Integer): Word;

implementation
{$R *.DFM}

function GetUserData(var aUserName: String; var aUserID: Integer): Word;
var
  UserDataForm: TUserDataForm;
begin
  UserDataForm := TUserDataForm.Create(Application);
  try
    UserDataForm.Caption := 'Getting User Data';
    Result := UserDataForm.ShowModal;
    if ( Result = mrOK ) then begin
      aUserName := UserDataForm.edtUserName.Text;
      aUserID   := StrToInt(UserDataForm.edtUserID.Text);
    end;
  finally
    UserDataForm.Free;
  end;
end;

end.
```

Data Modules

Data Module Naming Standard

A `DataModule` type shall be given a name descriptive of the data module's purpose. The type definition shall be prefixed with a `T`, and a descriptive name shall follow the prefix. Finally, the name shall be postfixed with the word `DataModule`. For example, the type name for a Customer data module would be something such as this:

```
TCustomerDataModule = class(TDataModule)
```

Similarly, an Orders data module might have the following name:

```
TOrdersDataModule = class(TDataModule)
```

Data Module Instance Naming Standard

Data module instances will be named the same as their corresponding types, without the `T` prefix. For example, for the preceding form types, the instance names are as follows:

Type Name	*Instance Name*
`TCustomerDataModule`	`CustomerDataModule`
`TOrdersDataModule`	`OrdersDataModule`

Packages

Use of Runtime Versus Design Packages

Runtime packages shall contain only units/components required by other components in that package. Other units containing property/component editors and other design-only code shall be placed into a design package. Registration units shall also be placed into a design package.

File Naming Standards

Packages shall be named according to the following templates:

- *iii*`lib`*vv*`.dpk` (design package)
- *iii*`std`*vv*`.dpk` (runtime package)

Here, the characters *iii* signify a three-character identifying prefix. This prefix may be used to identify a company, person, or any other identifying entity.

The characters *vv* signify a version for the package corresponding to the Delphi version for which the package is intended.

Note that the package name contains either `lib` or `std` to signify it as a runtime or design-time package.

In cases where there are both design-time and runtime packages, the files shall be named similarly. For example, packages for *Delphi 5 Developer's Guide* are named as follows:

- `DdgLib50.dpk` (design package)
- `DdgStd50.dpk` (runtime package)

Components

User-Defined Components

Component Type Naming Standards

Components shall be named similarly to classes as defined in the "Classes" section, with the exception that components are given a three-character identifying prefix. This prefix may be used to identify a company, person, or any other entity. For example, a clock component written for *Delphi 5 Developer's Guide* would be defined as follows:

```
TddgClock = class(TComponent)
```

Note that the three-character prefix is in lowercase.

Component Units

Component units shall contain only one major component. A *major component* is any component that appears on the Component Palette. Any ancillary components/objects may also reside in the same unit as the major component.

Use of Registration Units

The registration procedure for components shall be removed from the component unit and placed in a separate unit. This registration unit shall be used to register any components, property editors, component editors, experts, and so on.

Component registering shall be done only in the design packages; therefore, the registration unit shall be contained in the design package and not in the runtime package.

It's suggested that registration units be named as follows:

Xxx`Reg.pas`

Here, *Xxx* is a three-character prefix used to identify a company, person, or any other entity. For example, the registration unit for the components in the *Delphi 5 Developer's Guide* would be named `DdgReg.pas`.

Component Instance Naming Conventions

All componentsmust be given descriptive names. No components shall be left with the default names assigned to them by Delphi. Components shall be named using a variation of the

Hungarian naming convention. According to this standard, the component name shall consist of two parts: a component type prefix and qualifier name.

Component Type Prefixes

The component type prefix is a set of lower case letters that represent the component type. For example, the following are valid component type prefixes for the components specified.

TButton	btn
TEdit	edt
TSpeedButton	spdbtn
TListBox	lstbx

As shown above, the component type prefix is created by modifying the component type name (ie: TButton, TEdit) to a prefix. The following rules illustrate how to define a component type prefix:

1. Remove any "T" prefixes from the components type name. For example, "TButton" becomes "Button"
2. Remove any vowels from the name formed in step 1 with the exception of the first vowel. For example, "Button" becomes "bttn" and "edit" becomes "edt."
3. Suppress double consonants. For example, "bttn" becomes "btn."
4. If a naming conflict occurs, start adding vowels to the prefix for one of the components. For example, if a new component "TBatton" is added, it will conflict with "TButton." Therefore, the prefix for "TBatton" becomes "batn."

Component Qualifier Name

The component qualifier name shall be a descriptive of the component's purpose. For example, a TButton component with the purpose of closing a form would have the name "btnClose." A TEdit component used for editing the first name of a person would have the name "edtFirstName."

Coding Standards Document Updates

This document will be updated regularly to reflect changes and enhancements to the Object Pascal language and Visual Component Library. You can retrieve updates at `http://www.xapware.com/ddg`.

Using ActiveX Controls with Delphi

CHAPTER

7

IN THIS CHAPTER

Delphi gives you the great advantage of easily integrating industry-standard ActiveX controls (formerly known as *OCX* or *OLE controls*) into your applications. Unlike Delphi's own custom components, ActiveX controls are designed to be independent of any particular development tool. This means that you can count on many vendors to provide a variety of ActiveX solutions that open up a world of features and functionality.

ActiveX control support in 32-bit Delphi works similarly to the way VBX support works in 16-bit Delphi 1. You select an option to add new ActiveX controls from Delphi's IDE main menu or package editor, and Delphi builds an Object Pascal wrapper for the ActiveX control—which is then compiled into a package and added to the Delphi Component Palette. Once there, the ActiveX control seamlessly merges into the Component Palette along with your other VCL and ActiveX components. From that point, you're just a click and a drop away from adding an ActiveX control to any of your applications. This chapter discusses integrating ActiveX controls into Delphi, using an ActiveX control in your application, and shipping ActiveX-equipped applications.

NOTE

Delphi 1 was the last version of Delphi to support VBX (Visual Basic Extension) controls. If you have a Delphi 1 project that relies on one or more VBX controls, check with the VBX vendors to see whether they supply a comparable ActiveX solution for use in your 32-bit Delphi applications.

What Is an ActiveX Control?

ActiveX controls are custom controls for 16-bit and 32-bit Windows applications that take advantage of the COM-based OLE and ActiveX technologies. Unlike VBX controls, which are designed for 16-bit Visual Basic (and therefore share Visual Basic's limitations), ActiveX controls were designed from the ground up with application independence in mind. Roughly speaking, you can think of ActiveX controls as a merging of the easy-to-use VBX technology with the open ActiveX standard. For the purposes of this chapter, you can think of OLE and ActiveX as the same thing. If you're looking for greater distinction between these terms, take a look at Chapter 23, "COM and ActiveX."

Under the skin, an ActiveX control is really an ActiveX server that, in one package, can provide all the power of ActiveX —including all OLE functions and services, visual editing, drag and drop, and OLE Automation. Like all ActiveX servers, ActiveX controls are registered in the System Registry. ActiveX controls can be developed using a variety of products, including Delphi, Borland C++Builder, Visual C++, and Visual Basic.

Microsoft is actively promoting ActiveX controls as the choice medium for application-independent custom controls; Microsoft has stated that VBX technology will not be directly supported in the Win32 operating systems and beyond. For this reason, developers should look to ActiveX controls rather than VBX controls when developing 32-bit applications.

NOTE

For a more complete description of ActiveX control technology, see Chapter 25, "Creating ActiveX Controls."

Deciding When To Use an ActiveX Control

There are typically two reasons why you would use an ActiveX control rather than a native Delphi component. The first reason is that no Delphi component is available that fits your particular need. Because the ActiveX control market is larger than that for VCL controls, you're likely to find a greater variety of fully featured "industrial strength" controls, such as word processors, World Wide Web browsers, and spreadsheets, as ActiveX controls. The second reason you would use an ActiveX control instead of a native Delphi control is if you develop in multiple programming languages and you want to leverage your expertise in some particular control or controls across the multiple development platforms.

Although ActiveX controls integrate seamlessly into the Delphi IDE, keep in mind some inherent disadvantages to using ActiveX controls in your applications. The most obvious issue is that, although Delphi components are built directly into an application executable, ActiveX controls typically require one or more additional runtime files that must be deployed with an executable. Another issue is that ActiveX controls communicate with applications through the COM layer, whereas Delphi components communicate directly with applications and other components. This means that a well-written Delphi component typically performs better than a well-written ActiveX control. A more subtle disadvantage of ActiveX controls is that they're a "lowest common denominator" solution, so they won't exploit all the capabilities of the development tool in which they're used.

Adding an ActiveX Control to the Component Palette

The first step in using a particular ActiveX control in your Delphi application is adding that control to the Component Palette in the Delphi IDE. This places an icon for the ActiveX control on the Component Palette among your other Delphi and ActiveX controls. After you add a particular ActiveX control to the Component Palette, you can drop it on any form and use it as you would any other Delphi control.

To add an ActiveX control to the Component Palette, follow these steps:

1. Choose Component, Import ActiveX Control from the main menu. The Import ActiveX dialog box appears (see Figure 7.1).

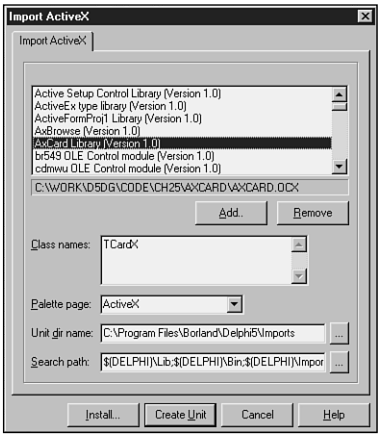

FIGURE 7.1
The Import ActiveX dialog box.

2. The Import ActiveX dialog box is divided into two parts: the top portion contains a list box of registered ActiveX controls and provides Add and Remove buttons that enable you to register and unregister controls. The bottom portion of the dialog box allows you to specify parameters for creating a Delphi component and unit that encapsulates the control.
3. If the name of the ActiveX control you want to use is listed in the top portion of the dialog box, proceed to step 4. Otherwise, click the Add button to register a new control with the system. Clicking the Add button invokes the Register OLE Control dialog box (see Figure 7.2). Select the name of the OCX or DLL file that represents the ActiveX control you want to add to the system and click the Open button. This registers the selected ActiveX control with the System Registry and dismisses the Register OLE Control dialog box.
4. In the upper portion of the Import ActiveX dialog box, select the name of the ActiveX control you want to add to the Component Palette. The lower portion of the dialog box contains edit controls for unit directory name, palette page, and search path as well as a memo control that lists the classes contained within the OCX file. The pathname shown in the Unit Dir Name edit box is the pathname of the Delphi wrapper component created to interface with the ActiveX control. The filename defaults to the same name as the

OCX file (with a `.pas` extension); the path defaults to the `\Delphi5\Imports` subdirectory. Although the default is fine to use, you can edit the directory path to your liking.

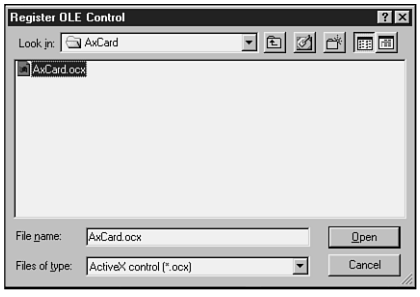

FIGURE 7.2
The Register OLE Control dialog box.

5. The Palette Page edit control in the Import ActiveX dialog box contains the name of the page on the Component Palette where you want this control to reside. The default is the ActiveX page. You can choose another existing page; alternatively, if you make up a new name, a corresponding page is created on the Component Palette.
6. The Class Names memo control in the Import ActiveX dialog box contains the names of the new objects created in this control. You should normally leave these names set to the default unless you have a specific reason for doing otherwise. For example, such a reason would be if the default class name conflicts with another component already installed in the IDE.
7. At this point, you can click either the Install or the Create Unit button in the Import ActiveX dialog box. The Create Unit button will generate the source code for the unit for the ActiveX control component wrapper. The Install button will generate the wrapper code and then invoke the Install dialog box, which allows you to choose a package into which you may install the component (see Figure 7.3).

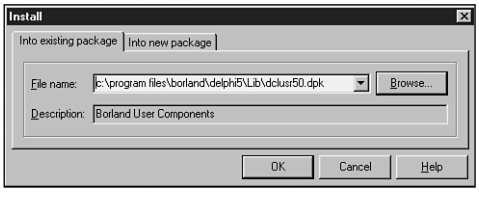

FIGURE 7.3
The Install dialog box.

8. In the Install dialog box, you can choose to add the control to an existing package or create a new package that will be installed to the Component Palette. Click OK in this dialog box, and the component will be installed to the palette.

Now your ActiveX control is on the Component Palette and ready to roll.

The Delphi Component Wrapper

Now is a good time to look into the Object Pascal wrapper created to encapsulate the ActiveX control. Doing so can help shed some light on how Delphi's ActiveX support works so that you can understand the capabilities and limitations inherent in ActiveX controls. Listing 7.1 shows the `Card_TLB.pas` unit generated by Delphi; this unit encapsulates the `AxCard.ocx` ActiveX control.

NOTE

`AxCard.ocx` is an ActiveX control developed in Chapter 25, "Creating ActiveX Controls."

LISTING 7.1 The Delphi Component Wrapper Unit for `AxCard.ocx`.

```
unit AxCard_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be
// regenerated and all manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// File generated on 8/24/99 9:24:19 AM from Type Library described below


// ************************************************************************ //
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly created
// via a function call prior to any access via the property. These items
```

```
// have been disabled in order to prevent accidental use from within the
// object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
// created via a method of the appropriate CoClass before they can be
// used.
// ************************************************************* //
// Type Lib: C:\work\d5dg\code\Ch25\AxCard\AxCard.tlb (1)
// IID\LCID: {7B33D940-0A2C-11D2-AE5C-04640BC10000}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************* //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked
                    // pointers.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  AxCardMajorVersion = 1;
  AxCardMinorVersion = 0;

  LIBID_AxCard: TGUID = '{7B33D940-0A2C-11D2-AE5C-04640BC10000}';

  IID_ICardX: TGUID = '{7B33D941-0A2C-11D2-AE5C-04640BC10000}';
  DIID_ICardXEvents: TGUID = '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
  CLASS_CardX: TGUID = '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';

// ************************************************************************//
// Declaration of Enumerations defined in Type Library
// ************************************************************************//
// Constants for enum TxDragMode
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;
```

continues

LISTING 7.1 Continued

```
// Constants for enum TxCardSuit
type
  TxCardSuit = TOleEnum;
const
  csClub = $00000000;
  csDiamond = $00000001;
  csHeart = $00000002;
  csSpade = $00000003;

// Constants for enum TxCardValue
type
  TxCardValue = TOleEnum;
const
  cvAce = $00000000;
  cvTwo = $00000001;
  cvThree = $00000002;
  cvFour = $00000003;
  cvFive = $00000004;
  cvSix = $00000005;
  cvSeven = $00000006;
  cvEight = $00000007;
  cvNine = $00000008;
  cvTen = $00000009;
  cvJack = $0000000A;
  cvQueen = $0000000B;
  cvKing = $0000000C;

// Constants for enum TxMouseButton
type
  TxMouseButton = TOleEnum;
const
  mbLeft = $00000000;
  mbRight = $00000001;
  mbMiddle = $00000002;

// Constants for enum TxAlignment
type
  TxAlignment = TOleEnum;
const
  taLeftJustify = $00000000;
  taRightJustify = $00000001;
  taCenter = $00000002;

// Constants for enum TxBiDiMode
type
```

```
  TxBiDiMode = TOleEnum;
const
  bdLeftToRight = $00000000;
  bdRightToLeft = $00000001;
  bdRightToLeftNoAlign = $00000002;
  bdRightToLeftReadingOnly = $00000003;

type

// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
  ICardX = interface;
  ICardXDisp = dispinterface;
  ICardXEvents = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
  CardX = ICardX;


// *********************************************************************//
// Interface: ICardX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// *********************************************************************//
  ICardX = interface(IDispatch)
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    function  Get_BackColor: OLE_COLOR; safecall;
    procedure Set_BackColor(Value: OLE_COLOR); safecall;
    function  Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function  Get_DragCursor: Smallint; safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    function  Get_DragMode: TxDragMode; safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    function  Get_FaceUp: WordBool; safecall;
    procedure Set_FaceUp(Value: WordBool); safecall;
    function  Get_ParentColor: WordBool; safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    function  Get_Suit: TxCardSuit; safecall;
    procedure Set_Suit(Value: TxCardSuit); safecall;
    function  Get_Value: TxCardValue; safecall;
    procedure Set_Value(Value: TxCardValue); safecall;
```

continues

LISTING 7.1 Continued

```
    function  Get_DoubleBuffered: WordBool; safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure FlipChildren(AllLevels: WordBool); safecall;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
    function  Get_Enabled: WordBool; safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    function  GetControlsAlignment: TxAlignment; safecall;
    procedure InitiateAction; safecall;
    function  IsRightToLeft: WordBool; safecall;
    function  UseRightToLeftAlignment: WordBool; safecall;
    function  UseRightToLeftReading: WordBool; safecall;
    function  UseRightToLeftScrollBar: WordBool; safecall;
    function  Get_BiDiMode: TxBiDiMode; safecall;
    procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
    function  Get_Visible: WordBool; safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    function  Get_Cursor: Smallint; safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    function  ClassNameIs(const Name: WideString): WordBool; safecall;
    procedure AboutBox; safecall;
    property BackColor: OLE_COLOR read Get_BackColor write Set_BackColor;
    property Color: OLE_COLOR read Get_Color write Set_Color;
    property DragCursor: Smallint read Get_DragCursor write
      Set_DragCursor;
    property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
    property FaceUp: WordBool read Get_FaceUp write Set_FaceUp;
    property ParentColor: WordBool read Get_ParentColor write
      Set_ParentColor;
    property Suit: TxCardSuit read Get_Suit write Set_Suit;
    property Value: TxCardValue read Get_Value write Set_Value;
    property DoubleBuffered: WordBool read Get_DoubleBuffered write
      Set_DoubleBuffered;
    property Enabled: WordBool read Get_Enabled write Set_Enabled;
    property BiDiMode: TxBiDiMode read Get_BiDiMode write Set_BiDiMode;
    property Visible: WordBool read Get_Visible write Set_Visible;
    property Cursor: Smallint read Get_Cursor write Set_Cursor;
  end;

// ******************************************************************//
// DispIntf:  ICardXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {7B33D941-0A2C-11D2-AE5C-04640BC10000}
// ******************************************************************//
  ICardXDisp = dispinterface
```

```
    ['{7B33D941-0A2C-11D2-AE5C-04640BC10000}']
    property BackColor: OLE_COLOR dispid 1;
    property Color: OLE_COLOR dispid -501;
    property DragCursor: Smallint dispid 2;
    property DragMode: TxDragMode dispid 3;
    property FaceUp: WordBool dispid 4;
    property ParentColor: WordBool dispid 5;
    property Suit: TxCardSuit dispid 6;
    property Value: TxCardValue dispid 7;
    property DoubleBuffered: WordBool dispid 10;
    procedure FlipChildren(AllLevels: WordBool); dispid 11;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; dispid 14;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; dispid 15;
    property Enabled: WordBool dispid -514;
    function  GetControlsAlignment: TxAlignment; dispid 16;
    procedure InitiateAction; dispid 18;
    function  IsRightToLeft: WordBool; dispid 19;
    function  UseRightToLeftAlignment: WordBool; dispid 24;
    function  UseRightToLeftReading: WordBool; dispid 25;
    function  UseRightToLeftScrollBar: WordBool; dispid 26;
    property BiDiMode: TxBiDiMode dispid 27;
    property Visible: WordBool dispid 28;
    property Cursor: Smallint dispid 29;
    function  ClassNameIs(const Name: WideString): WordBool; dispid 33;
    procedure AboutBox; dispid -552;
  end;

// ******************************************************************//
// DispIntf:  ICardXEvents
// Flags:     (4096) Dispatchable
// GUID:      {7B33D943-0A2C-11D2-AE5C-04640BC10000}
// ******************************************************************//
  ICardXEvents = dispinterface
    ['{7B33D943-0A2C-11D2-AE5C-04640BC10000}']
    procedure OnClick; dispid 1;
    procedure OnDblClick; dispid 2;
    procedure OnKeyPress(var Key: Smallint); dispid 7;
  end;


// ******************************************************************//
// OLE Control Proxy class declaration
// Control Name     : TCardX
// Help String      : CardX Control
// Default Interface: ICardX
// Def. Intf. DISP? : No
```

continues

LISTING 7.1 Continued

```
// Event   Interface: ICardXEvents
// TypeFlags        : (34) CanCreate Control
// ********************************************************************//
  TCardXOnKeyPress = procedure(Sender: TObject; var Key: Smallint) of
    object;

  TCardX = class(TOleControl)
  private
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TCardXOnKeyPress;
    FIntf: ICardX;
    function  GetControlInterface: ICardX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    procedure FlipChildren(AllLevels: WordBool);
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer;
    function  GetControlsAlignment: TxAlignment;
    procedure InitiateAction;
    function  IsRightToLeft: WordBool;
    function  UseRightToLeftAlignment: WordBool;
    function  UseRightToLeftReading: WordBool;
    function  UseRightToLeftScrollBar: WordBool;
    function  ClassNameIs(const Name: WideString): WordBool;
    procedure AboutBox;
    property  ControlInterface: ICardX read GetControlInterface;
    property  DefaultInterface: ICardX read GetControlInterface;
    property DoubleBuffered: WordBool index 10 read GetWordBoolProp write
      SetWordBoolProp;
    property Enabled: WordBool index -514 read GetWordBoolProp write
      SetWordBoolProp;
    property BiDiMode: TOleEnum index 27 read GetTOleEnumProp write
      SetTOleEnumProp;
    property Visible: WordBool index 28 read GetWordBoolProp write
      SetWordBoolProp;
  published
    property  TabStop;
    property  Align;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
```

```
    property  OnDragDrop;
    property  OnDragOver;
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
    property BackColor: TColor index 1 read GetTColorProp write
      SetTColorProp stored False;
    property Color: TColor index -501 read GetTColorProp write
      SetTColorProp stored False;
    property DragCursor: Smallint index 2 read GetSmallintProp write
      SetSmallintProp stored False;
    property DragMode: TOleEnum index 3 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property FaceUp: WordBool index 4 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property ParentColor: WordBool index 5 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property Suit: TOleEnum index 6 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Value: TOleEnum index 7 read GetTOleEnumProp write
      SetTOleEnumProp stored False;
    property Cursor: Smallint index 29 read GetSmallintProp write
      SetSmallintProp stored False;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnKeyPress: TCardXOnKeyPress read FOnKeyPress write
      FOnKeyPress;
  end;

procedure Register;

implementation

uses ComObj;

procedure TCardX.InitControlData;
const
  CEventDispIDs: array [0..2] of DWORD = (
    $00000001, $00000002, $00000007);
  CControlData: TControlData2 = (
    ClassID: '{7B33D945-0A2C-11D2-AE5C-04640BC10000}';
    EventIID: '{7B33D943-0A2C-11D2-AE5C-04640BC10000}';
    EventCount: 3;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil (*HR:$00000000*);
```

LISTING 7.1 Continued

```
    Flags: $00000009;
    Version: 401);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs :=
    Cardinal(@@FOnClick) - Cardinal(Self);
end;

procedure TCardX.CreateControl;

  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as ICardX;
  end;

begin
  if FIntf = nil then DoCreate;
end;

function TCardX.GetControlInterface: ICardX;
begin
  CreateControl;
  Result := FIntf;
end;

procedure TCardX.FlipChildren(AllLevels: WordBool);
begin
  DefaultInterface.FlipChildren(AllLevels);
end;

function  TCardX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlags(Flags);
end;

function  TCardX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlagsReadingOnly;
end;

function  TCardX.GetControlsAlignment: TxAlignment;
begin
  Result := DefaultInterface.GetControlsAlignment;
end;
```

```
procedure TCardX.InitiateAction;
begin
  DefaultInterface.InitiateAction;
end;

function  TCardX.IsRightToLeft: WordBool;
begin
  Result := DefaultInterface.IsRightToLeft;
end;

function  TCardX.UseRightToLeftAlignment: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftAlignment;
end;

function  TCardX.UseRightToLeftReading: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftReading;
end;

function  TCardX.UseRightToLeftScrollBar: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftScrollBar;
end;

function  TCardX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := DefaultInterface.ClassNameIs(Name);
end;

procedure TCardX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;

procedure Register;
begin
  RegisterComponents('ActiveX',[TCardX]);
end;

end.
```


Now that you've seen the code generated by the type library editor, let's look a little deeper at the type library import mechanism.

Where Do Wrapper Files Come From?

The first thing you might notice is that the filename ends in `_TLB`. More subtly, you might have caught on to the fact that there are several references to "library" in the generated source file. Both of these are clues as to the origin of the wrapper file: the control's type library. An ActiveX control's type library is special information linked to the control as a resource that describes the different elements of an ActiveX control. In particular, type libraries contain information such as the interfaces supported by a control, the properties, methods, and events of a control, and the enumerated types used by the control. The first entry in the wrapper file is the GUID of the control's type library.

NOTE

Type libraries are used more generally for any type of Automation object. Chapter 23, "COM and ActiveX," contains more information on type libraries and their use.

Enumerations

Looking at the generated unit from the top down, immediately following the type library GUID, are the enumerated types used by the control. Notice that the enumerations are declared as simple constants rather than true enumerated types. This is done because type library enumerations, like those in the C language, do not need to start at zero, and the element ordinals do not need to be contiguous. Because this type of declaration isn't legal in Object Pascal, the enumerations must be declared as constants.

Control Interfaces

Next in the wrapper file, the control's primary interface is declared. Here, you'll find all the properties and methods of the ActiveX control. The properties are also redeclared in a `dispinterface`, thus allowing the control to be used as a dual interface. The events are declared separately next in a `dispinterface`. You definitely don't need to know about interfaces to use ActiveX controls in your applications. What's more, working with interfaces can be a complicated topic, so we won't go into too much detail right now. You'll find more information on interfaces in general in Chapter 23, "COM and ActiveX," and information on interfaces with ActiveX controls in Chapter 25, "Creating ActiveX Controls."

TOleControl Descendant

Next in the unit file comes the class definition for the control wrapper. By default, the name of the ActiveX control wrapper object is `TXX`, where *`X`* is the name of the control's `coclass` in the type library. This object, like all ActiveX control wrappers, descends from the `TOleControl`

class. `TOleControl` is a window handle–bearing component that descends from `TWinControl`. `TOleControl` encapsulates the complexities of mapping the functionality of ActiveX controls to Delphi components so that ActiveX controls work seamlessly from Delphi. `TOleControl` is an *abstract class*—meaning that you never want to create an instance of one but instead use it as a starting place for other classes.

The Methods

The first procedure shown is the `InitControlData()` procedure. This procedure is introduced in the `TOleControl` object and is overridden for all descendants. It sets up the unique OLE class and event identification numbers in addition to other control-specific information. In particular, this method makes the `TOleControl` aware of important ActiveX control details such as class IDs, control miscellaneous flags, and a license key if the control is licensed. This method is found in the `protected` part of the class definition because it's not useful to users of the class—it only has meaning internal to the class.

The `InitControlInterface()` method is overridden to initialize the private `FIntf` interface field with a pointer to the control's `ICardsX` interface.

The `CardX` ActiveX control exposes only one other method: `AboutBox()`. It's standard for ActiveX controls to contain a method called `AboutBox()` that invokes a custom About dialog box for the control. This method is called through the `vTable` interface using the `ControlInterface` property, which is read from the `FIntf` field.

NOTE

In addition to `vTable` calls using the `ControlInterface` property, `Control` methods can also be invoked via Automation using `TOleControl`'s `OleObject` property. As you'll learn in Chapter 23, "COM and ActiveX," it's usually more efficient to call methods via the `vTable` rather than through Automation.

The Properties

You might have noticed that the `TCardX` class has two distinct groups of properties. One group does not have read and write values specified. These are standard Delphi component properties and events inherited from the `TWinControl` and `TComponent` ancestor classes. The other group of properties all have an index as well as get and set methods specified. This group of properties includes the ActiveX control properties being encapsulated by the `TOleControl`.

The specialized get and set methods for the encapsulated properties provide the magic that bridges the gap between the ActiveX control properties and the Object Pascal component properties. Notice the read and write methods that get and set properties for every specific type

(such as `GetBoolProp()`, `SetBoolProp()`, `GetStringProp()`, `SetStringProp()`, and so on). Although there are get and set methods for each property type, they all operate similarly. In fact, the following code shows generic get and set methods for `TOleControl` that would work given a property of type *`X`*:

```
function TOleControl.GetXProp(Index: Integer): X;
var
  Temp: TVarData;
begin
  GetProperty(Index, Temp);
  Result := Temp.VX;
end;

procedure TOleControl.SetXProp(Index: Integer; Value: X);
var
  Temp: TVarData;
begin
  Temp.VType := varX;
  Temp.VX := Value;
  SetProperty(Index, Temp);
end;
```

In this code, the index of the property (as indicated by the `index` directive in the properties of the `TCardsCtrl` component) is implicitly passed to the procedures. The variable of type *`X`* is packaged into a `TVarData` (a record that represents a Variant) data record, and those parameters are passed to the `GetProperty()` or `SetProperty()` method of `TOleControl`. Each property of the ActiveX control has a unique index that acts as an identifier. Using this index and the `TVarData` variable `Temp`, `GetProperty()` and `SetProperty()` use OLE Automation to get and set the property values inside the ActiveX control.

If you've worked with other development packages before, you'll appreciate that Delphi provides easy access not only to the ActiveX control's own properties but also to normal `TWinControl` properties and methods. This enables you to use an ActiveX control like other handle-bearing controls in Delphi and makes it possible for you to use object-oriented principles to override the behavior of an ActiveX control by creating customized descendants of ActiveX controls in the Delphi environment.

Using ActiveX Controls in Your Applications

After you link your ActiveX control wrapper into the component library, you've actually fought most of the battle. After an ActiveX control has been placed on the Component Palette, its usage is much the same as that of a regular Delphi component. Figure 7.4 shows the Delphi environment with a `TCardX` focused in the Form Designer. Notice the `TCardX` properties listed in the Object Inspector.

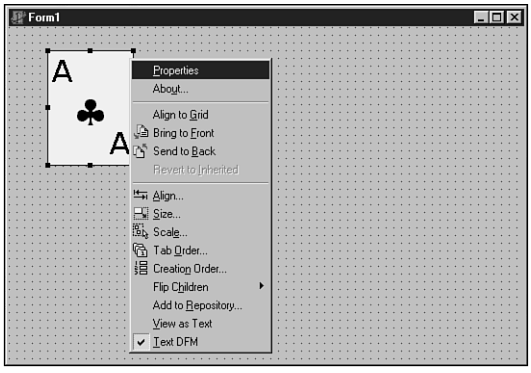

FIGURE 7.4
Working with an ActiveX control in Delphi.

In addition to an ActiveX control's properties being set with the Object Inspector, some ActiveX controls also provide a Properties dialog box that's revealed by selecting the Properties option from the context menu in the Delphi Form Designer. The context menu, also shown in Figure 7.4, is revealed by right-clicking over a particular control. The Properties dialog box actually lives within the ActiveX control; its look, feel, and contents are determined entirely by the control designer. Figure 7.5 shows the Properties dialog box for the `TCardX` ActiveX control.

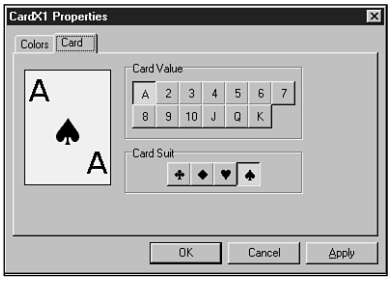

FIGURE 7.5
The `TCardX` *Properties dialog box.*

As you can imagine, this particular control comes equipped with properties that enable you to specify card suit, value, color and picture for card back as well as the standard properties that refer to position, tab order, and so on. The card in Figure 7.4 has its `Value` property set to `1 (Ace)` and its `Suit` property set to `3 (Spades)`.

Shipping ActiveX Control–Equipped Applications

When you're ready to ship your ActiveX control–equipped application, there are some deployment issues to bear in mind as you prepare to send your ActiveX control and associated files to your customers:

- You must ship the OCX or DLL file that contains the ActiveX controls you're using in your application. OCX files, being DLLs, are not linked into your application's executable. Additionally, before the user can use your application, the ActiveX control must be registered in that user's System Registry. ActiveX control registration is discussed in the following section.
- Some ActiveX controls require one or more external DLLs or other files to operate. Check the documentation for your third-party ActiveX controls to determine whether any additional files must be deployed with your ActiveX control. See Chapter 25, "Creating ActiveX Controls," for information on what additional files might need to be deployed along with your Delphi-written controls.
- Many ActiveX controls come with a license file that's required if you want to use the control at design time. This file comes from the ActiveX control vendor, and it prevents your end users from designing applications with ActiveX controls you ship with your applications. You should not ship these LIC files with your application unless you intend for users of your application to use the licensed controls in a development tool and you have the appropriate license for such redistribution.

ActiveX Control Registration

Before an ActiveX control can be used on any system (including those of customers or clients who run your applications), it must be registered with the System Registry. Most commonly, this is accomplished using the `RegSvr32.exe` application, which comes with most versions of Windows. Alternatively, you can use the `TRegSvr.exe` command-line registration utility found in the Delphi `bin` directory. Occasionally, you might want to register the control more transparently to give your application an integrated feel. Luckily, it's not difficult to integrate ActiveX control registration (and unregistration) into your application. Inprise provides the source code for the `TRegSvr` utility as a sample application, and it provides an excellent demonstration for how to register ActiveX servers and type libraries.

BlackJack: An OCX Application Example

The best way to demonstrate how to use an ActiveX control in an application is to show you how to write a useful application that incorporates an ActiveX control. This example uses the `TCardX` ActiveX control; what better way to demonstrate a card control than to make a blackjack game? For the sake of argument, assume that all programmers are high rollers and don't

need to be told the rules of the game (didn't know this book was a comedy, did you?). This way, you can concentrate on the programming job at hand.

As you can imagine, most of the code for this application deals with the logic of the game of blackjack. All the code is provided in the listings later in this chapter; right now, the discussion is narrowed to the individual portions of code that deal directly with managing and manipulating the ActiveX controls. The name of this project is `BJ`; to give you an idea of where the code comes from, Figure 7.6 shows a game of DDG BlackJack in progress.

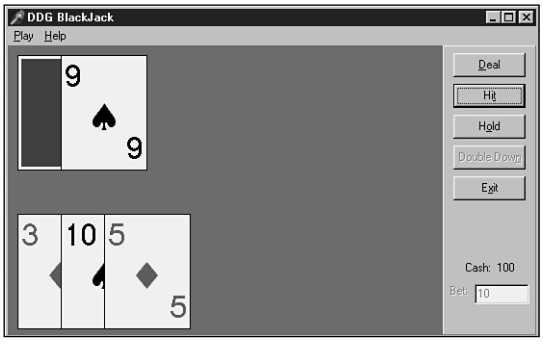

FIGURE 7.6
Playing DDG BlackJack.

The Card Deck

Before writing the game itself, you must first write an object that encapsulates a deck of playing cards. Unlike a real card deck (in which cards are picked from the top of a scrambled deck), this card deck object contains an unscrambled card deck and uses pseudorandom numbers to pick a random card from the unscrambled deck. This is possible because each card has a notion of whether it has been used. This greatly simplifies the shuffle procedure, too, because all the object has to do is set each of the cards to unused. The code for the `PlayCard.pas` unit, which contains the `TCardDeck` object, is shown in Listing 7.2.

LISTING 7.2 The `PlayCard.pas` Unit

```
unit PlayCard;

interface

uses SysUtils, Cards;

type
```

continues

LISTING 7.2 Continued

```
  ECardError = class(Exception);  // generic card exception

  TPlayingCard = record           // represents one card
    Face: TCardValue;             // card face value
    Suit: TCardSuit;              // card suit value
  end;

  { an array of 52 cards representing one deck }
  TCardArray = array[1..52] of TPlayingCard;

  { Object which represents a deck of 52 UNIQUE cards. }
  { This is a scrambled deck of 52 cards, and the      }
  { object keeps track of how far throughout the deck  }
  { the user has picked. }
  TCardDeck = class
  private
    FCardArray: TCardArray;
    FTop: integer;
    procedure InitCards;
    function GetCount: integer;
  public
    property Count: integer read GetCount;
    constructor Create; virtual;
    procedure Shuffle;
    function Draw: TPlayingCard;
  end;

{ GetCardValue returns the numeric value of any card }
function GetCardValue(C: TPlayingCard): Integer;

implementation

function GetCardValue(C: TPlayingCard): Integer;
{ returns a card's numeric value }
begin
  Result := Ord(C.Face) + 1;
  if Result > 10 then Result := 10;
end;

procedure TCardDeck.InitCards;
{ initializes the deck by assigning a unique value/suit combination }
{ to each card. }
var
  i: integer;
```

```
  AFace: TCardValue;
  ASuit: TCardSuit;
begin
  AFace := cvAce;                        // start with ace
  ASuit := csClub;                       // start with clubs
  for i := 1 to 52 do                    // for each card in deck...
  begin
    FCardArray[i].Face := AFace;         // assign face
    FCardArray[i].Suit := ASuit;         // assign suit
    if (i mod 4 = 0) and (i <> 52) then  // every four cards...
      inc(AFace);                        // increment the face
    if ASuit <> High(TCardSuit) then     // always increment the suit
      inc(ASuit)
    else
      ASuit := Low(TCardSuit);
  end;
end;

constructor TCardDeck.Create;
{ constructor for TCardDeck object. }
begin
  inherited Create;
  InitCards;
  Shuffle;
end;

function TCardDeck.GetCount: integer;
{ Returns a count of unused cards }
begin
  Result := 52 - FTop;
end;

procedure TCardDeck.Shuffle;
{ Re-mixes cards and sets top card to 0. }
var
  i: integer;
  RandCard: TPlayingCard;
  RandNum: integer;
begin
  for i := 1 to 52 do
  begin
    RandNum := Random(51) + 1;            // pick random number
    RandCard := FCardArray[RandNum];      // swap next card with
    FCardArray[RandNum] := FCardArray[i]; // random card in deck
    FCardArray[i] := RandCard;
  end;
```

continues

LISTING 7.2 Continued

```
  FTop := 0;
end;

function TCardDeck.Draw: TPlayingCard;
{ Picks the next card from the deck. }
begin
  inc(FTop);
  if FTop = 53 then
    raise ECardError.Create('Deck is empty');
  Result := FCardArray[FTop];
end;

initialization
  Randomize;                    // must seed random number generator
end.
```

The Game

The DDG BlackJack game's interaction with the `TCardX` object occurs primarily in three procedures. One procedure, called `Hit()`, is called when the player chooses to accept another card. Another procedure, `DealerHit()`, is called when the dealer wants another card. Finally, the `FreeCards()` procedure is called to dispose of all the cards on the screen to prepare for another hand to be dealt.

The `Hit()` procedure works with the `TCardX` ActiveX control uniquely in a couple ways. First, it creates all the controls dynamically rather than using controls dropped from the Component Palette. Also, it never uses an instance variable of type `TCardX`—instead, it takes advantage of a `with..do` construct to create and use the object in one step. The following code shows the `Hit()` procedure:

```
procedure TMainForm.Hit;
{ Player hit }
begin
  CurCard := CardDeck.Draw;              // draw card
  with TCardX.Create(Self) do            // create card OCX
  begin
    Left := NextPlayerPos;               // set position
    Top := PYPos;
    Suit := Ord(CurCard.Suit);           // set suit
    Value := Ord(CurCard.Face);          // set value
    Parent := Self;                      // assign parent
    Inc(NextPlayerPos, Width div 2);     // track position
    Update;                              // Display card
```

```
  end;
  DblBtn.Enabled := False;               // hit disables double down
  if CurCard.Face = cvAce then PAceFlag := True;  // set ace flag
  Inc(PlayerTotal, GetCardValue(CurCard));        // keep running total
  PlayLbl.Caption := IntToStr(PlayerTotal);        // cheat
  if PlayerTotal > 21 then                        // track bust
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;
```


In this procedure, a random card, called `CurCard`, is drawn from a `TCardDeck` object called `CardDeck`. A `TCardX` ActiveX control is then created, and property values are assigned. `NextPlayerPos` is a variable that keeps track of the position on the X-axis for the next card. `PYPos` is a constant that dictates the Y-axis position of the player hand. The `Suit` and `Value` properties are assigned values that correspond to the `Suit` and `Face` of `CurCard`. `MainForm` is assigned to be the `Parent` of the control, and the `NextPlayerPos` variable is incremented by half the width of a card. After all that, the `PlayerTotal` variable is incremented by the card value to keep track of the player's score.

The `DealerHit()` procedure works similarly to the `Hit()` procedure. The code from that procedure is shown here:

```
procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Dealer takes a hit }
begin
  CurCard := CardDeck.Draw;              // dealer draws a card
  with TCardX.Create(Self) do            // create the ActiveX control
  begin
    Left := NextDealerPos;               // place card on form
    Top := DYPos;
    Suit := Ord(CurCard.Suit);           // assign suit
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face);          // assign face
    Parent := Self;                      // assign parent for OCX
    Inc(NextDealerPos, Width div 2);     // track where to place next card
    Update;                              // Display card
  end;
  if CurCard.Face = cvAce then DAceFlag := True;   // set Ace flag
  Inc(DealerTotal, GetCardValue(CurCard));         // keep count
  DealLbl.Caption := IntToStr(DealerTotal);        // cheat
  if DealerTotal > 21 then                         // track dealer bust
    ShowMessage('Dealer Busted!');
end;
```

This method accepts a Boolean parameter called `CardVisible`, which indicates whether the card should be dealt face up. This is because blackjack rules dictate that the dealer's first card must remain face down until the player has chosen to hold or has busted. Observing this rule, the first call to `DealerHit()` will result in `False` being passed in `CardVisible`.

The `FreeCards()` procedure is responsible for removing all the `TCardX` controls on the main form. Because the application doesn't keep an array or a bunch of variables of type `TCardX` around to manage the cards on the screen, this procedure iterates through the form's `Controls` array property looking for elements of type `TCardX`. When a control of that type is found, its `Free` method is called to remove it from memory. The trick here is to be sure to go *backward* through the array. If you don't go backward, you run the risk of changing the order of controls in the array while you're traversing the array, which can cause errors. The code for the `FreeCards` procedure is shown here:

```
procedure TMainForm.FreeCards;
{ frees all AX Ctl cards on the screen }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // go backward!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;
```

That completes the explanation of the main portions of the code that manipulates the ActiveX controls. The complete listing for `Main.pas`, the main unit for this application, is shown in Listing 7.3.

LISTING 7.3 The `Main.pas` Unit for the `BJ` Project

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OleCtrls, Card_TLB, Cards, PlayCard, StdCtrls, ExtCtrls, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    MainMenu1: TMainMenu;
    Play1: TMenuItem;
    Deal1: TMenuItem;
    Hit1: TMenuItem;
```

```
    Hold1: TMenuItem;
    DoubleDown1: TMenuItem;
    N1: TMenuItem;
    Close1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    Panel2: TPanel;
    Label3: TLabel;
    CashLabel: TLabel;
    BetLabel: TLabel;
    HitBtn: TButton;
    DealBtn: TButton;
    HoldBtn: TButton;
    ExitBtn: TButton;
    BetEdit: TEdit;
    DblBtn: TButton;
    CheatPanel: TPanel;
    DealLbl: TLabel;
    PlayLbl: TLabel;
    Label4: TLabel;
    Label6: TLabel;
    Cheat1: TMenuItem;
    N2: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure Cheat1Click(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure DblBtnClick(Sender: TObject);
    procedure DealBtnClick(Sender: TObject);
    procedure HitBtnClick(Sender: TObject);
    procedure HoldBtnClick(Sender: TObject);
  private
    CardDeck: TCardDeck;
    CurCard: TPlayingCard;
    NextPlayerPos: integer;
    NextDealerPos: integer;
    PlayerTotal: integer;
    DealerTotal: integer;
    PAceFlag: Boolean;
    DAceFlag: Boolean;
    PBJFlag: Boolean;
    DBJFlag: Boolean;
    DDFlag: Boolean;
    Procedure Deal;
    procedure DealerHit(CardVisible: Boolean);
```

continues

LISTING 7.3 Continued

```
    procedure DoubleDown;
    procedure EnableMoves(Enable: Boolean);
    procedure FreeCards;
    procedure Hit;
    procedure Hold;
    procedure ShowFirstCard;
    procedure ShowWinner;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses AboutU;

const
  PYPos = 175;        // starting y pos for player cards
  DYPos = 10;         // ditto for dealer's cards

procedure TMainForm.FormCreate(Sender: TObject);
begin
  CardDeck := TCardDeck.Create;
end;

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

procedure TMainForm.About1Click(Sender: TObject);
{ Creates and invokes about box }
begin
  with TAboutBox.Create(Self) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TMainForm.Cheat1Click(Sender: TObject);
begin
```

```
  Cheat1.Checked := not Cheat1.Checked;
  CheatPanel.Visible := Cheat1.Checked;
end;

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

procedure TMainForm.DblBtnClick(Sender: TObject);
begin
  DoubleDown;
end;

procedure TMainForm.DealBtnClick(Sender: TObject);
begin
  Deal;
end;

procedure TMainForm.HitBtnClick(Sender: TObject);
begin
  Hit;
end;

procedure TMainForm.HoldBtnClick(Sender: TObject);
begin
  Hold;
end;

procedure TMainForm.Deal;
{ Deals a new hand for dealer and player }
begin
  FreeCards;                        // remove any cards from screen
  BetEdit.Enabled := False;         // disable bet edit ctrl
  BetLabel.Enabled := False;        // disable bet label
  if CardDeck.Count < 11 then       // reshuffle deck if < 11 cards
  begin
    Panel1.Caption := 'Reshuffling and dealing...';
    CardDeck.Shuffle;
  end
  else
    Panel1.Caption := 'Dealing...';
  Panel1.Show;                      // show "dealing" panel
  Panel1.Update;                    // make sure it's visible
  NextPlayerPos := 10;              // set horiz position of cards
  NextDealerPos := 10;
```

continues

LISTING 7.3 Continued

```
  PlayerTotal := 0;                 // reset card totals
  DealerTotal := 0;
  PAceFlag := False;                // reset flags
  DAceFlag := False;
  PBJFlag := False;
  DBJFlag := False;
  DDFlag := False;
  Hit;                              // hit player
  DealerHit(False);                 // hit dealer
  Hit;                              // hit player
  DealerHit(True);                  // hit dealer
  Panel1.Hide;                      // hide panel
  if (PlayerTotal = 11) and PAceFlag then
    PBJFlag := True;                // check player blackjack
  if (DealerTotal = 11) and DAceFlag then
    DBJFlag := True;                // check dealer blackjack
  if PBJFlag or DBJFlag then        // if a blackjack occurred
  begin
    ShowFirstCard;                  // flip dealer's card
    ShowMessage('Blackjack!');
    ShowWinner;                     // determine winner
  end
  else
    EnableMoves(True);              // enable hit, hold double down
end;

procedure TMainForm.DealerHit(CardVisible: Boolean);
{ Dealer takes a hit }
begin
  CurCard := CardDeck.Draw;           // dealer draws a card
  with TCardX.Create(Self) do         // create the ActiveX control
  begin
    Left := NextDealerPos;            // place card on form
    Top := DYPos;
    Suit := Ord(CurCard.Suit);        // assign suit
    FaceUp := CardVisible;
    Value := Ord(CurCard.Face);       // assign face
    Parent := Self;                   // assign parent for AX Ctl
    Inc(NextDealerPos, Width div 2);  // track where to place next card
    Update;                           // show card
  end;
  if CurCard.Face = cvAce then DAceFlag := True;   // set Ace flag
  Inc(DealerTotal, GetCardValue(CurCard));         // keep count
  DealLbl.Caption := IntToStr(DealerTotal);        // cheat
  if DealerTotal > 21 then                         // track dealer bust
```

```
    ShowMessage('Dealer Busted!');
end;

procedure TMainForm.DoubleDown;
{ Called to double down on dealt hand }
begin
  DDFlag := True;               // set double down flag to adjust bet
  Hit;                          // take one card
  Hold;                         // let dealer take his cards
end;

procedure TMainForm.EnableMoves(Enable: Boolean);
{ Enables/disables moves buttons/menu items }
begin
  HitBtn.Enabled := Enable;             // Hit button
  HoldBtn.Enabled := Enable;            // Hold button
  DblBtn.Enabled := Enable;             // Double down button
  Hit1.Enabled := Enable;               // Hit menu item
  Hold1.Enabled := Enable;              // Hold menu item
  DoubleDown1.Enabled := Enable;        // Double down menu item
end;

procedure TMainForm.FreeCards;
{ frees all AX Ctl cards on the screen }
var
  i: integer;
begin
  for i := ControlCount - 1 downto 0 do  // go backward!
    if Controls[i] is TCardX then
      Controls[i].Free;
end;

procedure TMainForm.Hit;
{ Player hit }
begin
  CurCard := CardDeck.Draw;          // draw card
  with TCardX.Create(Self) do        // create card AX Ctl
  begin
    Left := NextPlayerPos;           // set position
    Top := PYPos;
    Suit := Ord(CurCard.Suit);       // set suit
    Value := Ord(CurCard.Face);      // set value
    Parent := Self;                  // assign parent
    Inc(NextPlayerPos, Width div 2); // track position
    Update;                          // Display card
  end;
```

continues

LISTING 7.3 Continued

```
  DblBtn.Enabled := False;               // hit disables double down
  if CurCard.Face = cvAce then PAceFlag := True;  // set ace flag
  Inc(PlayerTotal, GetCardValue(CurCard));        // keep running total
  PlayLbl.Caption := IntToStr(PlayerTotal);       // cheat
  if PlayerTotal > 21 then                        // track bust
  begin
    ShowMessage('Busted!');
    ShowFirstCard;
    ShowWinner;
  end;
end;

procedure TMainForm.Hold;
{ Player holds. This procedure allows dealer to draw cards. }
begin
  EnableMoves(False);
  ShowFirstCard;                  // show dealer card
  if PlayerTotal <= 21 then       // if player hasn't busted...
  begin
    if DAceFlag then              // if dealer has an Ace...
    begin
      { Dealer must hit soft 17 }
      while (DealerTotal <= 7) or ((DealerTotal >= 11) and
        (DealerTotal < 17)) do
        DealerHit(True);
    end
    else
      // if no Ace, keep hitting until 17 is reached
      while DealerTotal < 17 do DealerHit(True);
  end;
  ShowWinner;                     // Determine winner
end;

procedure TMainForm.ShowFirstCard;
var
  i: integer;
begin
  // make sure all cards are face-up
  for i := 0 to ControlCount - 1 do
    if Controls[i] is TCardX then
    begin
      TCardX(Controls[i]).FaceUp := True;
      TCardX(Controls[i]).Update;
    end;
```

```
end;

procedure TMainForm.ShowWinner;
{ Determines winning hand }
var
  S: string;
begin
  if DAceFlag then                      // if dealer has an Ace...
  begin
    if DealerTotal + 10 <= 21 then      // figure best hand
      inc(DealerTotal, 10);
  end;
  if PACeFlag then                      // if player has an Ace...
  begin
    if PlayerTotal + 10 <= 21 then      // figure best hand
      inc(PlayerTotal, 10);
  end;
  if DealerTotal > 21 then              // set score to 0 if busted
    DealerTotal := 0;
  if PlayerTotal > 21 then
    PlayerTotal := 0;
  if PlayerTotal > DealerTotal then     // if player wins...
  begin
    S := 'You win!';
    if DDFlag then                      // pay 2:1 on double down
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) * 2)
    else                                // pay 1:1 normally
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text));
    if PBJFlag then                     // pay 1.5:1 on blackjack
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) +
        StrToInt(BetEdit.Text) div 2)
  end
  else if DealerTotal > PlayerTotal then    // if dealer wins...
  begin
    S := 'Dealer wins!';
    if DDFlag then                      // lose 2x on double down
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text) * 2)
    else                                // normal loss
      CashLabel.Caption := IntToStr(StrToInt(CashLabel.Caption) -
        StrToInt(BetEdit.Text));
  end
  else
    S := 'Push!';                       // push, no one wins
```

continues

LISTING 7.3 Continued

```
  if MessageDlg(S + #13#10'Do you want to play again with the same bet?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    Deal;
  BetEdit.Enabled := True;             // allow bet to change
  BetLabel.Enabled := True;
end;

end.
```

Invoking an ActiveX Control Method

In Listing 7.3, you might have noticed that the main form contains a method that creates and displays an About dialog box. Figure 7.7 shows what this About dialog box looks like when invoked.

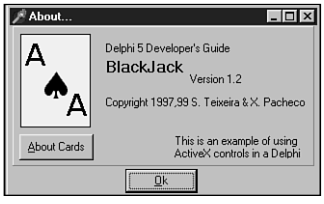

FIGURE 7.7
DDG BlackJack's About dialog box.

This About dialog box is special because it contains a button that, when selected, shows an About box for the `CardX` ActiveX control by calling its `AboutBox()` method. The About box for the `CardX` control is shown in Figure 7.8.

The code that accomplishes this task follows. Looking ahead, the same technique used to make `vTable` calls to OLE Automation servers in Chapter 23, "COM and ActiveX," is used here.

```
procedure TAboutBox.CardBtnClick(Sender: TObject);
begin
  Card.AboutBox;
end;
```

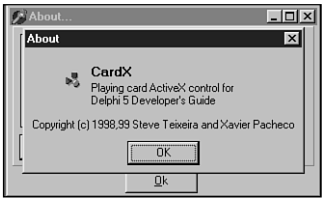

FIGURE 7.8
The `Cards` *ActiveX control About dialog box.*

Summary

After reading this chapter, you should understand all the important aspects of using ActiveX controls in the Delphi environment. You learned about integrating an ActiveX control into Delphi, how the Object Pascal ActiveX control wrapper works, how to deploy an ActiveX control–equipped application, how to register an ActiveX control, and how to incorporate ActiveX controls into an application. Because of their market presence, ActiveX controls often can offer a blast of instant productivity. However, because ActiveX controls have some disadvantages, also remember to look for native VCL components when shopping for controls.

Graphics Programming with GDI and Fonts

CHAPTER

8

IN THIS CHAPTER

In previous chapters, you worked with a property called `Canvas`. `Canvas` is appropriately named because you can think of a window as an artist's blank canvas on which various Windows objects are painted. Each button, window, cursor, and so on is nothing more than a collection of pixels in which the colors have been set to give it some useful appearance. In fact, think of each individual window as a separate surface on which its separate components are painted. To take this analogy a bit further, imagine that you're an artist who requires various tools to accomplish your task. You need a palette from which to choose different colors. You'll probably use different styles of brushes, drawing tools, and special artist's techniques as well. Win32 makes use of similar tools and techniques—in the programming sense—to paint the various objects with which users interact. These tools are made available through the Graphics Device Interface, otherwise known as the *GDI*.

Win32 uses the GDI to paint or draw the images you see on your computer screen. Before Delphi, in traditional Windows programming, programmers worked directly with the GDI functions and tools. Now, the `TCanvas` object encapsulates and simplifies the use of these functions, tools, and techniques. This chapter teaches you how to use `TCanvas` to perform useful graphics functions. You'll also see how you can create advanced programming projects with Delphi 5 and Win32 GDI. We illustrate this by creating a paint program and animation program.

Delphi's Representation of Pictures: TImage

The `TImage` component represents a graphical image that can be displayed anywhere on a form and is available from Delphi 5's Component Palette. With `TImage`, you can load and display a bitmap file (`.bmp`), a 16-bit Windows metafile (`.wmf`), a 32-bit enhanced metafile (`.emf`), an icon file (`.ico`), a JPEG file (`.jpg`, `.jpeg`) or other file formats handled by add-in `TGraphic` classes. The image data actually is stored by `TImage`'s `Picture` property, which is of the type `TPicture`.

Graphic Images: Bitmaps, Metafiles, and Icons

Bitmaps

Win32 *bitmaps* are binary information arranged in a pattern of bits that represent a graphical image. More specifically, these bits store color information items called *pixels*. There are two types of bitmaps: device-dependent bitmaps (DDB) and device-independent bitmaps (DIB).

As a Win32 programmer, you probably won't be dealing much with DDBs because this format was kept solely for backward compatibility. Device-dependent bitmaps, as the name implies, are dependent on the device in which they're created. Bitmaps in this format, when saved, do not store information regarding the color palette they use nor do they store information regarding their resolution.

In contrast, device-independent bitmaps (DIBs) do store information to allow them to be displayed on any device without radically changing their appearance.

In memory, both DDBs and DIBs are represented with the same structures, for the most part. One key difference is that DDBs use the palette provided by the system, whereas DIBs provide their own palette. To take this explanation further, DDBs are simply native storage, handled by video driver routines and video hardware. DIBs are standardized pixel formats, handled by GDI generic routines and stored in global memory. Some video cards use DIB pixel formats as native storage, so you get DDB=DIB. In general, DIB gives you more flexibility and simplicity, sometimes at a slight performance cost. DDBs are always faster but not as convenient.

Metafiles

Unlike bitmaps, *metafiles* are vector-based graphical images. Metafiles are files in which a series of GDI routines are stored, enabling you to save GDI function calls to disk so that you can redisplay the image later. This also enables you to share your drawing routines with other programs without having to call the specific GDI functions in each program. Other advantages to metafiles are that they can be scaled to arbitrary dimensions and still retain their smooth lines and arcs—bitmaps don't do this as well. In fact, this is one of the reasons the Win32 printing engine is built around the enhanced metafile storage for print jobs.

There are two metafile formats: standard metafiles, typically stored in a file with a `.wmf` extension, and enhanced metafiles, typically stored in a file with an `.emf` extension.

Standard metafiles are a holdover from the Win16 system. Enhanced metafiles are more robust and accurate. Use EMFs if you're producing metafiles for your own applications. If you're exporting your metafiles to older programs that might not be able to use the enhanced format, use the 16-bit WMFs. Know, however, that by stepping down to the 16-bit WMFs, you'll also lose several GDI primitives that EMFs support but WMFs do not. Delphi 5's `TMetafile` class knows about both types of metafiles.

Icons

Icons are Win32 resources that usually are stored in an icon file with an `.ico` extension. They may also reside in a resource file (`.res`). There are two typical sizes of icons in Windows: large icons that are 32×32 pixels, and small icons that are 16×16 pixels. All Windows applications use both icon sizes. Small icons are displayed in the application's upper-left corner of the main window and also in the Windows List view control. Delphi's encapsulation of this control is the `TListView` component. This control appears on the Win32 page of the Component Palette.

Icons are made up of two bitmaps. One bitmap, referred to as the *image*, is the actual icon image as it is displayed. The other bitmap, referred to as the *mask*, makes it possible to achieve transparency when the icon is displayed. Icons are used for a variety of purposes. For example, icons appear on an application's taskbar and in message boxes where the question mark, exclamation point, or stop sign icons are used as attention grabbers.

TPicture is a container class for the TGraphic abstract class. A *container class* means that TPicture can hold a reference to and display a TBitmap, TMetafile, TIcon, or any other TGraphic type, without really caring which is which. You use TImage.Picture's properties and methods to load image files into a TImage component. For example, use the following statement:

```
MyImage.Picture.LoadFromFile('FileName.bmp');
```

Use a similar statement to load icon files or metafiles. For example, the following code loads a Win32 metafile:

```
MyImage.Picture.LoadFromFile('FileName.emf');
```

This code loads a Win32 icon file:

```
MyImage.Picture.LoadFromFile('FileName.ico');
```

In Delphi 5, TPicture can now load JPEG images using the same technique for loading bitmaps:

```
MyImage.Picture.LoadFromFile('FileName.jpeg');
```

Saving Images

To save an image use the SaveToFile() method:

```
MyImage.Picture.SaveToFile('FileName.bmp');
```

The TBitmap class encapsulates the Win32 bitmap and palette, and it provides the methods to load, store, display, save, and copy the bitmapped images. TBitmap also manages palette realization automatically. This means that the tedious task of managing bitmaps has been simplified substantially with Delphi 5's TBitmap class, which enables you to focus on using the bitmap and frees you from having to worry about all the underlying implementation details.

NOTE

TBitmap isn't the only object that manages palette realization. Components such as TImage, TMetafile, and every other TGraphic descendant also realize their bitmaps' palettes on request. If you build components that contain a TBitmap object that might have 256-color images, you'll need to override your component's GetPalette() method to return the color palette of the bitmap.

To create an instance of a TBitmap class and load a bitmap file, for example, you use the following commands:

```
MyBitmap := TBitmap.Create;
MyBitmap.LoadFromFile('MyBMP.BMP');
```

CAUTION

Another method of loading bitmaps into an application is to load them from a resource file. We'll discuss this method shortly.

To copy one bitmap to another, you use the `TBitmap.Assign()` method, as in this example:

```
Bitmap1.Assign(Bitmap2);
```

You also can copy a portion of a bitmap from one `TBitmap` instance to another `TBitmap` instance or even to the form's canvas by using the `CopyRect()` method:

```
var
  R1: TRect;
begin
  with R1 do
  begin
    Top := 0;
    Left := 0;
    Right := BitMap2.Width div 2;
    Bottom := BitMap2.Height div 2;
  end;
  Bitmap1.Canvas.CopyRect(ClientRect, BitMap2.Canvas, R1);
end;
```

In the preceding code, you first calculate the appropriate values in a `TRect` record and then use the `TCanvas.CopyRect()` method to copy a portion of the bitmap. A `TRect` is defined as follows:

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

This technique will be used in the paint program later in the chapter. `CopyRect()` automatically stretches the copied portion of the source canvas to fill the destination rectangle.

CAUTION

You should be aware of a significant difference in resource consumption for copying bitmaps in the previous two examples. The `CopyRect()` technique doubles the memory

continues

use in that two separate copies of the image exist in memory. The `Assign()` technique costs nothing because the two bitmap objects will share a reference to the same image in memory. If you happen to modify one of the bitmap objects, VCL will clone the image using a copy-on-write scheme.

Another method you can use to copy the entire bitmap to the form's canvas so that it shrinks or expands to fit inside the canvas's boundaries is the `StretchDraw()` method. Here's an example:

```
Canvas.StretchDraw(R1, MyBitmap);
```

We'll discuss `TCanvas`'s methods later in this chapter.

Using the TCanvas Properties

Higher-level classes such as `TForm` and `TGraphicControl` descendants have a `Canvas` property. The canvas serves as the painting surface for your form's other components. The tools that `Canvas` uses to do the drawing are pens, brushes, and fonts.

Using Pens

In this section, we first explain how to use the `TPen` properties and then show you some code in a sample project that uses these properties.

Pens enable you to draw lines on the canvas and are accessed from the `Canvas.Pen` property. You can change how lines are drawn by modifying the pen's properties: `Color`, `Width`, `Style`, and `Mode`.

The `Color` property specifies a pen's color. Delphi 5 provides predefined color constants that match many common colors. For example, the constants `clRed` and `clYellow` correspond to the colors red and yellow. Delphi 5 also defines constants to represent the Win32 system screen element colors such as `clActiveCaption` and `clHighlightText`, which correspond to the Win32 active captions and highlighted text. The following line assigns the color blue to the canvas's pen:

```
Canvas.Pen.color := clblue;
```

This line shows you how to assign a random color to `Canvas`'s `Pen` property:

```
Pen.Color := TColor(RGB(Random(255),Random(255), Random(255)));
```

RGB() and TColor

Win32 represents colors as long integers in which the lowest three bytes each signify a red, green, and blue intensity level. The combination of the three values makes up a valid Win32 color. The `RGB(R, G, B)` function takes three parameters for the red,

green, and blue intensity levels. It returns a Win32 color as a long integer value. This is represented as a `TColor` Delphi type. There are 255 possible values for each intensity level and approximately 16 million colors that can be returned from the `RGB()` function. `RGB(0, 0, 0)`, for example, returns the color value for black, whereas `RGB(255, 255, 255)` returns the color value for white. `RGB(255, 0 ,0)`, `RGB(0, 255, 0)`, and `RGB(0, 0, 255)` return the color values for red, green, and blue, respectively. By varying the values passed to `RGB()`, you can obtain a color anywhere within the color spectrum.

`TColor` is specific to VCL and refers to constants defined in the `Graphics.pas` unit. These constants map to either the closest matching color in the system palette or to a defined color in the Windows Control Panel. For example, `clBlue` maps to the color blue whereas the `clBtnFace` maps to the color specified for button faces. In addition to the three bytes to represent the color, `TColor`'s highest order byte specifies how a color is matched. Therefore, if the highest order byte is `$00`, the represented color is the closest matching color in the system palette. A value of `$01` represents the closest matching color in the currently realized palette. Finally, a color of `$02` matches with the nearest color in the logical palette of the current device context. You will find additional information in the Delphi help file under "`TColor type`."

TIP

Use the `ColorToRGB()` function to convert Win32 system colors, such as `clWindow`, to a valid RGB color. The function is described in Delphi 5's online help.

The pen can also draw lines with different drawing styles, as specified by its `Style` property. Table 8.1 shows the different styles you can set for `Pen.Style`.

TABLE 8.1 Pen Styles

Style	*Draws*
`psClear`	An invisible line
`psDash`	A line made up of a series of dashes
`psDashDot`	A line made up of alternating dashes and dots
`psDashDotDot`	A line made up of a series of dash-dot-dot combinations
`psDot`	A line made up of a series of dots
`psInsideFrame`	A line within a frame of closed shapes that specify a bounding rectangle
`psSolid`	A solid line

The following line shows how you would change the pen's drawing style:

```
Canvas.Pen.Style := psDashDot;
```

Figure 8.1 shows how the different pen styles appear when drawn on the form's canvas. One thing to note: The "in between" colors in the stippled lines come from the brush color. If you want to make a black dashed line run across a red square, you would need to set the `Canvas.Brush.Color` to `clRed` or set the `Canvas.Brush.Style` to `bsClear`. Setting both the pen and brush color is how you would draw, for example, a red and blue dashed line across a white square.

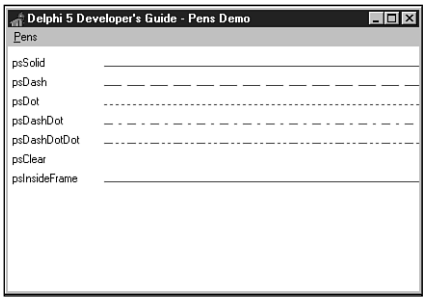

FIGURE 8.1

Different pen styles.

The `Pen.Width` property enables you to specify the width, in pixels, that the pen uses for drawing. When this property is set to a larger width, the pen draws with thicker lines.

NOTE

The stipple line style applies only to pens with a width of 1. Setting the pen width to 2 will draw a solid line. This is a holdover from the 16-bit GDI that Win32 emulates for compatibility. Windows 95/98 does not do fat stippled lines, but Windows NT/2000 can if you use only the extended GDI feature set.

Three factors determine how Win32 draws pixels or lines to a canvas surface: the pen's color, the surface or destination color, and the bitwise operation that Win32 performs on the two-color values. This operation is known as a *raster operation* (ROP). The `Pen.Mode` property specifies the ROP to be used for a given canvas. Sixteen modes are predefined in Win32, as shown in Table 8.2.

TABLE 8.2 Win32 Pen Modes on Source `Pen.Color` (`S`) and `Destination` (`D`) Color

Mode	*Result Pixel Color*	*Boolean Operation*
`pmBlack`	Always black	`0`
`pmWhite`	Always white	`1`
`pmNOP`	Unchanged	`D`
`pmNOT`	Inverse of `D` color	`not D`
`pmCopy`	Color specified by `S`	`S`
`pmNotCopy`	Inverse of `S`	`not S`
`pmMergePenNot`	Combination `S` and inverse of `D`	`S or not D`
`pmMaskPenNot`	Combination of colors common to `S` and inverse of `D`	`S and not D`
`pmMergeNotPen`	Combination of `D` and inverse of `S`	`not S or D`
`pmMaskNotPen`	Combination of colors common to `D` and inverse of `S`	`not S and D`
`pmMerge`	Combination of `S` and `D`	`S or D`
`pmNotMerge`	Inverse of `pmMerge` operation on `S` and `D`	`not (S or D)`
`pmMask`	Combination of colors common to `S` and `D`	`S and D`
`pmNotMask`	Inverse of `pmMask` operation on `S` and `D`	`not (S and D)`
`pmXor`	Combination of colors in either `S` or `D` but not both	`S XOR D`
`pmNotXor`	Inverse of `pmXOR` operation on `S` and `D`	`not (S XOR D)`

`Pen.mode` is `pmCopy` by default. This means that the pen draws with the color specified by its `Color` property. Suppose that you want to draw black lines on a white background. If a line crosses over a previously drawn line, it should draw white rather than black.

One way to do this would be to check the color of the area you're going to draw to—if it's white, set `pen.Color` to black; if it is black, set `pen.Color` to white. Although this approach works, it would be cumbersome and slow. A better approach would be to set `Pen.Color` to `clBlack` and `Pen.Mode` to `pmNot`. This would result in the pen drawing the inverse of the merging operation with the pen and surface color. Figure 8.2 shows you the result of this operation when drawing with a black pen in a crisscross fashion.

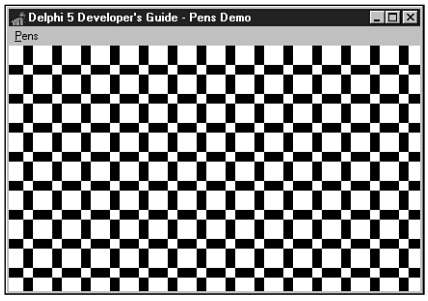

FIGURE 8.2

The output of a `pmNotMerge` *operation.*

Listing 8.1 is an example of the project on the CD that illustrates the code that resulted in Figures 8.1 and 8.2. You'll find this demo on the CD.

LISTING 8.1 Illustration of Pen Operations

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiPens: TMenuItem;
    mmiStyles: TMenuItem;
    mmiPenColors: TMenuItem;
    mmiPenMode: TMenuItem;
    procedure mmiStylesClick(Sender: TObject);
    procedure mmiPenColorsClick(Sender: TObject);
    procedure mmiPenModeClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure ClearCanvas;
    procedure SetPenDefaults;
  end;

var
```

```
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
 // Clear the contents of the canvas
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    Canvas.FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetPenDefaults;
begin
  with Canvas.Pen do
  begin
    Width := 1;
    Mode := pmCopy;
    Style := psSolid;
    Color := clBlack;
  end;
end;

procedure TMainForm.mmiStylesClick(Sender: TObject);
var
  yPos: integer;
  PenStyle: TPenStyle;
begin
  ClearCanvas;     // First clear Canvas's contents
  SetPenDefaults;
  // yPos represent the Y coordinate
  YPos := 20;
  with Canvas do
  begin
    for PenStyle := psSolid to psInsideFrame do
    begin
      Pen.Color := clBlue;
      Pen.Style := PenStyle;
      MoveTo(100, yPos);
```

continues

LISTING 8.1 Continued

```
      LineTo(ClientWidth, yPos);
      inc(yPos, 20);
    end;

    // Write out titles for the various pen styles
    TextOut(1, 10, ' psSolid ');
    TextOut(1, 30, ' psDash ');
    TextOut(1, 50, ' psDot ');
    TextOut(1, 70, ' psDashDot ');
    TextOut(1, 90, ' psDashDotDot ');
    TextOut(1, 110, ' psClear ');
    TextOut(1, 130, ' psInsideFrame ');
  end;
end;

procedure TMainForm.mmiPenColorsClick(Sender: TObject);
var
  i: integer;
begin
  ClearCanvas;    // Clear Canvas's contents
  SetPenDefaults;
  with Canvas do
  begin
    for i := 1 to 100 do
    begin
      // Get a random pen color draw a line using that color
      Pen.Color := RGB(Random(256),Random(256), Random(256));
      MoveTo(random(ClientWidth), Random(ClientHeight));
      LineTo(random(ClientWidth), Random(ClientHeight));
    end
  end;
end;

procedure TMainForm.mmiPenModeClick(Sender: TObject);
var
  x,y: integer;
begin
  ClearCanvas;  // Clear the Canvas's contents
  SetPenDefaults;
  y := 10;
  canvas.Pen.Width := 20;
```

```
  while y < ClientHeight do
  begin
    canvas.MoveTo(0, y);
    // Draw a line and increment Y value
    canvas.LineTo(ClientWidth, y);
    inc(y, 30);
  end;
  x := 5;

  canvas.pen.Mode := pmNot;
  while x < ClientWidth do
  begin
    Canvas.MoveTo(x, 0);
    canvas.LineTo(x, ClientHeight);
    inc(x, 30);
  end;
end;

end.
```

Listing 8.1 shows three examples of dealing with the canvas's pen. The two helper functions, `ClearCanvas()` and `SetPenDefaults()`, are used to clear the contents of the main form's canvas and to reset the `Canvas.Pen` properties to their default values as these properties are modified by each of the three event handlers.

`ClearCanvas()` is a useful technique for erasing the contents of any component containing a `Canvas` property. `ClearCanvas()` uses a solid white brush to erase whatever was previously painted on the `Canvas`. `FillRect()` is responsible for painting a rectangular area as specified by a `TRect` structure, `ClientRect`, which is passed to it.

The `mmiStylesClick()` method shows how to display the various `TPen` styles as shown in Figure 8.1 by drawing horizontal lines across the form's `Canvas` using a different `TPen` style. Both `TCanvas.MoveTo()` and `TCanvas.LineTo()` enable you to draw lines on the canvas.

The `mmiPenColorsClick()` method illustrates drawing lines using a different `TPen` color. Here you use the `RGB()` function to retrieve a color to assign to `TPen.Color`. The three values you pass to `RGB()` are each random values within range of `0` to `255`. The output of this method is shown in Figure 8.3.

Finally, the `mmiPenModeClick()` method illustrates how to draw lines using a different pen mode. Here you use the `pmNot` mode to perform the actions previously discussed, resulting in the output shown in Figure 8.2.

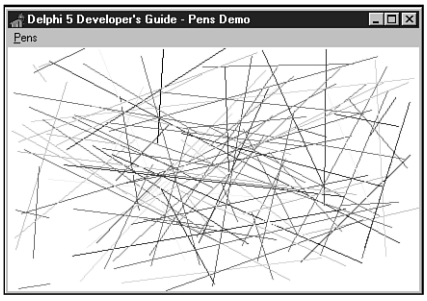

FIGURE 8.3

Output from the `mmiPenColorsClick()` *method.*

Using TCanvas's Pixels

The `TCanvas.Pixels` property is a two-dimensional array in which each element represents a pixel's `TColor` value on the form's surface or client area. The upper-left corner of your form's painting surface is given by

```
Canvas.Pixels[0,0]
```

and the lower-right corner is

```
Canvas.Pixels[clientwidth, clientheight];
```

It's rare that you'll ever have to access individual pixels on your form. In general, you do not want to use the `Pixels` property because it's slow. Accessing this property uses the `GetPixel()`/`SetPixel()` GDI functions, which Microsoft has acknowledged are flawed and will never be efficient. This is because both functions rely on 24-bit RGB values. When not working with 24-bit RGB device contexts, these functions must perform serious color-matching gymnastics to convert the RGB into a device pixel format. For quick pixel manipulation, use the `TBitmap.ScanLine` array property instead. To fetch or set one or two pixels at a time, using `Pixels` is okay.

Using Brushes

This section discusses the `TBrush` properties and shows you some code in a sample project that uses these properties.

Using the TBrush Properties

A canvas's brush fills in areas and shapes drawn on the canvas. This differs from a `TPen` object, which enables you to draw lines to the canvas. A brush enables you to fill an area on the canvas using various colors, styles, and patterns.

`Canvas`'s `TBrush` object has three important properties that specify how the brush paints on the canvas's surface: `Color`, `Style`, and `Bitmap`. `Color` specifies the brush's color, `Style` specifies

the pattern of the brush background, and `Bitmap` specifies a bitmap you can use to create custom patterns for the brush's background.

Eight brush options are specified by the `Style` property: `bsSolid`, `bsClear`, `bsHorizontal`, `bsVertical`, `bsFDiagonal`, `bsBDiagonal`, `bsCross`, and `bsDiagCross`. By default, the brush color is `clWhite` with a `bsSolid` style and no bitmap. You can change the color and style to fill an area with different patterns. The example in the following section illustrates using each of the `TBrush` properties.

TBrush Code Example

Listing 8.2 shows you the unit for a project that illustrates the use of the `TBrush` properties just discussed. You can load this project from the CD.

LISTING 8.2 `TBrush` Example

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms,
Dialogs, Menus, ExtCtrls;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiBrushes: TMenuItem;
    mmiPatterns: TMenuItem;
    mmiBitmapPattern1: TMenuItem;
    mmiBitmapPattern2: TMenuItem;
    procedure mmiPatternsClick(Sender: TObject);
    procedure mmiBitmapPattern1Click(Sender: TObject);
    procedure mmiBitmapPattern2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation
```

continues

LISTING 8.2 Continued

```
{$R *.DFM}

procedure TMainForm.ClearCanvas;
var
  R: TRect;
begin
 // Clear the contents of the canvas
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetWindowRect(Handle, R);
    R.TopLeft := ScreenToClient(R.TopLeft);
    R.BottomRight := ScreenToClient(R.BottomRight);
    FillRect(R);
  end;
end;

procedure TMainForm.mmiPatternsClick(Sender: TObject);
begin
  ClearCanvas;
  with Canvas do
  begin
    // Write out titles for the various brush styles
    TextOut(120, 101, 'bsSolid');
    TextOut(10, 101, 'bsClear');
    TextOut(240, 101, 'bsCross');
    TextOut(10, 221, 'bsBDiagonal');
    TextOut(120, 221, 'bsFDiagonal');
    TextOut(240, 221, 'bsDiagCross');
    TextOut(10, 341, 'bsHorizontal');
    TextOut(120, 341, 'bsVertical');

    // Draw a rectangle with the various brush styles

    Brush.Style := bsClear;
    Rectangle(10, 10, 100, 100);
    Brush.Color := clBlack;

    Brush.Style := bsSolid;
    Rectangle(120, 10, 220, 100);

    { Demonstrate that the brush is transparent by drawing
      colored rectangle, over which the brush style rectangle will
      be drawn. }
```

```
    Brush.Style := bsSolid;
    Brush.Color := clRed;
    Rectangle(230, 0, 330, 90);

    Brush.Style := bsCross;
    Brush.Color := clBlack;
    Rectangle(240, 10, 340, 100);

    Brush.Style := bsBDiagonal;
    Rectangle(10, 120, 100, 220);

    Brush.Style := bsFDiagonal;
    Rectangle(120, 120, 220, 220);

    Brush.Style := bsDiagCross;
    Rectangle(240, 120, 340, 220);

    Brush.Style := bsHorizontal;
    Rectangle(10, 240, 100, 340);

    Brush.Style := bsVertical;
    Rectangle(120, 240, 220, 340);

  end;
end;

procedure TMainForm.mmiBitmapPattern1Click(Sender: TObject);
begin
  ClearCanvas;
  // Load a bitmap from the disk
  FBitMap.LoadFromFile('pattern.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Draw a rectangle to cover the form's entire
      client area using the bitmap pattern as the
      brush with which to paint. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.mmiBitmapPattern2Click(Sender: TObject);
begin
  ClearCanvas;
  // Load a bitmap from the disk
```

continues

LISTING 8.2 Continued

```
  FBitMap.LoadFromFile('pattern2.bmp');
  Canvas.Brush.Bitmap := FBitmap;
  try
    { Draw a rectangle to cover the form's entire
      client area using the bitmap pattern as the
      brush with which to paint. }
    Canvas.Rectangle(0, 0, ClientWidth, ClientHeight);
  finally
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
end;

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

end.
```

TIP

The `ClearCanvas()` method that you use here is a handy routine for a utility unit. You can define `ClearCanvas()` to take `TCanvas` and `TRect` parameters to which the erase code will be applied:

```
procedure ClearCanvas(ACanvas: TCanvas; ARect: TRect);
begin
 // Clear the contents of the canvas
  with ACanvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ARect);
  end;
end;
```

The `mmiPatternsClick()` method illustrates drawing with various `TBrush` patterns. First, you draw out the titles and then, using each of the available brush patterns, draw rectangles on the form's canvas. Figure 8.4 shows the output of this method.

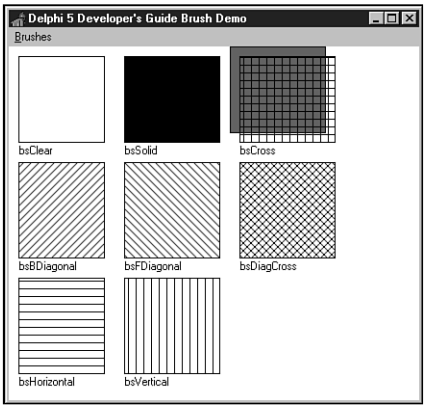

FIGURE 8.4

Brush patterns.

The `mmiBitmapPattern1Click()` and `mmiBitmapPattern2Click()` methods illustrate how to use a bitmap pattern as a brush. The `TCanvas.Brush` property contains a `TBitmap` property to which you can assign a bitmap pattern. This pattern will be used to fill the area painted by the brush instead of the pattern specified by the `TBrush.Style` property. There are a few rules for using this technique, however. First, you must assign a valid bitmap object to the property. Second, you must assign `nil` to the `Brush.Bitmap` property when you're finished with it because the brush does not take ownership of the bitmap object when you assign a bitmap to it. Figures 8.5 and 8.6 show the output of `mmiBitmapPattern1Click()` and `mmiBitmapPattern2Click()`, respectively.

NOTE

Windows limits the size of pattern brush bitmaps to 8×8 pixels, and they must be device-dependent bitmaps, not device-independent bitmaps. Windows will reject brush pattern bitmaps larger than 8×8; NT will accept larger but will only use the top-left 8×8 portion.

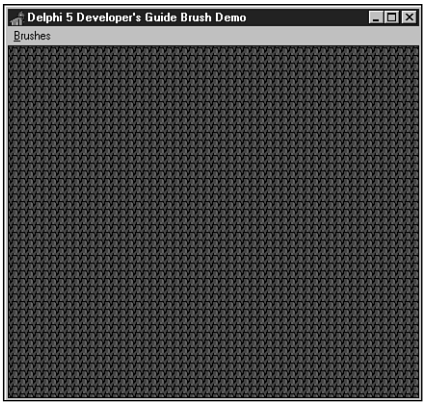

FIGURE 8.5
Output from `mmiBitmapPattern1Click()`.

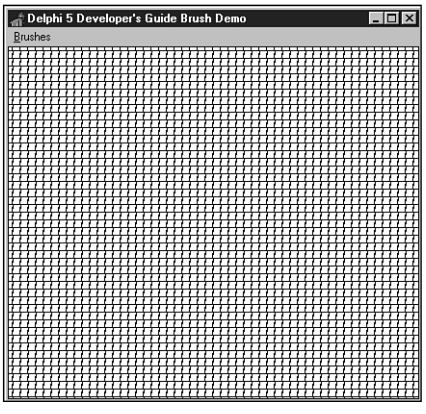

FIGURE 8.6
Output from `mmiBitmapPattern2Click()`.

TIP

Using a bitmap pattern to fill an area on the canvas doesn't only apply to the form's canvas but also to any component that contains a `Canvas` property. Just access the methods and/or properties of the `Canvas` property of the component rather than the form. For example, here's how to perform pattern drawing on a `TImage` component:

```
Image1.Canvas.Brush.Bitmap := SomeBitmap;
try
```

```
    Image1.Canvas.Rectangle(0, 0, Image1.Width, Image1.Height);
  finally
     Image1.Canvas.Brush.Bitmap := nil;
  end;
```

Using Fonts

The `Canvas.Font` property enables you to draw text using any of the available Win32 fonts. You can change the appearance of text written to the canvas by modifying the font's `Color`, `Name`, `Size`, `Height`, or `Style` property.

You can assign any of Delphi 5's predefined colors to `Font.Color`. The following code, for example, assigns the color red to the canvas's font:

```
Canvas.Font.Color := clRed;
```

The `Name` property specifies the Window's font name. For example, the following two lines of code assign different typefaces to `Canvas`'s font:

```
Canvas.Font.Name := 'New Times Roman';
```

`Canvas.Font.Size` specifies the font's size in points.

`Canvas.Font.Style` is a set composed of one style or a combination of the styles shown in Table 8.3.

TABLE 8.3 Font Styles

Value	*Style*
`fsBold`	Boldface
`fsItalic`	Italic
`fsUnderline`	Underlined
`fsStrikeOut`	A horizontal line through the font, giving it a strikethrough appearance

To combine two styles, use the syntax for combining multiple set values:

```
Canvas.Font.Style := [fsBold, fsItalic];
```

You can use `TFontDialog` to obtain a Win32 font and assign that font to the `TMemo.Font` property:

```
if FontDialog1.Execute then
    Memo1.Font.Assign(FontDialog1.Font);
```

The same can be done to assign the font selected in `TFontDialog` to the `Canvas`'s font:

```
Canvas.Font.Assign(FontDialog1.Font);
```

CAUTION

Make sure to use the `Assign()` method when copying `TBitMap`, `TBrush`, `TIcon`, `TMetaFile`, `TPen`, and `TPicture` instance variables. A statement such as

```
MyBrush1 := MyBrush2
```

might seem valid, but it performs a direct pointer copy so that both instances point to the same brush object, which can result in a heap leak. By using `Assign()`, you ensure that previous resources are freed.

This is not so when assigning between two `TFont` properties. Therefore, a statement such as

```
Form1.Font := Form2.Font
```

is a valid statement because the `TForm.Font` is a property whose write method internally calls `Assign()` to copy the data from the given font object. Be careful, however; this is only valid with when assigning `TFont` properties, not `TFont` variables. As a general rule, always use `Assign()`.

Additionally, you can assign individual attributes from the selected font in `TFontDialog` to the `Canvas`'s font:

```
Canvas.Font.Name := Font.Dialog1.Font.Name;
Canvas.Font.Size := Font.Dialog1.Font.Size;
```

We've quickly brushed over fonts here. A more thorough discussion on fonts appears at the end of this chapter.

Using the CopyMode Property

The `TCanvas.CopyMode` property determines how a canvas copies an image from another canvas onto itself. For example, when `CopyMode` holds the value `cmSrcCopy`, this means that the source image will be copied over the destination entirely. `CmSrcInvert`, however, causes the pixels of both the source and destination images to be combined using the bitwise `XOR` operator. `CopyMode` is used for achieving different effects when copying from one bitmap to another bitmap. A typical place where you would change the default value of `CopyMode` from `cmSrcCopy` to another value is when writing animation applications. You learn how to write animation later in this chapter.

To see how to use the `CopyMode` property, take a look at Figure 8.7.

Figure 8.7 shows a form that contains two images, both of which have an ellipse drawn on them. You select a `CopyMode` setting from the `TComboBox` component, and you would get various results when copying the one image over the other by clicking the Copy button. Figures 8.8 and 8.9 show what the effects would be by copying `imgFromImage` to `imgToImage` using the `cmSrcAnd` and `cmSrcInvert` copy modes.

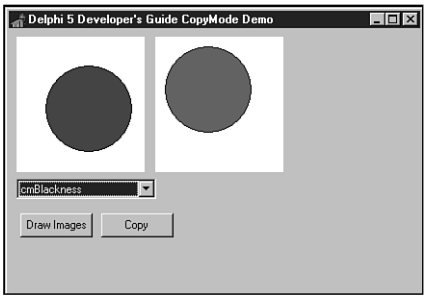

FIGURE 8.7

A form that contains two images to illustrate `CopyMode`.

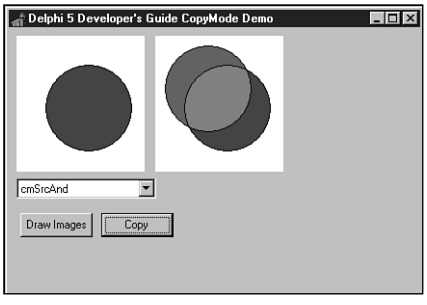

FIGURE 8.8

A copy operation using the `cmSrcAnd` *copy mode setting.*

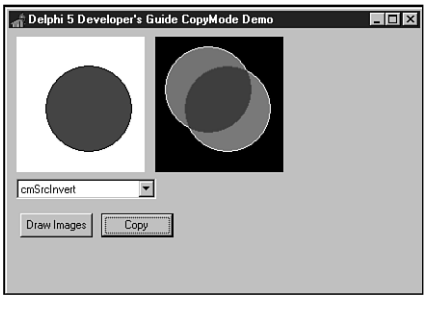

FIGURE 8.9

A copy operation using the `cmSrcInvert` *copy mode setting.*

Listing 8.3 shows the source code for the project, illustrating the various copy modes. You'll find this code on the CD.

LISTING 8.3 Project Illustrating `CopyMode` Usage

```
unit MainFrm;

interface

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

type

  TMainForm = class(TForm)
    imgCopyTo: TImage;
    imgCopyFrom: TImage;
    cbCopyMode: TComboBox;
    btnDrawImages: TButton;
    btnCopy: TButton;
    procedure FormShow(Sender: TObject);
    procedure btnCopyClick(Sender: TObject);
    procedure btnDrawImagesClick(Sender: TObject);
  private
    procedure DrawImages;
    procedure GetCanvasRect(AImage: TImage; var ARect: TRect);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.GetCanvasRect(AImage: TImage; var ARect: TRect);
var
  R: TRect;
  R2: TRect;
begin
  R := AImage.Canvas.ClipRect;
  with AImage do begin
    ARect.TopLeft     := Point(0, 0);
    ARect.BottomRight := Point(Width, Height);
  end;
  R2 := ARect;
  ARect := R2;
end;
```

```
procedure TMainForm.DrawImages;
var
  R: TRect;
begin
  // Draw an ellipse in img1
  with imgCopyTo.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyTo, R);
    FillRect(R);
    Brush.Color := clRed;
    Ellipse(10, 10, 100, 100);
  end;

  // Draw an ellipse in img2
  with imgCopyFrom.Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    GetCanvasRect(imgCopyFrom, R);
    FillRect(R);
    Brush.Color := clBlue;
    Ellipse(30, 30, 120, 120);
  end;

end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  // Initialize the combobox to the first item
  cbCopyMode.ItemIndex := 0;
  DrawImages;
end;

procedure TMainForm.btnCopyClick(Sender: TObject);
var
  cm: Longint;
  CopyToRect,
  CopyFromRect: TRect;
begin
  // Determine the copy mode based on the combo box selection
  case cbCopyMode.ItemIndex of
   0:  cm := cmBlackNess;
   1:  cm := cmDstInvert;
   2:  cm := cmMergeCopy;
```

continues

LISTING 8.3 Continued

```
   3:  cm := cmMergePaint;
   4:  cm := cmNotSrcCopy;
   5:  cm := cmNotSrcErase;
   6:  cm := cmPatCopy;
   7:  cm := cmPatInvert;
   8:  cm := cmPatPaint;
   9:  cm := cmSrcAnd;
   10: cm := cmSrcCopy;
   11: cm := cmSrcErase;
   12: cm := cmSrcInvert;
   13: cm := cmSrcPaint;
   14: cm := cmWhiteness;
   else
     cm := cmSrcCopy;
  end;

  // Assign the selected copymode to Image1's CopyMode property.
  imgCopyTo.Canvas.CopyMode := cm;


  GetCanvasRect(imgCopyTo, CopyToRect);
  GetCanvasRect(imgCopyFrom, CopyFromRect);

  // Now copy Image2 onto Image1 using Image1's CopyMode setting
  imgCopyTo.Canvas.CopyRect(CopyToRect, imgCopyFrom.Canvas, CopyFromRect);
end;

procedure TMainForm.btnDrawImagesClick(Sender: TObject);
begin
  DrawImages;
end;

end.
```

This project initially paints an ellipse on the two `TImage` components: `imgFromImage` and `imgToImage`. When the Copy button is clicked, `imgFromImage` is copied onto `imgToImage1` using the `CopyMode` setting specified from `cbCopyMode`.

Other Properties

`TCanvas` has other properties that we'll discuss more as we illustrate how to use them in coding techniques. This section briefly discusses these properties.

`TCanvas.ClipRect` represents a drawing region of the canvas to which drawing can be performed. You can use `ClipRect` to limit the area that can be drawn for a given canvas.

CAUTION

Initially, `ClipRect` represents the entire `Canvas` drawing area. You might be tempted to use the `ClipRect` property to obtain the bounds of a canvas. However, this could get you into trouble. `ClipRect` will not always represent the total size of its component. It can be less than the `Canvas`'s display area.

`TCanvas.Handle` gives you access to the actual device context that the `TCanvas` instance encapsulates. Device contexts are discussed later in this chapter.

`TCanvas.PenPos` is simply an X,Y coordinate location of the canvas's pen. You can change the pen's position by using the `TCanvas` methods—`MoveTo()`, `LineTo()`, `PolyLine()`, `TextOut()`, and so on.

Using the TCanvas Methods

The `TCanvas` class encapsulates many GDI drawing functions. With `TCanvas`'s methods, you can draw lines and shapes, write text, copy areas from one canvas to another, and even stretch an area on the canvas to fill a larger area.

Drawing Lines with TCanvas

`TCanvas.MoveTo()` changes `Canvas.Pen`'s drawing position on the `Canvas`'s surface. The following code, for example, moves the drawing position to the upper-left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

`TCanvas.LineTo()` draws a line on the canvas from its current position to the position specified by the parameters passed to `LineTo()`. Use `MoveTo()` with `LineTo()` to draw lines anywhere on the canvas. The following code draws a line from the upper-left position of the form's client area to the form's lower-right corner:

```
Canvas.MoveTo(0, 0);
Canvas.LineTo(ClientWidth, ClientHeight);
```

You already saw how to use the `MoveTo()` and `LineTo()` methods in the section covering the `TCanvas.Pen` property.

NOTE

Delphi now supports right-to-left-oriented text and control layouts; some controls (such as the grid) change the canvas coordinate system to flip the X axis. Therefore, if you're running Delphi on a Middle East version of Windows, `MoveTo(0,0)` may go to the top-right corner of the window.

Drawing Shapes with TCanvas

`TCanvas` offers various methods for rendering shapes to the canvas: `Arc()`, `Chord()`, `Ellipse()`, `Pie()`, `Polygon()`, `PolyLine()`, `Rectangle()`, and `RoundRect()`. To draw an ellipse in the form's client area, you would use `Canvas`'s `Ellipse()` method, as shown in the following code:

```
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

You also can fill an area on the canvas with a brush pattern specified in the `Canvas.Brush.Style` property. The following code draws an ellipse and fills the ellipse interior with the brush pattern specified by `Canvas.Brush.Style`:

```
Canvas.Brush.Style := bsCross;
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

Additionally, you saw how to add a bitmap pattern to the `TCanvas.Brush.Bitmap` property, which it uses to fill an area on the canvas. You can also use a bitmap pattern for filling in shapes. We'll demonstrate this later.

Some of `Canvas`'s other shape-drawing methods take additional or different parameters to describe the shape being drawn. The `PolyLine()` method, for example, takes an array of `TPoint` records that specify positions, or pixel coordinates, on the canvas to be connected by a line—sort of like connect the dots. A `TPoint` is a record in Delphi 5 that signifies an X,Y coordinate. A `TPoint` is defined as

```
TPoint = record
  X: Integer;
  Y: Integer;
end;
```

A Code Example for Drawing Shapes

Listing 8.4 illustrates using the various shape-drawing methods of `TCanvas`. You can find this project on the CD.

LISTING 8.4 An Illustration of Shape-Drawing Operations

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
```

```
    mmMain: TMainMenu;
    mmiShapes: TMenuItem;
    mmiArc: TMenuItem;
    mmiChord: TMenuItem;
    mmiEllipse: TMenuItem;
    mmiPie: TMenuItem;
    mmiPolygon: TMenuItem;
    mmiPolyline: TMenuItem;
    mmiRectangle: TMenuItem;
    mmiRoundRect: TMenuItem;
    N1: TMenuItem;
    mmiFill: TMenuItem;
    mmiUseBitmapPattern: TMenuItem;
    mmiPolyBezier: TMenuItem;
    procedure mmiFillClick(Sender: TObject);
    procedure mmiArcClick(Sender: TObject);
    procedure mmiChordClick(Sender: TObject);
    procedure mmiEllipseClick(Sender: TObject);
    procedure mmiUseBitmapPatternClick(Sender: TObject);
    procedure mmiPieClick(Sender: TObject);
    procedure mmiPolygonClick(Sender: TObject);
    procedure mmiPolylineClick(Sender: TObject);
    procedure mmiRectangleClick(Sender: TObject);
    procedure mmiRoundRectClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiPolyBezierClick(Sender: TObject);
  private
    FBitmap: TBitmap;
  public
    procedure ClearCanvas;
    procedure SetFillPattern;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}


procedure TMainForm.ClearCanvas;
begin
 // Clear the contents of the canvas
  with Canvas do
```

continues

LISTING 8.4 Continued

```
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClientRect);
  end;
end;

procedure TMainForm.SetFillPattern;
begin
 { Determine if shape is to be draw with a bitmap pattern in which
   case load a bitmap. Otherwise, use the brush pattern. }
  if mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := FBitmap
  else
    with Canvas.Brush do
    begin
      Bitmap := nil;
      Color := clBlue;
      Style := bsCross;
    end;
end;

procedure TMainForm.mmiFillClick(Sender: TObject);
begin
  mmiFill.Checked := not mmiFill.Checked;
  { If mmiUseBitmapPattern was checked, uncheck it set the
    brush's bitmap to nil. }
  if mmiUseBitmapPattern.Checked then
  begin
    mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
    Canvas.Brush.Bitmap := nil;
  end;
end;

procedure TMainForm.mmiUseBitmapPatternClick(Sender: TObject);
begin
  { Set mmiFil1.Checked mmiUseBitmapPattern.Checked. This will cause
    the SetFillPattern procedure to be called. However, if
    mmiUseBitmapPattern is being set, set Canvas.Brush.Bitmap to
    nil. }
  mmiUseBitmapPattern.Checked := not mmiUseBitmapPattern.Checked;
  mmiFill.Checked := mmiUseBitmapPattern.Checked;
  if not mmiUseBitmapPattern.Checked then
    Canvas.Brush.Bitmap := nil;
end;
```

```
procedure TMainForm.mmiArcClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
    Canvas.Arc(Left, Top, Right, Bottom, Right, Top, Left, Top);
end;

procedure TMainForm.mmiChordClick(Sender: TObject);
begin
  ClearCanvas;
  with ClientRect do
  begin
    if mmiFill.Checked then
      SetFillPattern;
    Canvas.Chord(Left, Top, Right, Bottom, Right, Top, Left, Top);
  end;
end;

procedure TMainForm.mmiEllipseClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
end;

procedure TMainForm.mmiPieClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Pie(0, 0, ClientWidth, ClientHeight, 50, 5, 300, 50);
end;

procedure TMainForm.mmiPolygonClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Polygon([Point(0, 0), Point(150, 20), Point(230, 130),
                  Point(40, 120)]);
end;
procedure TMainForm.mmiPolylineClick(Sender: TObject);
begin
  ClearCanvas;
```

continues

LISTING 8.4 Continued

```
  Canvas.PolyLine([Point(0, 0), Point(120, 30), Point(250, 120),
    Point(140, 200), Point(80, 100), Point(30, 30)]);
end;

procedure TMainForm.mmiRectangleClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.Rectangle(10 , 10, 125, 240);
end;

procedure TMainForm.mmiRoundRectClick(Sender: TObject);
begin
  ClearCanvas;
  if mmiFill.Checked then
    SetFillPattern;
  Canvas.RoundRect(15, 15, 150, 200, 50, 50);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FBitmap := TBitmap.Create;
  FBitMap.LoadFromFile('Pattern.bmp');
  Canvas.Brush.Bitmap := nil;
end;

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

procedure TMainForm.mmiPolyBezierClick(Sender: TObject);
begin
  ClearCanvas;
  Canvas.PolyBezier([Point(0, 100), Point(100, 0), Point(200, 50),
    Point(300, 100)]);
end;

end.
```

The main menu event handlers perform the shape-drawing functions. Two public methods, `ClearCanvas`() and `SetFillPattern()`, serve as helper functions for the event handlers. The

first eight menu items result in a shape-drawing function being drawn on the form's canvas. The last two items, "Fill" and "Use Bitmap Pattern," specify whether the shape is to be filled with a brush pattern or a bitmap pattern, respectively.

You should already be familiar with the `ClearCanvas()` functionality. The `SetFillPattern()` method determines whether to use a brush pattern or a bitmap pattern to fill the shapes drawn by the other methods. If a bitmap is selected, it's assigned to the `Canvas.Brush.Bitmap` property.

All the shape-drawing event handlers call `ClearCanvas()` to erase what was previously drawn on the canvas. They then call `SetFillPattern()` if the `mmiFill.Checked` property is set to `True`. Finally, the appropriate `TCanvas` drawing routine is called. The comments in the source discuss the purpose of each function. One method worth mentioning here is `mmiPolylineClick()`.

When you're drawing shapes, an enclosed boundary is necessary for filling an area with a brush or bitmap pattern. Although it's possible to use `PolyLine()` and `FloodFill()` to create an enclosed boundary filled with a pattern, this method is highly discouraged. `PolyLine()` is used specifically for drawing lines. If you want to draw filled polygons, call the `TCanvas.Polygon()` method. A one-pixel imperfection in where the `Polyline()` lines are drawn will allow the call to `FloodFill()` to leak out and fill the entire canvas. Drawing filled shapes with `Polygon()` uses math techniques that are immune to variations in pixel placement.

Painting Text with TCanvas

`TCanvas` encapsulates Win32 GDI routines for drawing text to a drawing surface. The following sections illustrate how to use these routines as well as how to use Win32 GDI functions that are not encapsulated by the `TCanvas` class.

Using the TCanvas Text-Drawing Routines

You used `Canvas`'s `TextOut()` function to draw text to the form's client area in previous chapters. `Canvas` has some other useful methods for determining the size, in pixels, of text contained in a string using `Canvas`'s rendered font. These functions are `TextWidth()` and `TextHeight()`. The following code determines the width and height for the string `"Delphi 5 -- Yes!"`:

```
var
  S: String;
  w, h: Integer;
begin
  S := 'Delphi 5 -- Yes!';
  w := Canvas.TextWidth(S);
  h := Canvas.TextHeight(S);
end.
```

The `TextRect()` method also writes text to the form but only within a rectangle specified by a `TRect` structure. The text not contained within the `TRect` boundaries is clipped. In the line

```
Canvas.TextRect(R,0,0,'Delphi 3.0 Yes!');
```

the string `"Delphi 5 Yes!"` is written to the canvas at location `0,0`. However, the portion of the string that falls outside the coordinates specified by `R`, a `TRect` structure, gets clipped.

Listing 8.5 illustrates using some of the text-drawing routines.

LISTING 8.5 A Unit That Illustrates Text-Drawing Operations

```
unit MainFrm;

interface

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

const
  DString = 'Delphi 5 YES!';
  DString2 = 'Delphi 5 Rocks!';

type

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiText: TMenuItem;
    mmiTextRect: TMenuItem;
    mmiTextSize: TMenuItem;
    mmiDrawTextCenter: TMenuItem;
    mmiDrawTextRight: TMenuItem;
    mmiDrawTextLeft: TMenuItem;
    procedure mmiTextRectClick(Sender: TObject);
    procedure mmiTextSizeClick(Sender: TObject);
    procedure mmiDrawTextCenterClick(Sender: TObject);
    procedure mmiDrawTextRightClick(Sender: TObject);
    procedure mmiDrawTextLeftClick(Sender: TObject);
  public
    procedure ClearCanvas;
  end;

var
  MainForm: TMainForm;

implementation
```

```
{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;

procedure TMainForm.mmiTextRectClick(Sender: TObject);
var
  R: TRect;
  TWidth, THeight: integer;
begin
  ClearCanvas;
  Canvas.Font.Size := 18;
  // Calculate the width/height of the text string
  TWidth := Canvas.TextWidth(DString);
  THeight := Canvas.TextHeight(DString);

  { Initialize a TRect structure. The height of this rectangle will
    be 1/2 the height of the text string height. This is to
    illustrate clipping the text by the rectangle drawn }
  R := Rect(1, THeight div 2, TWidth + 1, THeight+(THeight div 2));
  // Draw a rectangle based on the text sizes
  Canvas.Rectangle(R.Left-1, R.Top-1, R.Right+1, R.Bottom+1);
  // Draw the Text within the rectangle
  Canvas.TextRect(R,0,0,DString);
end;

procedure TMainForm.mmiTextSizeClick(Sender: TObject);
begin
  ClearCanvas;
  with Canvas do
  begin
    Font.Size := 18;
    TextOut(10, 10, DString);
    TextOut(50, 50, 'TextWidth = '+IntToStr(TextWidth(DString)));
    TextOut(100, 100, 'TextHeight = '+IntToStr(TextHeight(DString)));
  end;
end;

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

continues

LISTING 8.5 Continued

```
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels }
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text centered by specifying the dt_Center option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Center);
end;

procedure TMainForm.mmiDrawTextRightClick(Sender: TObject);
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text right-aligned by specifying the dt_Right option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Right);
end;

procedure TMainForm.mmiDrawTextLeftClick(Sender: TObject);
var
 R: TRect;
begin
  ClearCanvas;
  Canvas.Font.Size := 10;
  R := Rect(10, 10, 80, 100);
  // Draw a rectangle to surround the TRect boundaries by 2 pixels
  Canvas.Rectangle(R.Left-2, R.Top-2, R.Right+2, R.Bottom+2);
  // Draw text left-aligned by specifying the dt_Left option
  DrawText(Canvas.Handle, PChar(DString2), -1, R, dt_WordBreak or dt_Left);
end;

end.
```

Like the other projects, this project contains the `ClearCanvas()` method to erase the contents of the form's canvas.

The various methods of the main form are event handlers to the form's main menu.

The `mmiTextRectClick()` method illustrates how to use the `TCanvas.TextRect()` method. It determines the text width and height and draws the text inside a rectangle the height of the original text size. Its output is shown in Figure 8.10.

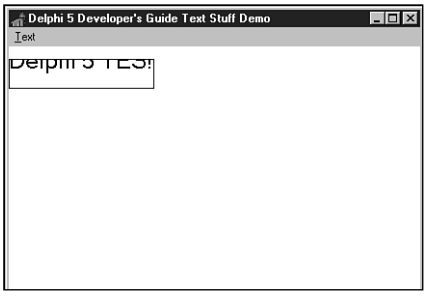

FIGURE 8.10

The output of `mmiTextRectClick()`.

`mmiTextSizeClick()` shows how to determine the size of a text string using the `TCanvas.TextWidth()` and `TCanvas.TextHeight()` methods. Its output is shown in Figure 8.11.

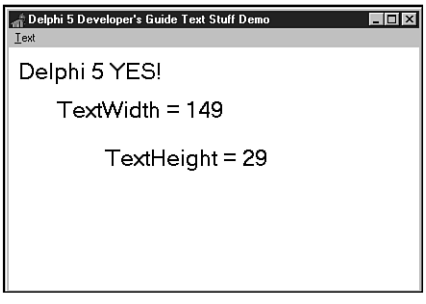

FIGURE 8.11

The output of `mmiTextSizeClick()`.

Using Non-TCanvas GDI Text Output Routines

In the sample project, the `mmiDrawTextCenter()`, `mmiDrawTextRight()`, and `mmiDrawTextLeft()` methods all illustrate using the Win32 GDI function `DrawText()`. `DrawText()` is a GDI function not encapsulated by the `TCanvas` class.

This code illustrates how Delphi 5's encapsulation of Win32 GDI through the `TCanvas` class doesn't prevent you from making use of the abundant Win32 GDI functions. Instead, `TCanvas` really just simplifies using the more common routines while still enabling you to call any Win32 GDI function you might need.

If you look at the various GDI functions such as `BitBlt()` and `DrawText()`, you'll find that one of the required parameters is a DC, or *device context*. The device context is accessible through the canvas's `Handle` property. `TCanvas.Handle` is the DC for that canvas.

Device Contexts

Device contexts (DCs) are handles provided by Win32 to identify a Win32 application's connection to an output device such as a monitor, printer, or plotter through a device driver. In traditional Windows programming, you're responsible for requesting a DC whenever you need to paint to a window's surface; then, when done, you have to return the DC back to Windows. Delphi 5 simplifies the management of DCs by encapsulating DC management in the `TCanvas` class. In fact, `TCanvas` even caches your DC, saving it for later so that requests to Win32 occur less often—thus, speeding up your program's overall execution.

To see how to use `TCanvas` with a Win32 GDI function, you used the GDI routine `DrawText()` to output text with advanced formatting capabilities. `DrawText` takes the following five parameters:

Parameter	*Description*
`DC`	Device context of the drawing surface.
`Str`	Pointer to a buffer containing the text to be drawn. This must be a null-terminated string if the `Count` parameter is `-1`.
`Count`	Number of bytes in `Str`. If this value is `-1`, `Str` is a pointer to a null-terminated string.
`Rect`	Pointer to a `TRect` structure containing the coordinates of the rectangle in which the text is formatted.
`Format`	A bit field that contains flags specifying the various formatting options for `Str`.

In the example, you initialize a `TRect` structure using the `Rect()` function. You'll use the structure to draw a rectangle around the text drawn with the `DrawText()` function. Each of the three methods passes a different set of formatting flags to the `DrawText()` function. The `dt_WordBreak` and `dt_Center` formatting flags are passed to the `DrawText()` function to center the text in the rectangle specified by the `TRect` variable `R`. `dt_WordBreak`, `OR`'ed with `dt_Right`, is used to right-justify the text in the rectangle. Likewise, `dt_WordBreak`, `OR`'ed with `dt_Left`, left-justifies the text. The `dt_WordBreak` specifier word-wraps the text within the width given by the rectangle parameter and modifies the rectangle height to bind the text after being word-wrapped.

The output of `mmiDrawTextCenterClick()` is shown in Figure 8.12.

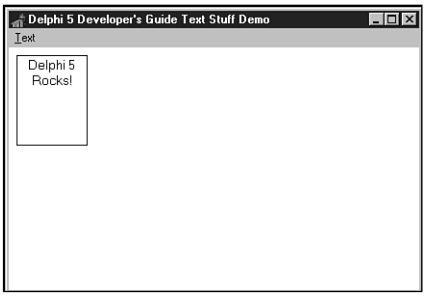

FIGURE 8.12

The output of `mmiDrawTextCenterClick()`.

`TCanvas` also has the methods `Draw()`, `Copy()`, `CopyRect()`, and `StretchDraw()`, which enable you to draw, copy, expand, and shrink an image or a portion of an image to another canvas. You'll use `CopyRect()` when we show you how to create a paint program later in this chapter. Also, Chapter 16, "MDI Applications," shows you how to use the `StretchDraw()` method to stretch a bitmap image onto the client area of a form.

Coordinate Systems and Mapping Modes

Most GDI drawing routines require a set of coordinates that specify the location where drawing is to occur. These coordinates are based on a unit of measurement, such as the pixel. Additionally, GDI routines assume an orientation for the vertical and horizontal axis—that is, how increasing or decreasing the values of the X,Y coordinates moves the position at which drawing occurs. Win32 relies on two factors to perform drawing routines. These are the Win32 coordinates system and the mapping mode of the area that's being drawn to.

Win32 coordinates systems are, generally, no different from any other coordinates system. You define a coordinate for an X,Y axis, and Win32 plots that location to a point on your drawing surface based on a given orientation. Win32 uses three coordinates systems to plot areas on drawing surfaces called the *device*, *logical*, and *world* coordinates. Windows 95 doesn't support world transformations (bitmap rotation, shearing, twisting, and so on). We'll cover the first two modes in this chapter.

Device Coordinates

Device coordinates, as the name implies, refer to the device on which Win32 is running. Its measurements are in pixels, and the orientation is such that the horizontal and vertical axes increase from left to right and top to bottom. For example, if you're running Windows on a 640×480 pixel display, the coordinates at the top-left corner on your device are (0,0), whereas the bottom-right coordinates are (639,479).

Logical Coordinates

Logical coordinates refer to the coordinates system used by any area in Win32 that has a device context or DC such as a screen, a form, or a form's client area. The difference between device and logical coordinates is explained in a moment. The screen, form window, and form client-area coordinates are explained first.

Screen Coordinates

Screen coordinates refer to the display device; therefore, it follows that coordinates are based on pixel measurements. On a 640×480 display, `Screen.Width` and `Screen.Height` are also 640 and 480 pixels, respectively. To obtain a device context for the screen, use the Win32 API function `GetDC()`. You must match any function that retrieves a device context with a call to `ReleaseDC()`. The following code illustrates this:

```
var
  ScreenDC: HDC;
begin
  Screen DC := GetDC(0);
  try
    { Do whatever you need to do with ScreenDC }
  finally
     ReleaseDC(0, ScreenDC);
  end;
end;
```

Form Coordinates

Form coordinates are synonymous with the term *window coordinates* and refer to an entire form or window, including the caption bar and borders. Delphi 5 doesn't provide a DC to the form's drawing area through a form's property, but you can obtain one by using the Win32 API function `GetWindowDC()`, as follows:

```
MyDC := GetWindowDC(Form1.Handle);
```

This function returns the DC for the window handle passed to it.

NOTE

You can use a `TCanvas` object to encapsulate the device contexts obtained from the calls to `GetDC()` and `GetWindowDC()`. This enables you to use the `TCanvas` methods against those device contexts. You just need to create a `TCanvas` instance and then assign the result of `GetDC()` or `GetWindowDC()` to the `TCanvas.Handle` property. This

works because the `TCanvas` object takes ownership of the handle you assign to it, and it will release the DC when the canvas is freed. The following code illustrates this technique:

```
var
  c: TCanvas;
begin
  c := TCanvas.Create;
  try
    c.Handle := GetDC(0);
    c.TextOut(10, 10, 'Hello World');
  finally
    c.Free;
  end;
end;
```

A form's *client-area coordinates* refer to a form's client area whose DC is the `Handle` property of the form's `Canvas` and whose measurements are obtained from `Canvas.ClientWidth` and `Canvas.ClientHeight`.

Coordinate Mapping

So why not just use device coordinates instead of logical coordinates when performing drawing routines? Examine the following line of code:

```
Form1.Canvas.TextOut(0, 0, 'Upper Left Corner of Form');
```

This line places the string at the upper-left corner of the form. The coordinates (0,0) map to the position (0,0) in the form's device context—logical coordinates. However, the position (0,0) for the form is completely different in device coordinates and depends on where the form is located on your screen. If the form just happens to be located at the upper-left corner of your screen, the form's coordinates (0,0) may in fact map to (0,0) in device coordinates. However, as you move the form to another location, the form's position (0,0) will map to a completely different location on the device.

TIP

You can obtain a point based on device coordinates from the point as it's represented in logical coordinates, and vice versa, using the Win32 API functions `ClientToScreen()` and `ScreenToClient()`, respectively. These are also `TControl` methods. Note that this works only with screen DCs associated with a visible control.

continues

For printer or metafile DCs that are not screen based, convert logical pixels to device pixels by using the `LPtoDP()` Win32 function. Also see `DPtoLP() in the Win32 online help`.

Underneath the call to `Canvas.TextOut()`, Win32 does actually use device coordinates. For Win32 to do this, it must "map" the logical coordinates of the DC being drawn to, to device coordinates. It does this using the mapping mode associated with the DC.

Another reason for using logical coordinates is that you might not want to use pixels to perform drawing routines. Perhaps you want to draw using inches or millimeters. Win32 enables you to change the unit with which you perform your drawing routines by changing its mapping mode, as you'll see in a moment.

Mapping modes define two attributes for the DC: the translation that Win32 uses to convert logical units to device units, and the orientation of the X,Y axis for the DC.

NOTE

It might not seem apparent that drawing routines, mapping modes, orientation, and so on are associated with a DC because, in Delphi 5, you use the canvas to draw. Remember that `TCanvas` is a wrapper for a DC. This becomes obvious when comparing Win32 GDI routines to their equivalent `Canvas` routines. Here are examples:

`Canvas` routine: `Canvas.Rectangle(0, 0, 50, 50));`

GDI routine: `Rectangle(ADC, 0, 0, 50, 50);`

When you're using the GDI routine, a DC is passed to the function, whereas the canvas's routine uses the DC that it encapsulates.

Win32 enables to you define the mapping mode for a DC or `TCanvas.Handle`. In fact, Win32 defines eight mapping modes you can use. These mapping modes, along with their attributes, are shown in Table 8.4. The sample project in the next section illustrates more about mapping modes.

TABLE 8.4 Win32 Mapping Modes

Mapping Mode	*Logical Unit Size*	*Orientation (X,Y)*
`MM_ANISOTROPIC`	Arbitrary (x <> y) or (x = y)	Definable/definable
`MM_HIENGLISH`	0.001 inch	Right/up

Mapping Mode	*Logical Unit Size*	*Orientation (X,Y)*
MM_HIMETRIC	0.01 mm	Right/up
MM_ISOTROPIC	arbitrary (x = y)	Definable/definable
MM_LOENGLISH	0.01 inch	Right/up
MM_LOMETRIC	0.1 mm	Right/up
MM_TEXT	1 pixel	Right/down
MM_TWIPS	1/1440 inch	Right/up

Win32 defines a few functions that enable you to change or retrieve information about the mapping modes for a given DC. Here's a summary of these functions:

- `SetMapMode()`. Sets the mapping mode for a given device context.
- `GetMapMode()`. Gets the mapping mode for a given device context.
- `SetWindowOrgEx()`. Defines an window origin (point 0,0) for a given DC.
- `SetViewPortOrgEx()`. Defines a viewport origin (point 0,0) for a given DC.
- `SetWindowExtEx()`. Defines the X,Y extents for a given window DC. These values are used in conjunction with the viewport X,Y extents to perform translation from logical units to device units.
- `SetViewPortExtEx()`. Defines the X,Y extents for a given viewport DC. These values are used in conjunction with the window X,Y extents to perform translation from logical units to device units.

Notice that these functions contain either the word `Window` or `ViewPort`. The window or viewport is simply a means by which Win32 GDI can perform the translation from logical to device units. The functions with `Window` refer to the logical coordinate system, whereas those with `ViewPort` refer to the device coordinates system. With the exception of the `MM_ANISOTROPIC` and `MM_ISOTROPIC` mapping modes, you don't have to worry about this much. In fact, Win32 uses the `MM_TEXT` mapping mode by default.

NOTE

`MM_TEXT` is the default mapping mode, and it maps logical coordinates 1:1 with device coordinates. So, you're always using device coordinates on all DCs, unless you change the mapping mode. There are some API functions where this is significant: Font heights, for example, are always specified in device pixels, not logical pixels.

Setting the Mapping Mode

You'll notice that each mapping mode uses a different logical unit size. In some cases, it might be convenient to use a different mapping mode for that reason. For example, you might want to display a line 2 inches wide, regardless of the resolution of your output device. In this instance, `MM_LOENGLISH` would be a good candidate for a mapping mode to use.

As an example of drawing a 1-inch rectangle to the form, you first change the mapping mode for `Form1.Canvas.Handle` to `MM_HIENGLISH` or `MM_LOENGLISH`:

```
SetMapMode(Canvas.Handle, MM_LOENGLISH);
```

Then you draw the rectangle using the appropriate units of measurement for a 1-inch rectangle. Because `MM_LOENGLISH` uses 1/100 inch, you simply pass the value `100`, as follows (this will be illustrated further in a later example):

```
Canvas.Rectangle(0, 0, 100, 100);
```

Because `MM_TEXT` uses pixels as its unit of measurement, you can use the Win32 API function `GetDeviceCaps()` to retrieve the information you need to perform translation from pixels to inches or millimeters. Then you can do your own calculations if you want. This is demonstrated in Chapter 10, "Printing in Delphi 5." Mapping modes are a way to let Win32 do the work for you. Note, however, that you'll most likely never be able to get exact measurements for screen displays. There are a few reasons for this: Windows cannot record the display size of the screen—it must guess. Also, Windows typically inflates display scales to improve text readability on relatively chunky monitors. So, for example, a 10-point font on a screen is about as tall as a 12- to 14-point font on paper.

Setting the Window/Viewport Extents

The `SetWindowExtEx()` and `SetViewPortExtEx()` functions enable you to define how Win32 translates logical units to device units. These functions have an effect only when the window's mapping mode is either `MM_ANISOTROPIC` or `MM_ISOTROPIC`. They are ignored otherwise. Therefore, the following lines of code mean that one logical unit requires two device units (pixels):

```
SetWindowExtEx(Canvas.Handle, 1, 1, nil)
SetViewportExtEx(Canvas.Handle, 2, 2, nil);
```

Likewise, these lines of code mean that five logical units require 10 device units:

```
SetWindowExtEx(Canvas.Handle, 5, 5, nil)
SetViewportExtEx(Canvas.Handle, 10, 10, nil);
```

Notice that this is exactly the same as the previous example. Both have the same effect of having a 1:2 ratio of logical to device units. Here's an example of how this may be used to change the units for a form:

```
SetWindowExtEx(Canvas.Handle, 500, 500, nil)
SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
```

NOTE

Changing the mapping mode for a device context represented by a VCL canvas is not "sticky," which means that it may revert back to its original mode. Generally, the map mode must be set within the handler doing the actual drawing.

This enables to you work with a form whose client width and height are 500×500 units (not pixels) despite any resizing of the form.

The `SetWindowOrgEx()` and `SetViewPortOrgEx()` functions enable you to relocate the origin or position (0,0), which, by default, is at the upper-left corner of a form's client area in the `MM_TEXT` mapping mode. Typically, you just modify the viewport origin. For example, the following line sets up a four-quadrant coordinate system like the one illustrated in Figure 8.13:

```
SetViewportOrgEx(Canvas.Handle, ClientWidth div 2, ClientHeight div 2, nil);
```


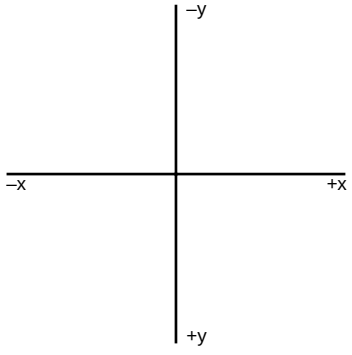

FIGURE 8.13

A four-quadrant coordinate system.

Notice that we pass a `nil` value as the last parameter in the `SetWindowOrgEx()`, `SetViewPortOrgEx()`, `SetWindowExtEx()`, and `SetViewPortExtEx()` functions. The `SetWindowOrgEx()` and `SetViewPortOrgEx()` functions take a `TPoint` variable that gets assigned the last origin value so that you can restore the origin for the DC, if necessary. The `SetWindowExtEx()` and `SetViewPortExtEx()` functions take a `TSize` structure to store the original extents for the DC for the same reason.

Mapping Mode Example Project

Listing 8.6 shows you the unit for a project. This project illustrates how to set mapping modes, window and viewport origins, and windows and viewport extents. It also illustrates how to draw various shapes using `TCanvas` methods. You can load this project from the CD.

LISTING 8.6 An Illustration of Mapping Modes

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, DB, DBCGrids, DBTables;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiMappingMode: TMenuItem;
    mmiMM_ISOTROPIC: TMenuItem;
    mmiMM_ANSITROPIC: TMenuItem;
    mmiMM_LOENGLISH: TMenuItem;
    mmiMM_HIINGLISH: TMenuItem;
    mmiMM_LOMETRIC: TMenuItem;
    mmiMM_HIMETRIC: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure mmiMM_ISOTROPICClick(Sender: TObject);
    procedure mmiMM_ANSITROPICClick(Sender: TObject);
    procedure mmiMM_LOENGLISHClick(Sender: TObject);
    procedure mmiMM_HIINGLISHClick(Sender: TObject);
    procedure mmiMM_LOMETRICClick(Sender: TObject);
    procedure mmiMM_HIMETRICClick(Sender: TObject);
  public
    MappingMode: Integer;
    procedure ClearCanvas;
    procedure DrawMapMode(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.ClearCanvas;
begin
```

```
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clWhite;
    FillRect(ClipRect);
  end;
end;

procedure TMainForm.DrawMapMode(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  Canvas.TextOut(0, 0, (Sender as TMenuItem).Caption);

  // Set mapping mode to MM_LOENGLISH and save the previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MappingMode);
  try
    // Set the viewport org to left, bottom
    SetViewPortOrgEx(Canvas.Handle, 0, ClientHeight, nil);
    { Draw some shapes to illustrate drawing shapes with different
      mapping modes specified by MappingMode }
    Canvas.Rectangle(0, 0, 200, 200);
    Canvas.Rectangle(200, 200, 400, 400);
    Canvas.Ellipse(200, 200, 400, 400);
    Canvas.MoveTo(0, 0);
    Canvas.LineTo(400, 400);
    Canvas.MoveTo(0, 200);
    Canvas.LineTo(200, 0);
  finally
    // Restore previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  MappingMode := MM_TEXT;
end;

procedure TMainForm.mmiMM_ISOTROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  // Set mapping mode to MM_ISOTROPIC and save the previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ISOTROPIC);
```

continues

LISTING 8.6 Continued

```
  try
    // Set the window extent to 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Set the Viewport extent to the Window's client area
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Set the ViewPortOrg to the center of the client area
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
       ClientHeight div 2, nil);
    // Draw a rectangle based on current settings
    Canvas.Rectangle(0, 0, 250, 250);
    { Set the viewport extent to a different value, and
      draw another rectangle. continue to do this three
      more times so that a rectangle is draw to represent
      the plane in a four-quadrant square }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Draw an ellipse in the center of the client area
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    // Restore the previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.mmiMM_ANSITROPICClick(Sender: TObject);
var
  PrevMapMode: Integer;
begin
  ClearCanvas;
  // Set the mapping mode to MM_ANISOTROPIC and save the
  // previous mapping mode
  PrevMapMode := SetMapMode(Canvas.Handle, MM_ANISOTROPIC);
  try
    // Set the window extent to 500 x 500
    SetWindowExtEx(Canvas.Handle, 500, 500, nil);
    // Set the Viewport extent to that of the Window's client area
    SetViewportExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
    // Set the ViewPortOrg to the center of the client area
    SetViewportOrgEx(Canvas.Handle, ClientWidth div 2,
       ClientHeight div 2, nil);
```

```
   // Draw a rectangle based on current settings
    Canvas.Rectangle(0, 0, 250, 250);
    { Set the viewport extent to a different value, and
      draw another rectangle. continue to do this three
      more times so that a rectangle is draw to represent
      the plane in a four-quadrant square }
    SetViewportExtEx(Canvas.Handle, ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, -ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);

    SetViewportExtEx(Canvas.Handle, -ClientWidth, ClientHeight, nil);
    Canvas.Rectangle(0, 0, 250, 250);
    // Draw an ellipse in the center of the client area
    Canvas.Ellipse(-50, -50, 50, 50);
  finally
    //Restore the previous mapping mode
    SetMapMode(Canvas.Handle, PrevMapMode);
  end;
end;

procedure TMainForm.mmiMM_LOENGLISHClick(Sender: TObject);
begin
  MappingMode := MM_LOENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIINGLISHClick(Sender: TObject);
begin
  MappingMode := MM_HIENGLISH;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_LOMETRICClick(Sender: TObject);
begin
  MappingMode := MM_LOMETRIC;
  DrawMapMode(Sender);
end;

procedure TMainForm.mmiMM_HIMETRICClick(Sender: TObject);
 begin
  MappingMode := MM_HIMETRIC;
  DrawMapMode(Sender);
end;

end.
```

The main form's field `MappingMode` is used to hold the current mapping mode that's initialized in the `FormCreate()` method to `MM_TEXT`. This variable gets set whenever the `MMLOENGLISH1Click()`, `MMHIENGLISH1Click()`, `MMLOMETRIC1Click()`, and `MMHIMETRIC1Click()` methods are invoked from their respective menus. These methods then call the method `DrawMapMode()`, which sets the main form's mapping mode to that specified by `MappingMode`. It then draws some shapes and lines using constant values to specify their sizes. When different mapping modes are used when drawing the shapes, they'll be sized differently on the form because the measurements used are used in the context of the specified mapping mode. Figures 8.14 and 8.15 illustrate `DrawMapMode()`'s output for the `MM_LOENGLISH` and `MM_LOMETRIC` mapping modes.

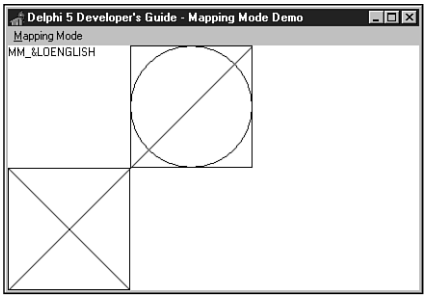

FIGURE 8.14

`DrawMapMode()` *output using* `MM_LOENGLISH` *mapping mode.*

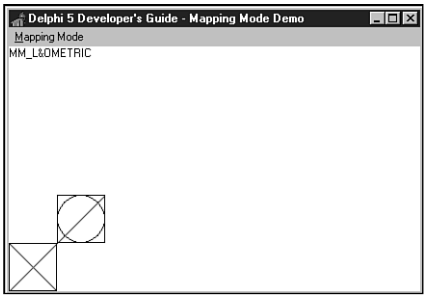

FIGURE 8.15

`DrawMapMode()` *output using* `MM_LOMETRIC` *mapping mode.*

The `mmiMM_ISOTROPICClick()` method illustrates drawing with the form's canvas in the `MM_ISOTROPIC` mode. This method first sets the mapping mode and then sets the canvas's viewport extent to that of the form's client area. The origin is then set to the center of the form's client area, which allows all four quadrants of the coordinate system to be viewed.

The method then draws a rectangle in each plane and an ellipse in the center of the client area. Notice how you can use the same values in the parameters passed to `Canvas.Rectangle()` yet draw to different areas of the canvas. This is accomplished by passing negative values to the X parameter, Y parameter, or both parameters passed to `SetViewPortExt()`.

The `mmiMM_ANISOTROPICClick()` method performs the same operations except it uses the `MM_ANISOTROPIC` mode. The purpose of showing both is to illustrate the principle difference between the `MM_ISOTROPIC` and `MM_ANISOTROPIC` mapping modes.

Using the `MM_ISOTROPIC` mode, Win32 ensures that the two axes use the same physical size and makes the necessary adjustments to see this is the case. The `MM_ANISOTROPIC` mode, however, uses physical dimensions that might not be equal. Figures 8.16 and 8.17 illustrate this more clearly. You can see that the `MM_ISOTROPIC` mode ensures equality with the two axes, whereas the same code using the `MM_ANISOTROPIC` mode does not ensure equality. In fact, the `MM_ISOTROPIC` mode further guarantees that the square logical coordinates will be mapped to device coordinates such that squareness will be preserved, even if the device coordinates system is not square.

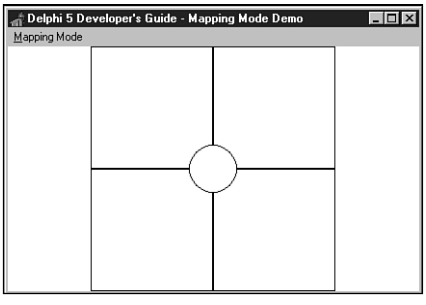

FIGURE 8.16

`MM_ISOTROPIC` *mapping mode output.*

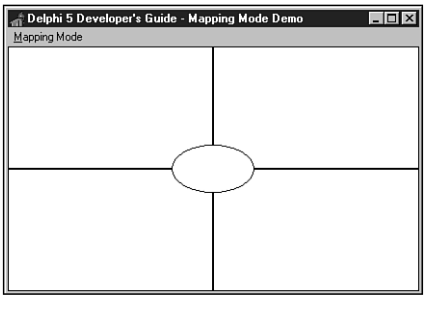

FIGURE 8.17

`MM_ANISOTROPIC` *mapping mode output.*

Creating a Paint Program

The paint program shown here uses several advanced techniques in working with GDI and graphics images. You can find this project on the CD as `DDGPaint.dpr`. Listing 8.7 shows the source code for this project.

LISTING 8.7 The Paint Program: `DDGPaint`

```
unit MainFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, ExtCtrls, ColorGrd, StdCtrls, Menus,
  ComCtrls;

const
  crMove = 1;
type

  TDrawType = (dtLineDraw, dtRectangle, dtEllipse, dtRoundRect,
               dtClipRect, dtCrooked);

  TMainForm = class(TForm)
    sbxMain: TScrollBox;
    imgDrawingPad: TImage;
    pnlToolBar: TPanel;
    sbLine: TSpeedButton;
    sbRectangle: TSpeedButton;
    sbEllipse: TSpeedButton;
    sbRoundRect: TSpeedButton;
    pnlColors: TPanel;
    cgDrawingColors: TColorGrid;
    pnlFgBgBorder: TPanel;
    pnlFgBgInner: TPanel;
    Bevel1: TBevel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N2: TMenuItem;
    mmiSaveAs: TMenuItem;
    mmiSaveFile: TMenuItem;
    mmiOpenFile: TMenuItem;
    mmiNewFile: TMenuItem;
    mmiEdit: TMenuItem;
```

```
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    sbRectSelect: TSpeedButton;
    SaveDialog: TSaveDialog;
    OpenDialog: TOpenDialog;
    stbMain: TStatusBar;
    pbPasteBox: TPaintBox;
    sbFreeForm: TSpeedButton;
    RgGrpFillOptions: TRadioGroup;
    cbxBorder: TCheckBox;
    procedure FormCreate(Sender: TObject);
    procedure sbLineClick(Sender: TObject);
    procedure imgDrawingPadMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure imgDrawingPadMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure cgDrawingColorsChange(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiSaveFileClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure mmiNewFileClick(Sender: TObject);
    procedure mmiOpenFileClick(Sender: TObject);
    procedure mmiEditClick(Sender: TObject);
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure pbPasteBoxMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure pbPasteBoxPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure RgGrpFillOptionsClick(Sender: TObject);
  public
    { Public declarations }
    MouseOrg: TPoint;    // Stores mouse information
    NextPoint: TPoint;   // Stores mouse information
    Drawing: Boolean;    // Drawing is being performed flag
    DrawType: TDrawType; // Holds the draw type information: TDrawType
    FillSelected,        // Fill shapes flag
```

continues

LISTING 8.7 Continued

```
    BorderSelected: Boolean;  // Draw Shapes with no border flag
    EraseClipRect: Boolean;     // Specifies whether or not to erase the
                                // clipping rectangle
    Modified: Boolean;     // Image modified flag
    FileName: String;      // Holds the filename of the image
    OldClipViewHwnd: Hwnd; // Holds the old clipboard view window
    { Paste Image variables }
    PBoxMoving: Boolean;   // PasteBox is moving flag
    PBoxMouseOrg: TPoint;  // Stores mouse coordinates for PasteBox
    PasteBitMap: TBitmap;  // Stores a bitmap image of the pasted data
    Pasted: Boolean;       // Data pasted flag
    LastDot: TPoint;       // Hold the TPoint coordinate for performing
                           // free line drawing
    procedure DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
    { This procedure paints the image specified by the DrawType field
      to imgDrawingPad }
    procedure SetDrawingStyle;
    { This procedure sets various Pen/Brush styles based on values
      specified by the form's controls. The Panels and color grid is
      used to set these values }
    procedure CopyPasteBoxToImage;
    { This procedure copies the data pasted from the Windows clipboard
      onto the main image component imgDrawingPad }
    procedure WMDrawClipBoard(var Msg: TWMDrawClipBoard);
       message WM_DRAWCLIPBOARD;
    { This message handler captures the WM_DRAWCLIPBOARD messages
      which is sent to all windows that have been added to the clipboard
      viewer chain. An application can add itself to the clipboard viewer
      chain by using the SetClipBoardViewer() Win32 API function as
      is done in FormCreate() }
    procedure CopyCut(Cut: Boolean);
    { This method copies a portion of the main image, imgDrawingPad,
       to the Window's clipboard. }
  end;

var
  MainForm: TMainForm;

implementation
uses ClipBrd, Math;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
{ This method sets the form's field to their default values. It then
```

```
  creates a bitmap for the imgDrawingPad.  This is the image on which
  drawing is done. Finally, it adds this application as part of the
  Windows clipboard viewer chain by using the SetClipBoardViewer()
  function. This makes enables the form to get WM_DRAWCLIPBOARD messages
  which are sent to all windows in the clipboard viewer chain whenever
  the clipboard data is modified. }
begin
  Screen.Cursors[crMove] := LoadCursor(hInstance, 'MOVE');

  FillSelected   := False;
  BorderSelected := True;

  Modified := False;
  FileName := '';
  Pasted := False;
  pbPasteBox.Enabled := False;

  // Create a bitmap for imgDrawingPad and set its boundaries
  with imgDrawingPad do
  begin
    SetBounds(0, 0, 600, 400);
    Picture.Graphic := TBitMap.Create;
    Picture.Graphic.Width := 600;
    Picture.Graphic.Height := 400;
  end;
  // Now create a bitmap image to hold pasted data
  PasteBitmap := TBitmap.Create;
  pbPasteBox.BringToFront;
  { Add the form to the Windows clipboard viewer chain. Save the handle
    of the next window in the chain so that it may be restored by the
    ChangeClipboardChange() Win32 API function in this form's
    FormDestroy() method. }
  OldClipViewHwnd := SetClipBoardViewer(Handle);
end;

procedure TMainForm.WMDrawClipBoard(var Msg: TWMDrawClipBoard);
begin
  { This method will be called whenever the clipboard data
    has changed. Because the main form was added to the clipboard
    viewer chain, it will receive the WM_DRAWCLIPBOARD message
    indicating that the clipboard's data was changed. }
  inherited;
  { Make sure that the data contained on the clipboard is actually
    bitmap data. }
  if ClipBoard.HasFormat(CF_BITMAP) then
    mmiPaste.Enabled := True
```

continues

LISTING 8.7 Continued

```
  else
    mmiPaste.Enabled := False;
  Msg.Result := 0;
end;


procedure TMainForm.DrawToImage(TL, BR: TPoint; PenMode: TPenMode);
{ This method performs the specified drawing operation. The
  drawing operation is specified by the DrawType field }
begin
  with imgDrawingPad.Canvas do
  begin
    Pen.Mode := PenMode;

    case DrawType of
      dtLineDraw:
        begin
          MoveTo(TL.X, TL.Y);
          LineTo(BR.X, BR.Y);
        end;
      dtRectangle:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
      dtEllipse:
        Ellipse(TL.X, TL.Y, BR.X, BR.Y);
      dtRoundRect:
        RoundRect(TL.X, TL.Y, BR.X, BR.Y,
          (TL.X - BR.X) div 2, (TL.Y - BR.Y) div 2);
      dtClipRect:
        Rectangle(TL.X, TL.Y, BR.X, BR.Y);
    end;
  end;
end;

procedure TMainForm.CopyPasteBoxToImage;
{ This method copies the image pasted from the Windows clipboard onto
  imgDrawingPad. It first erases any bounding rectangle drawn by PaintBox
  component, pbPasteBox. It then copies the data from pbPasteBox onto
  imgDrawingPad at the location where pbPasteBox has been dragged
  over imgDrawingPad. The reason we don't copy the contents of
  pbPasteBox's canvas and use PasteBitmap's canvas instead, is because
  when a portion of pbPasteBox is dragged out of the viewable area,
  Windows does not paint the portion pbPasteBox not visible. Therefore,
  it is necessary to the pasted bitmap from the off-screen bitmap }
var
```

```
  SrcRect, DestRect: TRect;
begin
  // First, erase the rectangle drawn by pbPasteBox
  with pbPasteBox do
  begin
    Canvas.Pen.Mode := pmNotXOR;
    Canvas.Pen.Style := psDot;
    Canvas.Brush.Style := bsClear;
    Canvas.Rectangle(0, 0, Width, Height);
    DestRect := Rect(Left, Top, Left+Width, Top+Height);
    SrcRect := Rect(0, 0, Width, Height);
  end;
  { Here we must use the PasteBitmap instead of the pbPasteBox because
    pbPasteBox will clip anything outside if the viewable area. }
  imgDrawingPad.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
  pbPasteBox.Visible := false;
  pbPasteBox.Enabled := false;
  Pasted := False;  // Pasting operation is complete
end;

procedure TMainForm.imgDrawingPadMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Modified := True;
  // Erase the clipping rectangle if one has been drawn
  if (DrawType = dtClipRect) and EraseClipRect then
    DrawToImage(MouseOrg, NextPoint, pmNotXOR)
  else if (DrawType = dtClipRect) then
    EraseClipRect := True; // Re-enable cliprect erasing

  { If an bitmap was pasted from the clipboard, copy it to the
    image and remove the PaintBox. }
  if Pasted then
    CopyPasteBoxToImage;

  Drawing := True;
  // Save the mouse information
  MouseOrg := Point(X, Y);
  NextPoint := MouseOrg;
  LastDot := NextPoint;   // Lastdot is updated as the mouse moves
  imgDrawingPad.Canvas.MoveTo(X, Y);
end;

procedure TMainForm.imgDrawingPadMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
{ This method determines the drawing operation to be performed and
```

continues

LISTING 8.7 Continued

```
  either performs free form line drawing, or calls the
  DrawToImage method which draws the specified shape }
begin
  if Drawing then
  begin
    if DrawType = dtCrooked then
    begin
      imgDrawingPad.Canvas.MoveTo(LastDot.X, LastDot.Y);
      imgDrawingPad.Canvas.LineTo(X, Y);
      LastDot := Point(X,Y);
    end
    else begin
      DrawToImage(MouseOrg, NextPoint, pmNotXor);
      NextPoint := Point(X, Y);
      DrawToImage(MouseOrg, NextPoint, pmNotXor)
    end;
  end;
  // Update the status bar with the current mouse location
  stbMain.Panels[1].Text := Format('X: %d, Y: %D', [X, Y]);
end;

procedure TMainForm.imgDrawingPadMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  { Prevent the clipping rectangle from destroying the images already
    on the image }
    if not (DrawType = dtClipRect) then
      DrawToImage(MouseOrg, Point(X, Y), pmCopy);
  Drawing := False;
end;

procedure TMainForm.sbLineClick(Sender: TObject);
begin
  // First erase the cliprect if current drawing type
  if DrawType = dtClipRect then
     DrawToImage(MouseOrg, NextPoint, pmNotXOR);

  { Now set the DrawType field to that specified by the TSpeedButton
    invoking this method. The TSpeedButton's Tag values match a
    specific TDrawType value which is why the typecasting below
    successfully assigns a valid TDrawType value to the DrawType field. }
  if Sender is TSpeedButton then
    DrawType := TDrawType(TSpeedButton(Sender).Tag);
```

```
  // Now make sure the dtClipRect style doesn't erase previous drawings
  if DrawType = dtClipRect then begin
    EraseClipRect := False;
  end;
  // Set the drawing style
  SetDrawingStyle;
end;

procedure TMainForm.cgDrawingColorsChange(Sender: TObject);
{ This method draws the rectangle representing fill and border colors
  to indicate the users selection of both colors. pnlFgBgInner and
  pnlFgBgBorder are TPanels arranged one on to of the other for the
  desired effect }
begin
  pnlFgBgBorder.Color := cgDrawingColors.ForeGroundColor;
  pnlFgBgInner.Color := cgDrawingColors.BackGroundColor;
  SetDrawingStyle;
end;


procedure TMainForm.SetDrawingStyle;
{  This method sets the various drawing styles based on the selections
   on the pnlFillStyle TPanel for Fill and Border styles }
begin
  with imgDrawingPad do
  begin
    if DrawType = dtClipRect then
    begin
      Canvas.Pen.Style := psDot;
      Canvas.Brush.Style := bsClear;
      Canvas.Pen.Color := clBlack;
    end

    else if FillSelected then
      Canvas.Brush.Style := bsSolid
    else
      Canvas.Brush.Style := bsClear;

    if BorderSelected then
      Canvas.Pen.Style := psSolid
    else
      Canvas.Pen.Style := psClear;


    if FillSelected and (DrawType <> dtClipRect) then
     Canvas.Brush.Color := pnlFgBgInner.Color;
```

continues

LISTING 8.7 Continued

```
    if DrawType <> dtClipRect then
      Canvas.Pen.Color := pnlFgBgBorder.Color;
  end;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close; // Terminate application
end;

procedure TMainForm.mmiSaveFileClick(Sender: TObject);
{ This method saves the image to the file specified by FileName. If
  FileName is blank, however, SaveAs1Click is called to get a filename.}
begin
  if FileName = '' then
    mmiSaveAsClick(nil)
  else begin
    imgDrawingPad.Picture.SaveToFile(FileName);
    stbMain.Panels[0].Text := FileName;
    Modified := False;
  end;
end;

procedure TMainForm.mmiSaveAsClick(Sender: TObject);
{ This method launches SaveDialog to get a file name to which
  the image's contents will be saved. }
begin
  if SaveDialog.Execute then
  begin
    FileName := SaveDialog.FileName;  // Store the filename
    mmiSaveFileClick(nil)
  end;
end;

procedure TMainForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
{ If the user attempts to close the form before saving the image, they
  are prompted to do so in this method. }
var
  Rslt: Word;
begin
  CanClose := False; // Assume fail.
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?',
      mtConfirmation, mbYesNOCancel, 0);
```

```
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // no need to do anything.
      mrCancel: Exit;
    end
  end;
  CanClose := True;    // Allow use to close application
end;

procedure TMainForm.mmiNewFileClick(Sender: TObject);
{ This method erases any drawing on the main image after prompting the
  user to save it to a file in which case the mmiSaveFileClick event handler
  is called. }
var
  Rslt: Word;
begin
  if Modified then begin
    Rslt := MessageDlg('File has changed, save?', mtConfirmation,
mbYesNOCancel, 0);
    case Rslt of
      mrYes: mmiSaveFileClick(nil);
      mrNo: ;  // no need to do anything.
      mrCancel: Exit;
    end
  end;

   with imgDrawingPad.Canvas do begin
     Brush.Style := bsSolid;
     Brush.Color := clWhite;  // clWhite erases the image
     FillRect(ClipRect);      // Erase the image
     FileName := '';
     stbMain.Panels[0].Text := FileName;
   end;
   SetDrawingStyle;   // Restore the previous drawing style
   Modified := False;
end;

procedure TMainForm.mmiOpenFileClick(Sender: TObject);
{ This method opens a bitmap file specified by OpenDialog.FileName. If
  a file was already created, the user is prompted to save
  the file in which case the mmiSaveFileClick event is called. }
var
  Rslt: Word;
begin

  if OpenDialog.Execute then
```

continues

LISTING 8.7 Continued

```
  begin

    if Modified then begin
      Rslt := MessageDlg('File has changed, save?',
        mtConfirmation, mbYesNOCancel, 0);
      case Rslt of
        mrYes: mmiSaveFileClick(nil);
        mrNo: ;  // no need to do anything.
        mrCancel: Exit;
      end
    end;

    imgDrawingPad.Picture.LoadFromFile(OpenDialog.FileName);
    FileName := OpenDialog.FileName;
    stbMain.Panels[0].Text := FileName;
    Modified := false;
  end;

end;

procedure TMainForm.mmiEditClick(Sender: TObject);
{ The timer is used to determine if an area on the main image is
  surrounded by a bounding rectangle. If so, then the Copy and Cut
  menu items are enabled. Otherwise, they are disabled. }
var
  IsRect: Boolean;
begin
  IsRect := (MouseOrg.X <> NextPoint.X) and (MouseOrg.Y <> NextPoint.Y);
  if (DrawType = dtClipRect) and IsRect then
  begin
    mmiCut.Enabled := True;
    mmiCopy.Enabled := True;
  end
  else begin
    mmiCut.Enabled := False;
    mmiCopy.Enabled := False;
  end;
end;

procedure TMainForm.CopyCut(Cut: Boolean);
{ This method copies a portion of the main image to the clipboard.
  The portion copied is specified by a bounding rectangle
  on the main image. If Cut is true, the area in the bounding rectangle
  is erased. }
var
```

```
   CopyBitMap: TBitmap;
   DestRect, SrcRect: TRect;
   OldBrushColor: TColor;
begin
  CopyBitMap := TBitMap.Create;
  try
    { Set CopyBitmap's size based on the coordinates of the
      bounding rectangle }
    CopyBitMap.Width := Abs(NextPoint.X - MouseOrg.X);
    CopyBitMap.Height := Abs(NextPoint.Y - MouseOrg.Y);
    DestRect := Rect(0, 0, CopyBitMap.Width, CopyBitmap.Height);
    SrcRect := Rect(Min(MouseOrg.X, NextPoint.X)+1,
                    Min(MouseOrg.Y, NextPoint.Y)+1,
                    Max(MouseOrg.X, NextPoint.X)-1,
                    Max(MouseOrg.Y, NextPoint.Y)-1);
    { Copy the portion of the main image surrounded by the bounding
      rectangle to the Windows clipboard }
    CopyBitMap.Canvas.CopyRect(DestRect, imgDrawingPad.Canvas, SrcRect);
    { Previous versions of Delphi required the bitmap's Handle property
      to be touched for the bitmap to be made available. This was due to
      Delphi's caching of bitmapped images. The step below may not be
      required. }
    CopyBitMap.Handle;
    // Assign the image to the clipboard.
    ClipBoard.Assign(CopyBitMap);
    { If cut was specified the erase the portion of the main image
      surrounded by the bounding Rectangle }
    if Cut then
      with imgDrawingPad.Canvas do
      begin
        OldBrushColor := Brush.Color;
        Brush.Color := clWhite;
        try
          FillRect(SrcRect);
        finally
          Brush.Color := OldBrushColor;
        end;
      end;
  finally
    CopyBitMap.Free;
  end;
end;

procedure TMainForm.mmiCutClick(Sender: TObject);
begin
  CopyCut(True);
```

continues

LISTING 8.7 Continued

```
end;

procedure TMainForm.mmiCopyClick(Sender: TObject);
begin
  CopyCut(False);
end;

procedure TMainForm.mmiPasteClick(Sender: TObject);
{ This method pastes the data contained in the clipboard to the
  paste bitmap. The reason it is pasted to the PasteBitmap, an off-
image elsewhere on to the main image. This is done by having the pbPasteBox,
  a TPaintBox component, draw the contents of PasteImage. When the
  user if done positioning the pbPasteBox, the contents of TPasteBitmap
  is drawn to imgDrawingPad at the location specified by pbPasteBox's
  location.}
begin
  { Clear the bounding rectangle }

  pbPasteBox.Enabled := True;
  if DrawType = dtClipRect then
  begin
    DrawToImage(MouseOrg, NextPoint, pmNotXOR);
    EraseClipRect := False;
  end;

  PasteBitmap.Assign(ClipBoard);   // Grab the data from the clipboard
  Pasted := True;
  // Set position of pasted image to top left
  pbPasteBox.Left := 0;
  pbPasteBox.Top := 0;
  // Set the size of pbPasteBox to match the size of PasteBitmap
  pbPasteBox.Width := PasteBitmap.Width;
  pbPasteBox.Height := PasteBitmap.Height;

  pbPasteBox.Visible := True;
  pbPasteBox.Invalidate;
end;

procedure TMainForm.pbPasteBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
{ This method set's up pbPasteBox, a TPaintBox for being moved by the
  user when the left mouse button is held down }
begin
  if Button = mbLeft then
```

```
  begin
    PBoxMoving := True;
    Screen.Cursor := crMove;
    PBoxMouseOrg := Point(X, Y);
  end
  else
    PBoxMoving := False;
end;

procedure TMainForm.pbPasteBoxMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
{ This method moves pbPasteBox if the PBoxMoving flag is true indicating
  that the user is holding down the left mouse button and is dragging
  PaintBox }
begin
  if PBoxMoving then
  begin
    pbPasteBox.Left := pbPasteBox.Left + (X - PBoxMouseOrg.X);
    pbPasteBox.Top := pbPasteBox.Top + (Y - PBoxMouseOrg.Y);
  end;
end;

procedure TMainForm.pbPasteBoxMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
{ This method disables moving of pbPasteBox when the user lifts the left
  mouse button }
  if PBoxMoving then
  begin
     PBoxMoving := False;
     Screen.Cursor := crDefault;
  end;
  pbPasteBox.Refresh; // Redraw the pbPasteBox.
end;

procedure TMainForm.pbPasteBoxPaint(Sender: TObject);
{ The paintbox is drawn whenever the user selects the Paste option
  form the menu. pbPasteBox draws the contents of PasteBitmap which
  holds the image gotten from the clipboard. The reason for drawing
  PasteBitmap's contents in pbPasteBox, a TPaintBox class, is so that
  the user can also move the object around on top of the main image.
  In other words, pbPasteBox can be moved, and hidden when necessary. }
var
  DestRect, SrcRect: TRect;
begin
  // Display the paintbox only if a pasting operation occurred.
```

continues

LISTING 8.7 Continued

```
  if Pasted then
  begin
    { First paint the contents of PasteBitmap using canvas's CopyRect
      but only if the paintbox is not being moved. This reduces
      flicker }
    if not PBoxMoving then
    begin
      DestRect := Rect(0, 0, pbPasteBox.Width, pbPasteBox.Height);
      SrcRect := Rect(0, 0, PasteBitmap.Width, PasteBitmap.Height);
      pbPasteBox.Canvas.CopyRect(DestRect, PasteBitmap.Canvas, SrcRect);
    end;
    { Now copy a bounding rectangle to indicate that pbPasteBox is
      a moveable object. We use a pen mode of pmNotXOR because we
      must erase this rectangle when the user copies PaintBox's
      contents to the main image and we must preserve the original
      contents. }
    pbPasteBox.Canvas.Pen.Mode := pmNotXOR;
    pbPasteBox.Canvas.Pen.Style := psDot;
    pbPasteBox.Canvas.Brush.Style := bsClear;
    pbPasteBox.Canvas.Rectangle(0, 0, pbPasteBox.Width,
      pbPasteBox.Height);
  end;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Remove the form from the clipboard chain
  ChangeClipBoardChain(Handle, OldClipViewHwnd);
  PasteBitmap.Free; // Free the PasteBitmap instance
end;

procedure TMainForm.RgGrpFillOptionsClick(Sender: TObject);
begin
  FillSelected   := RgGrpFillOptions.ItemIndex = 0;
  BorderSelected := cbxBorder.Checked;
  SetDrawingStyle;
end;

end.
```

How the Paint Program Works

The paint program is actually quite a bit of code. Because it would be difficult to explain how it works outside of the code, we've added ample comments to the source. We'll describe the general functionality of the paint program here. The main form is shown in Figure 8.18.

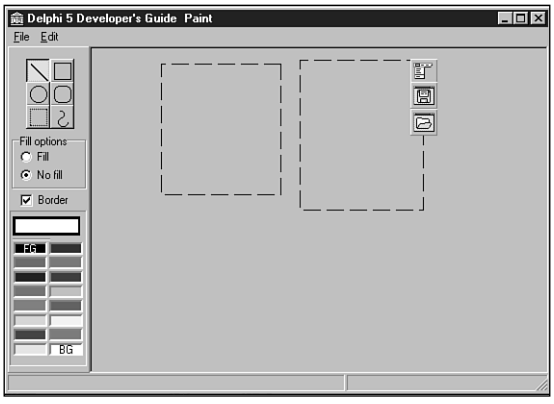

FIGURE 8.18

The main form for the paint program.

NOTE

Notice that a `TImage` component is used as a drawing surface for the paint program. Keep in mind that this can only be the case if the image uses a `TBitmap` object.

The main form contains a main image component, `imgDrawingPad`, which is placed on a `TScrollBox` component. `imgDrawingPad` is where the user performs drawing. The selected speed button on the form's toolbar specifies the type of drawing that the user performs.

The user can draw lines, rectangles, ellipses, and rounded rectangles as well as perform free-form drawing. Additionally, a portion of the main image can be selected and copied to the Windows Clipboard so that it can be pasted into another application that can handle bitmap data. Likewise, the paint program can accept bitmap data from the Windows Clipboard.

TPanel Techniques

The fill style and border type are specified by the Fill Options radio group. The fill and border colors are set using the color grid in the `ColorPanel` shown in Figure 8.18.

Clipboard Pasting of Bitmap Data

To paste data from the Clipboard, you use an offscreen bitmap, `PasteBitMap`, to hold the pasted data. A `TPaintBox` component, `pbPasteBox`, then draws the data from `PasteBitMap`. The reason for using a `TPaintBox` component for drawing the contents of `PasteBitMap` is so the user can move `pbPasteBox` to any location on the main image to designate where the pasted data is to be copied to the main image.

Attaching to the Win32 Clipboard Viewer Chain

Another technique shown by the paint program is how an application can attach itself to the Win32 Clipboard viewer chain. This is done in the `FormCreate()` method by the call to the Win32 API function `SetClipboardViewer()`. This function takes the handle of the window attaching itself to the chain and returns the handle to the next window in the chain. The return value must be stored so that when the application shuts down, it can restore the previous state of the chain using `ChangeClipboardChain()`, which takes the handle being removed and the saved handle. The paint program restores the chain in the main form's `FormDestroy()` method. When an application is attached to the Clipboard viewer chain, it receives the `WM_DRAWCLIPBOARD` messages whenever the data on the Clipboard is modified. You take advantage of this by capturing this message and enabling the Paste menu item if the changed data in the Clipboard is bitmap data. This is done in the `WMDrawClipBoard()` method.

Bitmap Copying

Bitmap copy operations are performed in the `CopyCut()`, `pbPasteBoxPaint()`, and `CopyPasteBoxToImage()` methods. The `CopyCut()` method copies a portion of the main image selected by a bounding rectangle to the Clipboard and then erases the bounded area if the `Cut` parameter passed to it is `True`. Otherwise, it leaves the area intact.

`PbPasteBoxPaint()` copies the contents of the offscreen bitmap to `pbPasteBox.Canvas` but only when `pbPasteBox` is not being moved. This helps reduce flicker as the user moves `pbPasteBox`.

`CopyPasteBoxToImage()` copies the contents of the offscreen bitmap to the main image `imgDrawingPad` at the location specified by `pbPasteBox`.

Paint Program Comments

As mentioned earlier, much of the functionality of the paint program is documented in the code's commentary. It would be a good idea to read through the source and comments and step through the code so that you can gain a good understanding of what's happening in the program.

Performing Animation with Graphics Programming

This section demonstrates how you can achieve simple sprite animation by mixing Delphi 5 classes with Win32 GDI functions. The animation project resides on the CD as `Animate.dpr`. Listing 8.8 shows the main form, which contains the main form functionality.

LISTING 8.8 The Animation Project's Main Form

```
unit MainFrm;

interface
```

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, Stdctrls, AppEvnts;

{$R SPRITES.RES } // Link in the bitmaps

type

  TSprite = class
  private
    FWidth: integer;
    FHeight: integer;
    FLeft: integer;
    FTop: integer;
    FAndImage, FOrImage: TBitMap;
  public
    property Top: Integer read FTop write FTop;
    property Left: Integer read FLeft write FLeft;
    property Width: Integer read FWidth write FWidth;
    property Height: Integer read FHeight write FHeight;
    constructor Create;
    destructor Destroy; override;
  end;

  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiSlower: TMenuItem;
    mmiFaster: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    appevMain: TApplicationEvents;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiSlowerClick(Sender: TObject);
    procedure mmiFasterClick(Sender: TObject);
    procedure appevMainIdle(Sender: TObject; var Done: Boolean);
  private
    BackGnd1, BackGnd2: TBitMap;
    Sprite: TSprite;
    GoLeft,GoRight,GoUp,GoDown: boolean;
    FSpeed, FSpeedIndicator: Integer;
    procedure DrawSprite;
  end;
```

continues

LISTING 8.8 Continued

```
const

  BackGround = 'BACK2.BMP';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

constructor TSprite.Create;
begin
  inherited Create;
  { Create the bitmaps to hold the sprite images that will
    be used for performing the AND/OR operation to create animation }
  FAndImage := TBitMap.Create;
  FAndImage.LoadFromResourceName(hInstance, 'AND');

  FOrImage := TBitMap.Create;
  FOrImage.LoadFromResourceName(hInstance, 'OR');

  Left := 0;
  Top := 0;
  Height := FAndImage.Height;
  Width := FAndImage.Width;

end;

destructor TSprite.Destroy;
begin
  FAndImage.Free;
  FOrImage.Free;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Create the original background image
  BackGnd1 := TBitMap.Create;
  with BackGnd1 do
  begin
    LoadFromResourceName(hInstance, 'BACK');
    Parent := nil;
    SetBounds(0, 0,  Width, Height);
  end;
```

```
  // Create a copy of the background image
  BackGnd2 := TBitMap.Create;
  BackGnd2.Assign(BackGnd1);


  // Create a sprite image
  Sprite := TSprite.Create;

  // Initialize the direction variables
  GoRight := true;
  GoDown := true;
  GoLeft := false;
  GoUp := false;


  FSpeed := 0;
  FSpeedIndicator := 0;

  { Set the application's OnIdle event to MyIdleEvent which will start
    the sprite moving }

  // Adjust the form's client width/height
  ClientWidth := BackGnd1.Width;
  ClientHeight := BackGnd1.Height;

end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  // Free all objects created in the form's create constructor
  BackGnd1.Free;
  BackGnd2.Free;
  Sprite.Free;
end;

procedure TMainForm.DrawSprite;
var
  OldBounds: TRect;
begin

  // Save the sprite's bounds in OldBounds
  with OldBounds do
  begin
    Left := Sprite.Left;
    Top := Sprite.Top;
    Right := Sprite.Width;
```

continues

LISTING 8.8 Continued

```
  Bottom := Sprite.Height;
end;

{ Now change the sprites bounds so that it moves in one direction
  or changes direction when it comes in contact with the form's
  boundaries }
with Sprite do
begin
  if GoLeft then
    if Left > 0 then
       Left := Left - 1
    else begin
       GoLeft := false;
       GoRight := true;
    end;

  if GoDown then
    if (Top + Height) < self.ClientHeight then
       Top := Top + 1
    else begin
      GoDown := false;
      GoUp := true;
    end;

  if GoUp then
    if  Top > 0 then
       Top := Top - 1
    else begin
      GoUp := false;
      GoDown := true;
    end;

  if GoRight then
  if (Left + Width) < self.ClientWidth then
    Left := Left + 1
  else begin
    GoRight := false;
    GoLeft := true;
  end;
end;

{ Erase the original drawing of the sprite on BackGnd2 by copying
  a rectangle from BackGnd1 }
with OldBounds do
  BitBlt(BackGnd2.Canvas.Handle, Left, Top, Right, Bottom,
         BackGnd1.Canvas.Handle, Left, Top, SrcCopy);
```

```
  { Now draw the sprite onto the off-screen bitmap. By performing the
   drawing in an off-screen bitmap, the flicker is eliminated. }
  with Sprite do
  begin
    { Now create a black hole where the sprite first existed by And-ing
      the FAndImage onto BackGnd2 }
    BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
           FAndImage.Canvas.Handle, 0, 0, SrcAnd);
    // Now fill in the black hole with the sprites original colors
    BitBlt(BackGnd2.Canvas.Handle, Left, Top, Width, Height,
           FOrImage.Canvas.Handle, 0, 0, SrcPaint);
  end;

  { Copy the sprite at its new location to the form's Canvas. A
    rectangle slightly larger than the sprite is needed
    to effectively erase the sprite by over-writing it, and draw the
    new sprite at the new location with a single BitBlt call }
  with OldBounds do
     BitBlt(Canvas.Handle, Left - 2, Top - 2, Right + 2, Bottom + 2,
            BackGnd2.Canvas.Handle, Left - 2, Top - 2, SrcCopy);

end;

procedure TMainForm.FormPaint(Sender: TObject);
begin
  // Draw the background image whenever the form gets painted
  BitBlt(Canvas.Handle, 0, 0, ClientWidth, ClientHeight,
         BackGnd1.Canvas.Handle, 0, 0, SrcCopy);
end;

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

procedure TMainForm.mmiSlowerClick(Sender: TObject);
begin
  Inc(FSpeedIndicator, 100);
end;

procedure TMainForm.mmiFasterClick(Sender: TObject);
begin
  if FSpeedIndicator >= 100 then
    Dec(FSpeedIndicator, 100)
  else
    FSpeedIndicator := 0;
```

continues

LISTING 8.8 Continued

```
end;

procedure TMainForm.appevMainIdle(Sender: TObject; var Done: Boolean);
begin
  if FSpeed >= FSpeedIndicator then
  begin
    DrawSprite;
    FSpeed := 0;
  end
  else
    inc(FSpeed);

Done := False;
end;

end.
```

The animation project consists of a background image on which a sprite, a flying saucer, is drawn and moved about the background's client area. The background is represented by a bitmap consisting of scattered stars (see Figure 8.19).

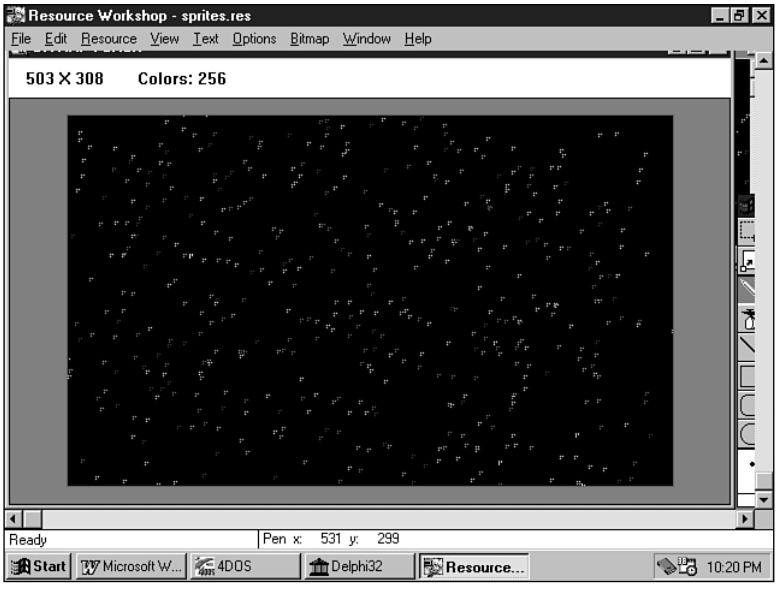

FIGURE 8.19

The background of the animation project.

The sprite is made up of two 64×32 bitmaps. More on these bitmaps later; for now, we'll discuss what goes on in the source code.

The unit defines a `TSprite` class. `TSprite` contains the fields that hold the sprite's location on the background image and two `TBitmap` objects to hold each of the sprite bitmaps. The `TSprite.Create` constructor creates both `TBitmap` instances and loads them with the actual bitmaps. Both the sprite bitmaps and the background bitmap are kept in a resource file that you link to the project by including the following statement in the main unit:

```
{$R SPRITES.RES }
```

After the bitmap is loaded, the sprite's boundaries are set. The `TSprite.Done` destructor frees both bitmap instances.

The main form contains two `TBitmap` objects, a `TSprite` object, and direction indicators to specify the direction of the sprite's motion. Additionally, the main form defines another method, `DrawSprite()`, which has the sprite-drawing functionality. The `TApplicationEvents` component is a new Delphi 5 component that allows you to hook into the `Application`-level events. Prior to Delphi 5, you had to do this by adding `Application`-level events at runtime. Now, with this component, you can do all event management for `TApplication` at design time. We will use this component to provide an OnIdle event for the `TApplication` object.

Note that two private variables are used to control the speed of the animation: `FSpeed` and `FSpeedIndicator`. These are used in the `DrawSprite()` method for slowing down the animation on faster machines.

The `FormCreate()` event handler creates both `TBitmap` instances and loads each with the same background bitmap. (The reason you use two bitmaps will be discussed in a moment.) It then creates a `TSprite` instance, and sets the direction indicators. Finally `FormCreate()` resizes the form to the background image's size.

The `FormPaint()` method paints the `BackGnd1` to its canvas, and the `FormDestroy()` frees the `TBitmap` and `TSprite` instances.

The `appevMainIdle()` method calls `DrawSprite()`, which moves and draws the sprite on the background. The bulk of the work is done in the `DrawSprite()` method.`appevMainIdle()` will be invoked whenever the application is in an idle state. That is, whenever there are no actions from the user for the application to respond to.

The `DrawSprite()` method repositions the sprite on the background image. A series of steps is required to erase the old sprite on the background and then draw it at its new location while preserving the background colors around the actual sprite image. Additionally, `DrawSprite()` must perform these steps without producing flickering while the sprite is moving.

To accomplish this, drawing is performed on the offscreen bitmap, `BackGnd2`. `BackGnd2` and `BackGnd1` are exact copies of the background image. However, `BackGnd1` is never modified, so it's a clean copy of the background. When drawing is complete on `BackGnd2`, the modified area of `BackGnd2` is copied to the form's canvas. This allows for only one `BitBlt()` operation to the

form's canvas to both erase and draw the sprite at its new location. The drawing operations performed on BackGnd2 are as follows.

First, a rectangular region is copied from BackGnd1 to BackGnd2 over the area occupied by the sprite. This effectively erases the sprite from BackGnd2. Then the FAndImage bitmap is copied to BackGnd2 at its new location using the bitwise AND operation. This effectively creates a black hole in BackGnd2 where the sprite exists and still preserves the colors on BackGnd2 surrounding the sprite. Figure 8.20 shows FAndImage.

In Figure 8.20, the sprite is represented by black pixels, and the rest of the image surrounding it consists of white pixels. The color black has a value of 0, and the color white has the value of 1. Tables 8.5, and 8.6 show the results of performing the AND operation with black and white colors.

TABLE 8.5 AND Operation with Black Color

Background	*Value*	*Color*
BackGnd2	1001	Some color
FAndImage	0000	Black
Result	0000	Black

TABLE 8.6 AND Operation with White Color<$AND operation; with white color>

Background	*Value*	*Color*
BackGnd2	1001	Some color
FAndImage	1111	White
Result	1001	Some color

These tables show how performing the AND operation results in blacking out the area where the sprite exists on BackGnd2. In Table 8.5, Value represents a pixel color. If a pixel on BackGnd2 contains some arbitrary color, combining this color with the color black using the AND operator results in that pixel becoming black. Combining this color with the color white using the AND operator results in the color being the same as the arbitrary color, as shown in Table 8.6. Because the color surrounding the sprite in FAndImage is white, the pixels on BackGnd2 where this portion of FAndImage is copied retain their colors.

After copying FAndImage to BackGnd2, FOrImage must be copied to the same location on BackGnd2 to fill in the black hole created by FAndImage with the actual sprite colors. FOrImage also has a rectangle surrounding the actual sprite image. Again, you're faced with getting the sprite colors to BackGnd2 while preserving BackGnd2's colors surrounding the sprite. This is accomplished by combining FOrImage with BackGnd2 using the OR operation. FOrImage is shown in Figure 8.21.

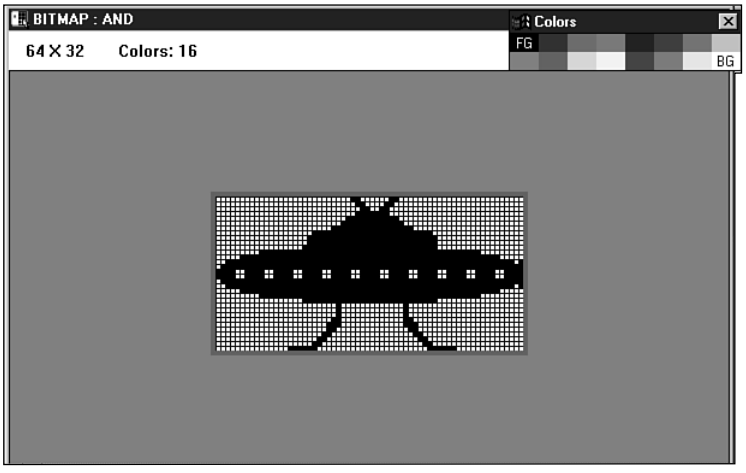

FIGURE 8.20

`FAndImage` *for a sprite.*

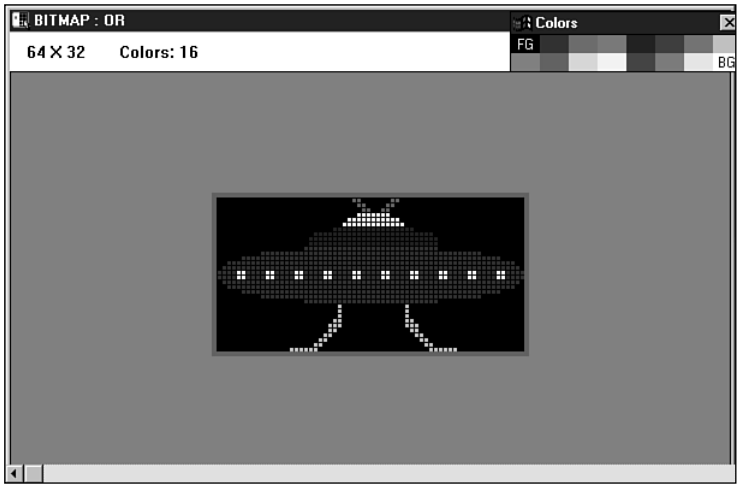

FIGURE 8.21

`FOrImage`.

Notice that the area surrounding the sprite image is black. Table 8.7 shows the results of performing the `OR` operation on `FOrImage` and `BackGnd2`.

TABLE 8.7 An OR Operation with the Color Black

Background	*Value*	*Color*
BackGnd2	1001	Some color
FAndImage	0000	Black
Result	1001	Black

Table 8.7 shows that if BackGnd2 contains an arbitrary color, using the OR operation to combine it with black will result in BackGnd2's color.

Recall that all this drawing is performed on the offscreen bitmap. When the drawing is complete, a single BitBlt() is made to the form's canvas to erase and copy the sprite.

The technique shown here is a fairly common method for performing animation. You might consider extending the functionality of the TSprite class to move and draw itself on a parent canvas.

Advanced Fonts

Although the VCL enables you to manipulate fonts with relative ease, it doesn't provide the vast font-rendering capabilities provided by the Win32 API. This section gives you a background on Win32 fonts and shows you how to manipulate them.

Types of Win32 Fonts

There are basically two types of fonts in Win32: GDI fonts and device fonts. GDI fonts are stored in font resource files and have an extension of .fon (for raster and vector fonts) or .tot and .ttf (for TrueType fonts). Device fonts are specific to a particular device, such as a printer. Unlike with the GDI fonts, when Win32 uses a device font for printing text, it only needs to send the ASCII character to the device, and the device takes care of printing the character in the specified font. Otherwise, Win32 converts the font to a bitmap or performs the GDI function to draw the font. Drawing the font using bitmaps or GDI functions generally takes longer, as is the case with GDI fonts. Although device fonts are faster, they are device-specific and often very limiting in what fonts a particular device supports.

Basic Font Elements

Before you learn how to use the various fonts in Win32, you should know the various terms and elements associated with Win32 fonts.

A Font's Typeface, Family, and Measurements

Think of a font as just a picture or *glyph* representing a character. Each character has two characteristics: a typeface and a size.

In Win32, a font's *typeface* refers to the font's style and its size. Probably the best definition of typeface and how it relates to a font is in the Win32 help file. This definition says, "A typeface

is a collection of characters that share design characteristics; for example, Courier is a common typeface. A font is a collection of characters that have the same typeface and size."

Win32 categorizes these different typefaces into five font families: Decorative, Modern, Roman, Script, and Swiss. The distinguishing font features in these families are the font's serifs and stroke widths.

A *serif* is a small line at the beginning or end of a font's main strokes that give the font a finished appearance. A *stroke* is the primary line that makes up the font. Figure 8.22 illustrates these two features.

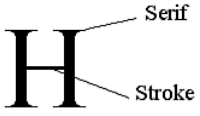

FIGURE 8.22

Serifs and strokes.

Some of the typical fonts you'll find in the different font families are listed in Table 8.8.

TABLE 8.8 Font Families and Typical Fonts

Font Family	*Typical Fonts*
Decorative	Novelty fonts: Old English
Modern	Fonts with constant strike widths that may or may not have serifs: Pica, Elite, and Courier New
Roman	Fonts with variable stroke widths and serifs: Times New Roman and New Century SchoolBook
Script	Fonts that look like handwriting: Script and Cursive
Swiss	Fonts with variable stroke widths without serifs: Arial and Helvetica

A font's size is represented in points (a *point* is 1/72 of an inch). A font's height consists of its *ascender* and *descender*. The ascender and descender are represented by the `tmAscent` and `tmDescent` values as shown in Figure 8.23. Figure 8.23 shows other values essential to the character measurement as well.

Characters reside in a character *cell*, an area surrounding the character that consists of white space. When referring to character measurements, keep in mind that the measurement may include both the character glyph (the character's visible portion) and the character cell. Others may refer to only one or the other.

Table 8.9 explains the meaning of the various character measurements.

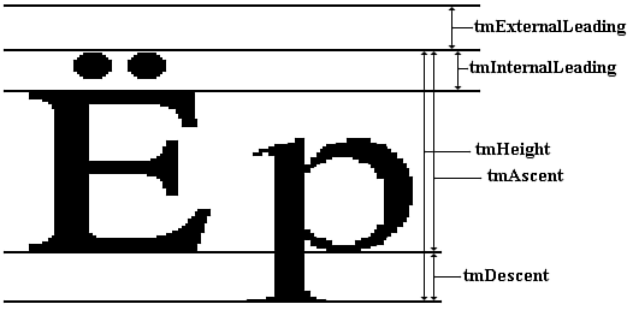

FIGURE 8.23

Character measurement values.

TABLE 8.9 Character Measurements

Measurement	*Meaning*
External leading	The space between text lines
Internal leading	The difference between the character's glyph height and the font's cell height
Ascent	Measurement from the baseline to the top of the character cell
Descent	Measurement from the baseline to the bottom of the character cell
Point size	The character height minus `tmInternalLeading`
Height	The sum of ascent, descent, and internal leading
Baseline	The line on which characters sit

GDI Font Categories

There are essentially three separate categories of GDI fonts: raster fonts, vector fonts (also referred to as *stroke fonts*), and TrueType fonts. The first two existed in older versions of Win32, whereas the latter was introduced in Windows 3.1.

Raster Fonts Explained

Raster fonts are basically bitmaps provided for a specific resolution or *aspect ratio* (ratio of the pixel height and width of a given device) and font size. Because these fonts are provided in specific sizes, Win32 can synthesize the font to generate a new font in the requested size, but it can do so only to produce a larger font from a smaller font. The reverse is not possible because the technique Win32 uses to synthesize the fonts is to duplicate the rows and columns that make up the original font bitmap. Raster fonts are convenient when the size requested is available. They're fast to display and look good when used at the intended size. The disadvantage is that they tend to look a bit sloppy when scaled to larger sizes, as shown in Figure 8.24, which displays the Win32 System font.

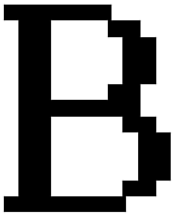

FIGURE 8.24

A raster font.

Vector Fonts Explained

Vector fonts are generated by Win32 with a series of lines created by GDI functions as opposed to bitmaps. These fonts offer better scalability than do raster fonts, but they have a much lower density when displayed, which may or may not be desired. Also, the performance of vector fonts is slow compared to raster fonts. Vector fonts lend themselves best to use with plotters but aren't recommended for designing appealing user interfaces. Figure 8.25 shows a typical vector font.

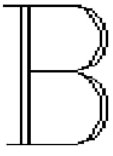

FIGURE 8.25

A vector font.

TrueType Fonts Explained

TrueType fonts are probably the most preferred of the three font types. The advantage to using TrueType fonts is that they can represent virtually any style of font in any size and look pleasing to the eye. Win32 displays TrueType fonts by using a collection of points and hints that describe the font outline. *Hints* are simply algorithms to distort a scaled font's outline to improve its appearance at different resolutions. Figure 8.26 shows a TrueType font.

FIGURE 8.26

A TrueType font.

Displaying Different Font Types

So far, we've given you the general concepts surrounding Window's font technology. If you're interested in getting down to the many nuts and bolts of fonts, take a look at the Win32 online help file on "Fonts Overview," which provides you with a vast amount of information on the topic. Now, you'll learn how to use the Win32 API and Win32 structures to create and display fonts of any shape and size.

A Font-Creation Sample Project

The example to follow illustrates the process of instantiating different font types in Windows. The project also illustrates how to obtain information about a rendered font. This project is located in on the CD as `MakeFont.dpr`.

How the Project Works

Through the main form, you select various font attributes to be used in creating the font. The font then gets drawn to a `TPaintBox` component whenever you change the value of one of the font's attributes. (All components are attached to the `FontChanged()` event handler through their `OnChange` or `OnClick` events.) You also can view information about a font by clicking the Font Information button. Figure 8.27 shows the main form for this project. Listing 8.9 shows the unit defining the main form.

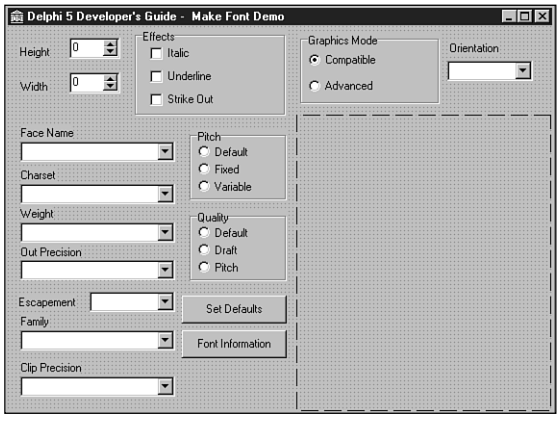

FIGURE 8.27

The main form for the font-creation project.

LISTING 8.9 The Font-Creation Project

```
unit MainFrm;

interface
```

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask, Spin;

const

// Array to represent the TLOGFONT.lfCharSet values
CharSetArray: array[0..4] of byte = (ANSI_CHARSET, DEFAULT_CHARSET,
  SYMBOL_CHARSET, SHIFTJIS_CHARSET, OEM_CHARSET);

// Array to represent the TLOGFONT.lfWeight values
WeightArray: array[0..9] of integer =
 (FW_DONTCARE, FW_THIN, FW_EXTRALIGHT, FW_LIGHT, FW_NORMAL, FW_MEDIUM,
  FW_SEMIBOLD, FW_BOLD, FW_EXTRABOLD, FW_HEAVY);

// Array to represent the TLOGFONT.lfOutPrecision values
OutPrecArray: array[0..7] of byte = (OUT_DEFAULT_PRECIS,
  OUT_STRING_PRECIS, OUT_CHARACTER_PRECIS, OUT_STROKE_PRECIS,
  OUT_TT_PRECIS, OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS,
  OUT_TT_ONLY_PRECIS);

// Array to represent the TLOGFONT.lfPitchAndFamily higher four-bit
// values
FamilyArray: array[0..5] of byte = (FF_DONTCARE, FF_ROMAN,
  FF_SWISS, FF_MODERN, FF_SCRIPT, FF_DECORATIVE);

// Array to represent the TLOGFONT.lfPitchAndFamily lower two-bit values
PitchArray: array[0..2] of byte = (DEFAULT_PITCH, FIXED_PITCH,
   VARIABLE_PITCH);

// Array to represent the TLOGFONT.lfClipPrecision values
ClipPrecArray: array[0..6] of byte = (CLIP_DEFAULT_PRECIS,
  CLIP_CHARACTER_PRECIS, CLIP_STROKE_PRECIS, CLIP_MASK, CLIP_LH_ANGLES,
  CLIP_TT_ALWAYS, CLIP_EMBEDDED);

// Array to represent the TLOGFONT.lfQuality values
QualityArray: array[0..2] of byte = (DEFAULT_QUALITY, DRAFT_QUALITY,
  PROOF_QUALITY);

type

  TMainForm = class(TForm)
    lblHeight: TLabel;
    lblWidth: TLabel;
    gbEffects: TGroupBox;
    cbxItalic: TCheckBox;
```

continues

LISTING 8.9 Continued

```
    cbxUnderline: TCheckBox;
    cbxStrikeOut: TCheckBox;
    cbWeight: TComboBox;
    lblWeight: TLabel;
    lblEscapement: TLabel;
    cbEscapement: TComboBox;
    pbxFont: TPaintBox;
    cbCharSet: TComboBox;
    lblCharSet: TLabel;
    cbOutPrec: TComboBox;
    lblOutPrecision: TLabel;
    cbFontFace: TComboBox;
    rgPitch: TRadioGroup;
    cbFamily: TComboBox;
    lblFamily: TLabel;
    lblClipPrecision: TLabel;
    cbClipPrec: TComboBox;
    rgQuality: TRadioGroup;
    btnSetDefaults: TButton;
    btnFontInfo: TButton;
    lblFaceName: TLabel;
    rgGraphicsMode: TRadioGroup;
    lblOrientation: TLabel;
    cbOrientation: TComboBox;
    seHeight: TSpinEdit;
    seWidth: TSpinEdit;
    procedure pbxFontPaint(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure btnFontInfoClick(Sender: TObject);
    procedure btnSetDefaultsClick(Sender: TObject);
    procedure rgGraphicsModeClick(Sender: TObject);
    procedure cbEscapementChange(Sender: TObject);
    procedure FontChanged(Sender: TObject);
  private
    { Private declarations }
    FLogFont: TLogFont;
    FHFont:   HFont;
    procedure MakeFont;
    procedure SetDefaults;
  public
    { Public declarations }
  end;

var
```

```
  MainForm: TMainForm;

implementation
uses FontInfoFrm;

{$R *.DFM}

procedure TMainForm.MakeFont;
begin
  // Clear the contents of FLogFont
  FillChar(FLogFont, sizeof(TLogFont), 0);
  // Set the TLOGFONT's fields
  with FLogFont do
  begin
    lfHeight         := StrToInt(seHeight.Text);
    lfWidth          := StrToInt(seWidth.Text);
    lfEscapement     :=
      StrToInt(cbEscapement.Items[cbEscapement.ItemIndex]);
    lfOrientation    :=
      StrToInt(cbOrientation.Items[cbOrientation.ItemIndex]);
    lfWeight         := WeightArray[cbWeight.ItemIndex];
    lfItalic         := ord(cbxItalic.Checked);
    lfUnderline      := ord(cbxUnderLine.Checked);
    lfStrikeOut      := ord(cbxStrikeOut.Checked);
    lfCharSet        := CharSetArray[cbCharset.ItemIndex];
    lfOutPrecision   := OutPrecArray[cbOutPrec.ItemIndex];
    lfClipPrecision  := ClipPrecArray[cbClipPrec.ItemIndex];
    lfQuality        := QualityArray[rgQuality.ItemIndex];
    lfPitchAndFamily := PitchArray[rgPitch.ItemIndex] or
       FamilyArray[cbFamily.ItemIndex];
    StrPCopy(lfFaceName, cbFontFace.Items[cbFontFace.ItemIndex]);
  end;
  // Retrieve the requested font
  FHFont := CreateFontIndirect(FLogFont);
  // Assign to the Font.Handle
  pbxFont.Font.Handle := FHFont;
  pbxFont.Refresh;
end;

procedure TMainForm.SetDefaults;
begin
  // Set the various controls to default values for ALogFont
  seHeight.Text           := '0';
  seWidth.Text            := '0';
  cbxItalic.Checked       := false;
  cbxStrikeOut.Checked    := false;
```

continues

LISTING 8.9 Continued

```
  cbxUnderline.Checked    := false;
  cbWeight.ItemIndex      := 0;
  cbEscapement.ItemIndex  := 0;
  cbOrientation.ItemIndex := 0;
  cbCharset.ItemIndex     := 1;
  cbOutPrec.Itemindex     := 0;
  cbFamily.ItemIndex      := 0;
  cbClipPrec.ItemIndex    := 0;
  rgPitch.ItemIndex       := 0;
  rgQuality.ItemIndex     := 0;
  // Fill CBFontFace TComboBox with the screen's fonts
  cbFontFace.Items.Assign(Screen.Fonts);
  cbFontFace.ItemIndex := cbFontFace.Items.IndexOf(Font.Name);
end;

procedure TMainForm.pbxFontPaint(Sender: TObject);
begin
  with pbxFont do
  begin
    { Note that in Windows 95, the graphics mode will always
      be GM_COMPATIBLE as GM_ADVANCED is recognized only by Windows NT.}
    case rgGraphicsMode.ItemIndex of
      0: SetGraphicsMode(pbxFont.Canvas.Handle, GM_COMPATIBLE);
      1: SetGraphicsMode(pbxFont.Canvas.Handle, GM_ADVANCED);
    end;
    Canvas.Rectangle(2, 2, Width-2, Height-2);
    // Write the fonts name
    Canvas.TextOut(Width div 2, Height div 2, CBFontFace.Text);
  end;
end;

procedure TMainForm.FormActivate(Sender: TObject);
begin
  SetDefaults;
  MakeFont;
end;

procedure TMainForm.btnFontInfoClick(Sender: TObject);
begin
  FontInfoForm.ShowModal;
end;

procedure TMainForm.btnSetDefaultsClick(Sender: TObject);
begin
  SetDefaults;
```

```
  MakeFont;
end;

procedure TMainForm.rgGraphicsModeClick(Sender: TObject);
begin
  cbOrientation.Enabled := rgGraphicsMode.ItemIndex = 1;
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
  MakeFont;
end;

procedure TMainForm.cbEscapementChange(Sender: TObject);
begin
  if not cbOrientation.Enabled then
    cbOrientation.ItemIndex := cbEscapement.ItemIndex;
end;

procedure TMainForm.FontChanged(Sender: TObject);
begin
  MakeFont;
end;

end.
```


In MAINFORM.PAS, you'll see several array definitions that will be explained shortly. For now, notice that the form has two private variables: FLogFont and FHFont. FLogFont is of type TLOGFONT, a record structure used to describe the font to create. FHFont is the handle to the font that gets created. The private method MakeFont() is where you create the font by first filling the FLogFont structure with values specified from the main form's components and then passing that structure to CreateFontIndirect(), a Win32 GDI function that returns a font handle to the new font. Before you go on, however, you need to understand the TLOGFONT structure.

The TLOGFONT Structure

As stated earlier, you use the TLOGFONT structure to define the font you want to create. This structure is defined in the WINDOWS unit as follows:

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
```

```
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..lf_FaceSize - 1] of Char;
end;
```

You place values in the `TLOGFONT`'s fields that specify the attributes you want your font to have. Each field represents a different type of attribute. By default, most of the fields can be set to zero, which is what the Set Defaults button on the main form does. In this instance, Win32 chooses the attributes for the font and returns whatever it pleases. The general rule is this: The more fields you fill in, the more you can fine-tune your font style. The following list explains what each `TLOGFONT` field represents. Some of the fields may be assigned constant values that are predefined in the `WINDOWS` unit. Refer to Win32 help for a detailed description of these values; we show you only the most commonly used ones here:

Field Value	*Description*
`lfHeight`	The font height. A value greater than zero indicates a cell height. A value less than zero indicates the glyph height (the cell height minus the internal leading). Set this field to zero to let Win32 decide a height for you.
`lfWidth`	The average font width. Set this field to zero to let Win32 choose a font width for you.
`lfEscapement`	The angle (in tenths of degrees) of rotation of the font's baseline (the line along which characters are drawn). In Windows 95/98, the text string and individual characters are drawn using the same angle. That is, `lfEscapement` and `lfOrientation` are the same. In Windows NT, text is drawn independently of the orientation angle of each character in the text string. To achieve the latter, you must set the graphics mode for the device to `GM_ADVANCED` using the `SetGraphicsMode()` Win32 GDI function. By default, the graphics mode is `GM_COMPATIBLE`, which makes the Windows NT behavior like Windows 95. This font-rotation effect is only available with TrueType fonts.
`lfOrientation`	Enables you to specify an angle at which to draw individual characters. In Windows 95/98, this has the same value as `lfEscapement`. In Windows NT, the values may be different. (See `lfEscapement`.)

`lfWeight`	This affects the font density. The `WINDOWS` unit defines several constants for this field, such as `FW_BOLD` and `FW_NORMAL`. Set this field to `FW_DONTCARE` to let Win32 choose a weight for you.
`lfItalic`	Nonzero means *italic*; zero means *nonitalic*.
`lfUnderline`	Nonzero means *underlined*; zero means *not underlined*.
`lfStrikeOut`	Nonzero means that a line gets drawn through the font, whereas a value of zero does not draw a line through the font.
`lfCharSet`	Win32 defines the character sets: `ANSI_CHARSET=0`, `DEFAULT_CHARSET=1`, `SYMBOL_CHARSET=2`, `SHIFTJIS_CHARSET=128`, and `OEM_CHARSET = 255`. Use the `DEFAULT_CHARSET` by default.
`lfOutPrecision`	Specifies how Win32 should match the requested font's size and characteristics to an actual font. Use `TT_ONLY_PRECIS` to specify only TrueType fonts. Other types are defined in the `WINDOWS` unit.
`lfClipPrecision`	Specifies how Win32 clips characters outside a clipping region. Use `CLIP_DEFAULT_PRECIS` to let Win32 choose.
`lfQuality`	Defines the font's output quality as GDI will draw it. Use `DEFAULT_QUALITY` to let Win32 decide, or you may specify `PROOF_QUALITY` or `DRAFT_QUALITY`.
`lfPitchAndFamily`	Defines the font's pitch in the two low-order bits. Specifies the family in the higher four high-order bits. Table 8.8 displays these families.
`lfFaceName`	The typeface name of the font.

The `MakeFont()` procedure uses the values defined in the `constant` section of `MainForm.pas`. These array constants contain the various predefined constant values for the `TLOGFONT` structure. These values are placed in the same order as the choices in the main form's `TComboBox` components. For example, the choices for the font family in the `CBFamily` combo box are in the same order as the values in `FamilyArray`. We used this technique to reduce the code required to fill in the `TLOGFONT` structure. The first line in the `MakeFont()` function

```
fillChar(FLogFont, sizeof(TLogFont), 0);
```

clears the `FLogFont` structure before any values are set. When `FLogFont` has been set, the line

```
FHFont := CreateFontIndirect(FLogFont);
```

calls the Win32 API function `CreateFontIndirect()`, which accepts the `TLOGFONT` structure as a parameter and returns a handle to the requested font. This handle is then set to the `TPaintBox.Font`'s handle property. Delphi 5 takes care of destroying the `TPaintBox`'s previous font before making the assignment. After the assignment is made, you redraw `pbxFont` by calling its `Refresh()` method.

The `SetDefaults()` method initializes the `TLOGFONT` structure with default values. This method is called when the main form is created and whenever the user clicks the Set Defaults button. Experiment with the project to see the different effects you can get with fonts, as shown in Figure 8.28.

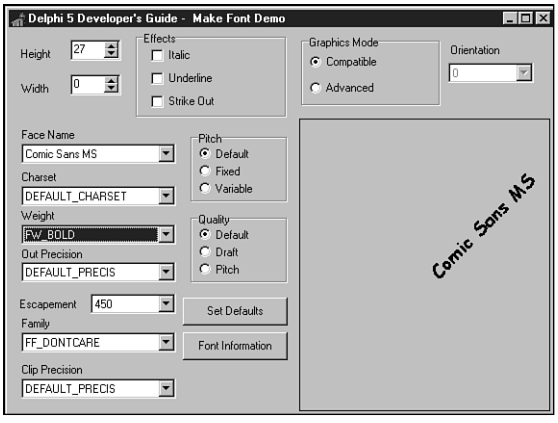

FIGURE 8.28

A rotated font.

Displaying Information About Fonts

The main form's Font Information button invokes the form `FontInfoForm`, which displays information about the selected font. When you specify font attributes in the `TLOGFONT` structure, Win32 attempts to provide you with a font that best resembles your requested font. It's entirely possible that the font you get back from the `CreateFontIndirect()` function has completely different attributes than what you requested. `FontInfoForm` lets you inspect your selected font's attributes. It uses the Win32 API function `GetTextMetrics()` to retrieve the font information.

`GetTextMetrics()` takes two parameters: the handle to the device context whose font you want to examine and a reference to another Win32 structure, `TTEXTMETRIC`. `GetTextMetrics()` then updates the `TTEXTMETRIC` structure with information about the given font. The `WINDOWS` unit defines the `TTEXTMETRIC` record as follows:

```
TTextMetric = record
   tmHeight: Integer;
   tmAscent: Integer;
   tmDescent: Integer;
   tmInternalLeading: Integer;
   tmExternalLeading: Integer;
   tmAveCharWidth: Integer;
```

```
    tmMaxCharWidth: Integer;
    tmWeight: Integer;
    tmItalic: Byte;
    tmUnderlined: Byte;
    tmStruckOut: Byte;
    tmFirstChar: Byte;
    tmLastChar: Byte;
    tmDefaultChar: Byte;
    tmBreakChar: Byte;
    tmPitchAndFamily: Byte;
    tmCharSet: Byte;
    tmOverhang: Integer;
    tmDigitizedAspectX: Integer;
    tmDigitizedAspectY: Integer;
end;
```

The `TTEXTMETRIC` record's fields contain much of the same information we've already discussed about fonts. For example, it shows a font's height, average character width, and whether the font is underlined, italicized, struck out, and so on. Refer to the Win32 API online help for detailed information on the `TTEXTMETRIC` structure. Listing 8.10 shows the code for the font information form.

LISTING 8.10 Source to the Font Information Form

```
unit FontInfoFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;

type

  TFontInfoForm = class(TForm)
    lbFontInfo: TListBox;
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

continues

LISTING 8.10 Continued

```
var
  FontInfoForm: TFontInfoForm;

implementation
uses MainFrm;

{$R *.DFM}

procedure TFontInfoForm.FormActivate(Sender: TObject);
const
  PITCH_MASK: byte = $0F;  // Set the lower order four bits
  FAMILY_MASK: byte = $F0; // Set to higher order four bits
var
  TxMetric: TTextMetric;
  FaceName: String;
  PitchTest, FamilyTest: byte;
begin

  // Allocate memory for FaceName string
  SetLength(FaceName, lf_FaceSize+1);

  // First get the font information
  with MainForm.pbxFont.Canvas do
  begin
    GetTextFace(Handle, lf_faceSize-1, PChar(FaceName));
    GetTextMetrics(Handle, TxMetric);
  end;

  // Now add the font information to the listbox from
  // the TTEXTMETRIC structure.
  with lbFontInfo.Items, TxMetric do
  begin
    Clear;
    Add('Font face name:     '+FaceName);
    Add('tmHeight:     '+IntToStr(tmHeight));
    Add('tmAscent:     '+IntToStr(tmAscent));
    Add('tmDescent:     '+IntToStr(tmDescent));
    Add('tmInternalLeading:     '+IntToStr(tmInternalLeading));
    Add('tmExternalLeading:     '+IntToStr(tmExternalLeading));
    Add('tmAveCharWidth:     '+IntToStr(tmAveCharWidth));
    Add('tmMaxCharWidth:     '+IntToStr(tmMaxCharWidth));
```

```
Add('tmWeight:     '+IntToStr(tmWeight));

if tmItalic <> 0  then
  Add('tmItalic: YES')
else
  Add('tmItalic: NO');

if tmUnderlined <> 0 then
  Add('tmUnderlined: YES')
else
  Add('tmUnderlined: NO');

if tmStruckOut <> 0 then
  Add('tmStruckOut: YES')
else
  Add('tmStruckOut: NO');

// Check the font's pitch type
PitchTest := tmPitchAndFamily and PITCH_MASK;
if (PitchTest and TMPF_FIXED_PITCH) = TMPF_FIXED_PITCH then
  Add('tmPitchAndFamily-Pitch: Fixed Pitch');
if (PitchTest and TMPF_VECTOR) = TMPF_VECTOR then
  Add('tmPitchAndFamily-Pitch: Vector');
if (PitchTest and TMPF_TRUETYPE) = TMPF_TRUETYPE then
  Add('tmPitchAndFamily-Pitch: True type');
if (PitchTest and TMPF_DEVICE) = TMPF_DEVICE then
  Add('tmPitchAndFamily-Pitch: Device');
if PitchTest = 0 then
  Add('tmPitchAndFamily-Pitch: Unknown');

// Check the fonts family type
FamilyTest := tmPitchAndFamily and FAMILY_MASK;
if (FamilyTest and FF_ROMAN) = FF_ROMAN then
  Add('tmPitchAndFamily-Family: FF_ROMAN');
if (FamilyTest and FF_SWISS) = FF_SWISS then
  Add('tmPitchAndFamily-Family: FF_SWISS');
if (FamilyTest and FF_MODERN) = FF_MODERN then
  Add('tmPitchAndFamily-Family: FF_MODERN');
if (FamilyTest and FF_SCRIPT) = FF_SCRIPT then
  Add('tmPitchAndFamily-Family: FF_SCRIPT');
if (FamilyTest and FF_DECORATIVE) = FF_DECORATIVE then
  Add('tmPitchAndFamily-Family: FF_DECORATIVE');
```

continues

LISTING 8.10 Continued

```
    if FamilyTest = 0 then
      Add('tmPitchAndFamily-Family: Unknown');

    Add('tmCharSet:     '+IntToStr(tmCharSet));
    Add('tmOverhang:     '+IntToStr(tmOverhang));
    Add('tmDigitizedAspectX:     '+IntToStr(tmDigitizedAspectX));
    Add('tmDigitizedAspectY:     '+IntToStr(tmDigitizedAspectY));
  end;
end;

end.
```

The `FormActive()` method first retrieves the font's name with the Win32 API function `GetTextFace()`, which takes a device context, a buffer size, and a null-terminated character buffer as parameters. `FormActivate()` then uses `GetTextMetrics()` to fill `TxMetric`, a `TTEXTMETRIC` record structure, for the selected font. The event handler then adds the values in `TxMetric` to the list box as strings. For the `tmPitchAndFamily` value, you mask out the high- or low-order bit, depending on the value you're testing for, and add the appropriate values to the list box. Figure 8.29 shows `FontInfoForm` displaying information about a font.

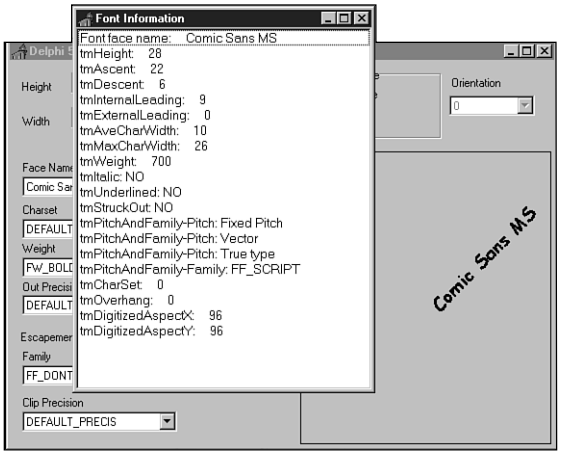

FIGURE 8.29

The font information form.

Summary

This chapter presented you with a lot of information about the Win32 Graphics Device Interface. We discussed Delphi 5's `TCanvas`, its properties, and its drawing methods. We also discussed Delphi 5's representation of images with its `TImage` component as well as mapping modes and Win32 coordinates systems. You saw how you can use graphics programming techniques to build a paint program and perform simple animation. Finally, we discussed fonts—how to create them and how to display information about them. One of the nice things about the GDI is that working with it can be a lot of fun. Entire books have been written on this subject alone. Take some time to experiment with the drawing routines, create your own fonts, or just fool around with the mapping modes to see what type of effects you can get.

Printing in Delphi 5

CHAPTER

10

IN THIS CHAPTER

Printing in Windows has been the bane of many a Windows programmer. However, don't be discouraged; Delphi simplifies most of what you need to know for printing. You can write simple printing routines to output text or bitmapped images with little effort. For more complex printing, a few concepts and techniques are all you really need to enable you to perform any type of custom printing. When you have that, printing isn't so difficult.

NOTE

You'll find a set of reporting components by QuSoft on the QReport page of the Component Palette. The documentation for this tool is located in the help file `QuickRpt.hlp`.

QuSoft's tools are suitable for applications that generate complex reports. However, they limit you from getting to the nuts and bolts of printing at the source-code level, where you have more control over what gets printed. This chapter doesn't cover QuickReports; instead, it covers creating your own reports in Delphi.

Delphi's `TPrinter` object, which encapsulates the Windows printing engine, does a great deal for you that you would otherwise have to handle yourself.

This chapter teaches you how to perform a whole range of printing operations by using `TPrinter`. You learn the simple tasks that Delphi has made much easier for generating printouts. You also learn the techniques for creating advanced printing routines that should start you on your way to becoming a printing guru.

The TPrinter Object

The `TPrinter` object encapsulates the Windows printing interface, making most of the printing management invisible to you. `TPrinter`'s methods and properties enable you to print onto its canvas as though you were drawing your output to a form's surface. The function `Printer()` returns a global `TPrinter` instance the first time it's called. `TPrinter`'s properties and methods are listed in Tables 10.1 and 10.2.

TABLE 10.1 `TPrinter` Properties

Property	*Purpose*
`Aborted`	Boolean variable that determines whether the user has aborted the print job.
`Canvas`	The printing surface for the current page.
`Fonts`	Contains a list of fonts supported by the printer.
`Handle`	A unique number representing the printer's device handle. See the sidebar "Handles" in Chapter 20, "Key Elements of the Visual Component Library."

Property	*Purpose*
`Orientation`	Determines horizontal (`poLandScape`) or vertical (`poPortrait`) printing.
`PageHeight`	Height, in pixels, of the printed page's surface.
`PageNumber`	Indicates the page being printed. This is incremented with each subsequent call to `TPrinter.NewPage()`.
`PageWidth`	Width, in pixels, of the printed page's surface.
`PrinterIndex`	Indicates the selected printer from the available printers on the user's system.
`Printers`	A list of the available printers on the system.
`Printing`	Determines whether a print job is printing.
`Title`	Text appearing in the Print Manager and on networked pages.

TABLE 10.2 `TPrinter` Methods

Method	*Purpose*
`Abort`	Terminates a print job.
`BeginDoc`	Begins a print job.
`EndDoc`	Ends a print job. (`EndDoc` ends a print job when printing is finished; `Abort` can terminate the job before printing is complete.)
`GetPrinter`	Retrieves the current printer.
`NewPage`	Forces the printer to start printing on a new page and increments the `PageCount` property.
`SetPrinter`	Specifies a printer as a current printer.

TPrinter.Canvas

`TPrinter.Canvas` is much like the canvas for your form; it represents the drawing surface on which text and graphics are drawn. The difference is that `TPrinter.Canvas` represents the drawing surface for your printed output as opposed to your screen. Most of the routines you use to draw text, to draw shapes, and to display images are used in the same manner for printed output. When printing, however, you must take into account some differences:

- Drawing to the screen is *dynamic*—you can erase what you've placed on the screen's output. Drawing to the printer isn't so flexible. What's drawn to the `TPrinter.Canvas` is printed to the printer.
- Drawing text or graphics to the screen is nearly instantaneous, whereas drawing to the printer is slow, even on some high-performance laser printers. You therefore must allow

users to abort a print job either by using an Abort dialog box or by some other method that enables them to terminate the print job.

- Because your users are running Windows, you can assume that their display supports graphics output. However, you can't assume the same for their printers. Different printers have different capabilities. Some printers may be high-resolution printers; other printers may be very low resolution and may not support graphics printing at all. You must take this into account in your printing routines.
- You'll never see an error message like this:

  ```
  Display ran out of screen space,
  please insert more screen space into your display.
  ```

  But you can bet that you'll see an error telling you that the printer ran out of paper. Windows NT/2000 and Windows 95/98 both provide error handling when this occurs. However, you should provide a way for the user to cancel the printout when this occurs.
- Text and graphics on your screen don't look the same on hard copy. Printers and displays have very different resolutions. That 300×300 bitmap might look spectacular on a 640×480 display, but it's a mere 1×1-inch square blob on your 300 dpi (dots per inch) laser printer. You're responsible for making adjustments to your drawing routines so that your users won't need a magnifying glass to read their printed output.

Simple Printing

In many cases, you want to send a stream of text to your printer without any regard for special formatting or placement of the text. Delphi facilitates simple printing, as the following sections illustrate.

Printing the Contents of a TMemo Component

Printing lines of text is actually quite simple using the `AssignPrn()` procedure. The `AssignPrn()` procedure enables you to assign a text file variable to the current printer. It's used with the `Rewrite()` and `CloseFile()` procedures. The following lines of code illustrate this syntax:

```
var
  f: TextFile;
begin
  AssignPrn(f);
  try
    Rewrite(f);
    writeln(f, 'Print the output');
  finally
```

```
    CloseFile(f);
  end;
end;
```

Printing a line of text to the printer is the same as printing a line of text to a file. You use this syntax:

```
writeln(f, 'This is my line of text');
```

In Chapter 16, "MDI Applications," you add menu options for printing the contents of the `TMdiEditForm` form. Listing 10.1 shows you how to print the contents from `TMdiEditForm`. You'll use this same technique for printing text from just about any source.

LISTING 10.1 Printing Code for `TMdiEditForm`

```
procedure TMdiEditForm.mmiPrintClick(Sender: TObject);
var
  i: integer;
  PText: TextFile;
begin
  inherited;
  if PrintDialog.Execute then
  begin
    AssignPrn(PText);
    Rewrite(PText);
    try
      Printer.Canvas.Font := memMainMemo.Font;
      for i := 0 to memMainMemo.Lines.Count -1 do
        writeln(PText, memMainMemo.Lines[i]);
    finally
      CloseFile(PText);
    end;
  end;
end;
```

Notice that the memo's font also was assigned to the `Printer`'s font, causing the output to print with the same font as `memMainMemo`.

CAUTION

Be aware that the printer will print with the font specified by `Printer.Font` only if the printer supports that font. Otherwise, the printer will use a font that approximates the characteristics of the specified font.

Printing a Bitmap

Printing a bitmap is simple as well. The `MdiApp` example in Chapter 16, "MDI Applications," shows how to print the contents of a bitmap in `TMdiBmpForm`. This event handler is shown in Listing 10.2.

LISTING 10.2 Printing Code for `TMdiBmpForm`

```
procedure TMdiBMPForm.mmiPrintClick(Sender: TObject);
begin
  inherited;

  with ImgMain.Picture.Bitmap do
  begin
    Printer.BeginDoc;
    Printer.Canvas.StretchDraw(Canvas.ClipRect, imgMain.Picture.Bitmap);
    Printer.EndDoc;
  end; { with }
end;
```

Only three lines of code are needed to print the bitmap using the `TCanvas.StretchDraw()` method. This vast simplification of printing a bitmap is made possible by the fact that since Delphi 3, bitmaps are in DIB format by default, and DIBs are what the printer driver requires. If you happen to have a handle to a bitmap that isn't in DIB format, you can copy (`Assign`) it into a temporary `TBitmap`, force the temporary bitmap into DIB format by assigning `bmDIB` to the `TBitmap.HandleType` property, and then print from the new DIB.

NOTE

One of the keys to printing is to be able to print images as they appear onscreen at approximately the same size. A 3×3-inch image on a 640×480 pixel screen uses fewer pixels than it would on a 300 dpi printer, for example. Therefore, stretch the image to `TPrinter`'s canvas as was done in the example in the call to `StretchDIBits()`. Another technique is to draw the image using a different mapping mode, as described in Chapter 8, "Graphics Programming with GDI and Fonts." Keep in mind that some older printers may not support the stretching of images. You can obtain valuable information about the printer's capabilities by using the Win32 API function `GetDeviceCaps()`.

Printing Rich Text–Formatted Data

Printing the contents of a `TRichEdit` component is a matter of one method call. The following code shows how to do this (this is also the code for printing `TMdiRtfForm` in the `MdiApp` example in Chapter 16, "MDI Applications"):

```
procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
  inherited;
  reMain.Print(Caption);
end;
```

Printing a Form

Conceptually, printing a form can be one of the more difficult tasks to perform. However, this task has been simplified greatly thanks to VCL's `Print()` method of `TForm`. The following one-line procedure prints your form's client areas as well as all components residing in the client area:

```
procedure TForm1.PrintMyForm(Sender: TObject);
begin
  Print;
end;
```

NOTE

Printing your form is a quick-and-dirty way to print graphical output. However, only what's visible onscreen will be printed, due to Windows' clipping. Also, the bitmap is created at screen pixel density and then stretched to printer resolution. Text on the form is not drawn at printer resolution; it's drawn at screen resolution and stretched, so overall the form will be noticeably jagged and blocky. You must use more elaborate techniques to print complex graphics; these techniques are discussed later in this chapter.

Advanced Printing

Often you need to print something very specific that isn't facilitated by the development tool you're using or a third-party reporting tool. In this case, you need to perform the low-level printing tasks yourself. The next several sections show you how to write such printing routines and present a methodology you can apply to all your printing tasks.

NOTE

Although this section covers printing, you should know that at the time of this writing, several third-party printing components are available that should handle most of your printing needs. You'll find demos of some of these tools on the CD with this book.

Printing a Columnar Report

Many applications, particularly those using databases, print some type of report. One common report style is the columnar report.

The next project prints a columnar report from one of the tables in Delphi's demo directories. Each page contains a header, column titles, and then the record list. Each subsequent page also has the header and column titles preceding the record list.

Figure 10.1 shows the main form for this project. The `TEdit`/`TUpDown` pairs enable the user to specify the column widths in tenths of inches. By using the `TUpDown` components, you can specify minimum and maximum values. The `TEdit1` control, `edtHeaderFont`, contains a header that can be printed using a font that differs from the one used for the rest of the report.

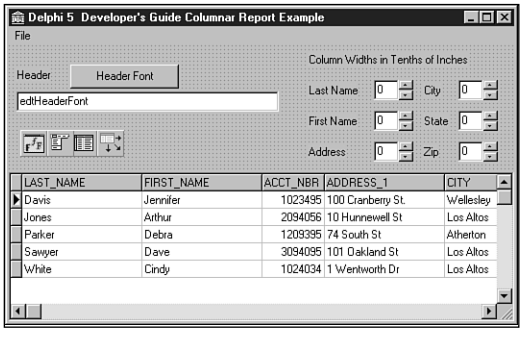

FIGURE 10.1
Columnar report main form.

Listing 10.3 shows the source code for the project. The `mmiPrintClick()` event handler basically performs the following steps:

1. Initiates a print job.
2. Prints a header.
3. Prints column names.

4. Prints a page.
5. Continues steps 2, 3, and 4 until printing finishes.
6. Ends the print job.

LISTING 10.3 Columnar Report Demo

```
unit MainFrm;
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, DBGrids, DB, DBTables, Menus, StdCtrls, Spin,
  Gauges, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
   { components not included in listing,
     please refer to CD source }
  procedure mmiPrintClick(Sender: TObject);
    procedure btnHeaderFontClick(Sender: TObject);
  private
    PixelsInInchx: integer;
    LineHeight: Integer;
   { Keeps track of vertical space in pixels, printed on a page }
    AmountPrinted: integer;
   { Number of pixels in 1/10 of an inch. This is used for line spacing }
    TenthsOfInchPixelsY: integer;
    procedure PrintLine(Items: TStringList);
    procedure PrintHeader;
    procedure PrintColumnNames;
  end;

var
  MainForm: TMainForm;

implementation
uses printers, AbortFrm;

{$R *.DFM}

procedure TMainForm.PrintLine(Items: TStringList);
var
  OutRect: TRect;
  Inches: double;
  i: integer;
```

continues

LISTING 10.3 Continued

```
begin
  // First position the print rect on the print canvas
  OutRect.Left := 0;
  OutRect.Top := AmountPrinted;
  OutRect.Bottom := OutRect.Top + LineHeight;
  With Printer.Canvas do
    for i := 0 to Items.Count - 1 do
    begin
      Inches := longint(Items.Objects[i]) * 0.1;
      // Determine Right edge
      OutRect.Right := OutRect.Left + round(PixelsInInchx*Inches);
      if not Printer.Aborted then
       // Print the line
        TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
      // Adjust right edge
      OutRect.Left := OutRect.Right;
    end;
 { As each line prints, AmountPrinted must increase to reflect how
   much of a page has been printed on based on the line height. }
  AmountPrinted := AmountPrinted + TenthsOfInchPixelsY*2;
end;

procedure TMainForm.PrintHeader;
var
  SaveFont: TFont;
begin
 { Save the current printer's font, then set a new print font based
  on the selection for Edit1 }
  SaveFont := TFont.Create;
  try
    Savefont.Assign(Printer.Canvas.Font);
    Printer.Canvas.Font.Assign(edtHeaderFont.Font);
    // First print out the Header
    with Printer do
    begin
      if not Printer.Aborted then
        Canvas.TextOut((PageWidth div 2)-(Canvas.TextWidth(edtHeaderFont.Text)
                        div 2),0, edtHeaderFont.Text);
     // Increment AmountPrinted by the LineHeight
      AmountPrinted := AmountPrinted + LineHeight+TenthsOfInchPixelsY;
    end;
    // Restore the old font to the Printer's Canvas property
    Printer.Canvas.Font.Assign(SaveFont);
  finally
    SaveFont.Free;
```

```
  end;
end;

procedure TMainForm.PrintColumnNames;
var
  ColNames: TStringList;
begin
  { Create a TStringList to hold the column names and the
    positions where the width of each column is based on values
    in the TEdit controls. }
  ColNames := TStringList.Create;
  try
    // Print the column headers using a bold/underline style
    Printer.Canvas.Font.Style := [fsBold, fsUnderline];

    with ColNames do
    begin
      // Store the column headers and widths in the TStringList object
      AddObject('LAST NAME',  pointer(StrToInt(edtLastName.Text)));
      AddObject('FIRST NAME', pointer(StrToInt(edtFirstName.Text)));
      AddObject('ADDRESS',    pointer(StrToInt(edtAddress.Text)));
      AddObject('CITY',       pointer(StrToInt(edtCity.Text)));
      AddObject('STATE',      pointer(StrToInt(edtState.Text)));
      AddObject('ZIP',        pointer(StrToInt(edtZip.Text)));
    end;

    PrintLine(ColNames);
    Printer.Canvas.Font.Style := [];
  finally
    ColNames.Free;   // Free the column name TStringList instance
  end;
end;

procedure TMainForm.mmiPrintClick(Sender: TObject);
var
  Items: TStringList;
begin
  { Create a TStringList instance to hold the fields and the widths
    of the columns in which they'll be drawn based on the entries in
    the edit controls }
  Items := TStringList.Create;
  try
    // Determine pixels per inch horizontally
    PixelsInInchx := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
    TenthsOfInchPixelsY := GetDeviceCaps(Printer.Handle,
                           LOGPIXELSY) div 10;
```

continues

LISTING 10.3 Continued

```
AmountPrinted := 0;
MainForm.Enabled := False; // Disable the parent form
try
  Printer.BeginDoc;
  AbortForm.Show;
  Application.ProcessMessages;
  { Calculate the line height based on text height using the
    currently rendered font }
  LineHeight := Printer.Canvas.TextHeight('X')+TenthsOfInchPixelsY;
  if edtHeaderFont.Text <> '' then
    PrintHeader;
  PrintColumnNames;
  tblClients.First;
  { Store each field value in the TStringList as well as its
    column width }
  while (not tblClients.Eof) or Printer.Aborted do
  begin

    Application.ProcessMessages;
    with Items do
    begin
      AddObject(tblClients.FieldByName('LAST_NAME').AsString,
                  pointer(StrToInt(edtLastName.Text)));
      AddObject(tblClients.FieldByName('FIRST_NAME').AsString,
                  pointer(StrToInt(edtFirstName.Text)));
      AddObject(tblClients.FieldByName('ADDRESS_1').AsString,
                  pointer(StrToInt(edtAddress.Text)));
      AddObject(tblClients.FieldByName('CITY').AsString,
                  pointer(StrToInt(edtCity.Text)));
      AddObject(tblClients.FieldByName('STATE').AsString,
                  pointer(StrToInt(edtState.Text)));
      AddObject(tblClients.FieldByName('ZIP').AsString,
                  pointer(StrToInt(edtZip.Text)));
    end;
    PrintLine(Items);
    { Force print job to begin a new page if printed output has
      exceeded page height }
    if AmountPrinted + LineHeight > Printer.PageHeight then
    begin
      AmountPrinted := 0;
      if not Printer.Aborted then
        Printer.NewPage;
      PrintHeader;
      PrintColumnNames;
```

```
        end;
        Items.Clear;
        tblClients.Next;
      end;
      AbortForm.Hide;
      if not Printer.Aborted then
        Printer.EndDoc;
    finally
      MainForm.Enabled := True;
    end;
  finally
    Items.Free;
  end;
end;

procedure TMainForm.btnHeaderFontClick(Sender: TObject);
begin
 { Assign the font selected with FontDialog1 to Edit1. }
  FontDialog.Font.Assign(edtHeaderFont.Font);
  if FontDialog.Execute then
    edtHeaderFont.Font.Assign(FontDialog.Font);
end;

end.
```

`mmiPrintClick()` first creates a `TStringList` instance to hold the strings for a line to be printed. Then the number of pixels per inch along the vertical axis is determined in `PixelsPerInchX`, which is used to calculate column widths. `TenthsOfInchPixelsY` is used to space each line by 0.1 inch. `AmountPrinted` holds the total amount of pixels along the printed surface's vertical axis for each line printed. This is required to determine whether to start a new page when `AmountPrinted` exceeds `Printer.PageHeight`.

If a header exists in `edtHeaderFont.Text`, it's printed in `PrintHeader()`. `PrintColumnNames()` prints the names of the columns for each field to be printed. (These two procedures are discussed later in this section.) Finally, the table's records are printed.

The following loop increments through `tblClients` records and prints selected fields within each of the records:

```
while (not tblClients.Eof) or Printer.Aborted do begin
```

Within the loop, the field values are added to the `TStringList` using the `AddObject()` method. Here, you store both the string and the column width. The column width is added to the `Items.Objects` array property. `Items` is then passed to the `PrintLine()` procedure, which prints the strings in a columnar format.

In much of the previous code, you saw references to `Printer.Aborted`. This is a test to determine whether the user has aborted the print job, which is covered in the next section.

TIP

The `TStrings` and `TStringList`'s `Objects` array properties are a convenient place to store integer values. Using `AddObject()` or `InsertObject()`, you can hold any number up to `MaxLongInt`. Because `AddObject()` expects a `TObject` reference as its second parameter, you must typecast that parameter as a pointer, as shown in the following code:

```
MyList.AddObject('SomeString', pointer(SomeInteger));
```

To retrieve the value, use a `Longint` typecast:

```
MyInteger := Longint(MyList.Objects[Index]);
```

The event handler then determines whether printing a new line will exceed the page height:

```
if AmountPrinted + LineHeight > Printer.PageHeight then
```

If this evaluates to `True`, `AmountPrinted` is set back to `0`, `Printer.NewPage` is invoked to print a new page, and the header and column names are printed again. `Printer.EndDoc` is called to end the print job after the `tblClients` records have printed.

The `PrintHeader()` procedure prints the header centered at the top of the report using `edtHeaderFont.Text` and `edtHeaderFont.Font`. `AmountPrinted` is then incremented and `Printer`'s font is restored to its original style.

As the name implies, `PrintColumnNames()` prints the column names of the report. In this method, names are added to a `TStringList` object, `ColNames`, which then is passed to `PrintLine()`. Notice that the column names are printed in a bold, underlined font. Setting `Printer.Canvas.Font` accordingly does this.

The `PrintLine()` procedure takes a `TStringList` argument called `Items` and prints each string in `Items` on a single line in a columnar manner. The variable `OutRect` holds values for a binding rectangle at a location on `Printer`'s canvas to which the text is drawn. `OutRect` is passed to `TextRect()`, along with the text to draw. By multiplying `Items.Object[i]` by 0.1, `OutRect.Right`'s value is obtained because `Items.Objects[i]` is in tenths of inches. Inside the `for` loop, `OutRect` is recalculated along the same X-axis to position it to the next column and draw the next text value. Finally, `AmountPrinted` is incremented by `LineHeight + TenthsOfInchPixelsY`.

Although this report is fully functional, you might consider extending it to include a footer, page numbers, and even margin settings.

Aborting the Printing Process

Earlier in this chapter, you learned that your users need a way to terminate printing after they've initiated it. The `TPrinter.Abort()` procedure and the `Aborted` property help you do this. The code in Listing 10.3 contains such logic. To add abort logic to your printing routines, your code must meet these three conditions:

- You must establish an event that, when activated, calls `Printer.Abort`, thus aborting the printing process.
- You must check for `TPrinter.Aborted = True` before calling any of `TPrinter`'s print functions, such as `TextOut()`, `NewPage()`, and `EndDoc()`.
- You must end your printing logic by checking the value of `TPrinter.Aborted` for `True`.

A simple Abort dialog box can satisfy the first condition. You used such a dialog box in the preceding example. This dialog box should contain a button that will invoke the abort process.

This button's event handler should simply call `TPrinter.Abort`, which terminates the print job and cancels any printing requests made to `TPrinter`.

In the unit `MainForm.pas`, examine the code to show `AbortForm` shortly after calling `TPrinter.Begindoc()`:

```
Printer.BeginDoc;
AbortForm.Show;
Application.ProcessMessages;
```

Because `AbortForm` is shown as a modeless dialog box, the call to `Application.ProcessMessages` ensures that it's drawn properly before any processing of the printing logic continues.

To satisfy the second condition, the test for `Printer.Aborted = True` is performed before calling any `TPrinter` methods. The `Aborted` property is set to `True` when the `Abort()` method is called from `AbortForm`. As an example, before you call `Printer.TextRect`, check for `Aborted = True`:

```
if not Printer.Aborted then
  TextRect(OutRect, OutRect.Left, OutRect.Top, Items[i]);
```

Also, you shouldn't call `EndDoc()` or any of `TPrinter.Canvas`'s drawing routines after calling `Abort()`, because the printer has been effectively closed.

To satisfy the third condition in this example, `while not Table.Eof` also checks whether the value of `Printer.Aborted` is `True`, which causes execution to jump out of the loop where the print logic is executed.

Printing Envelopes

The preceding example showed you a method for printing a columnar report. Although this technique was somewhat more complicated than sending a series of `writeln()` calls to the printer, it's still, for the most part, a line-by-line print. Printing envelopes introduces a few factors that complicate things a bit further and are common to most types of printing you'll do in Windows. First, the objects (items) you must print probably need to be positioned at some specific location on the printed surface. Second, the items' *metrics*, or units of measurement, can be completely different from those of the printer canvas. Taking these two factors into account, printing becomes much more than just printing a line and keeping track of how much print space you've used.

This envelope-printing example shows you a step-by-step process you can use to print just about anything. Keep in mind that everything drawn on the printer's canvas is drawn within some bounding rectangle on the canvas or to specific points on the printer canvas.

Printing in the Abstract

Think of the printing task in a more abstract sense for a moment. In all cases, two things are certain: You have a surface on which to print, and you have one or more elements to plot onto that surface. Take a look at Figure 10.2.

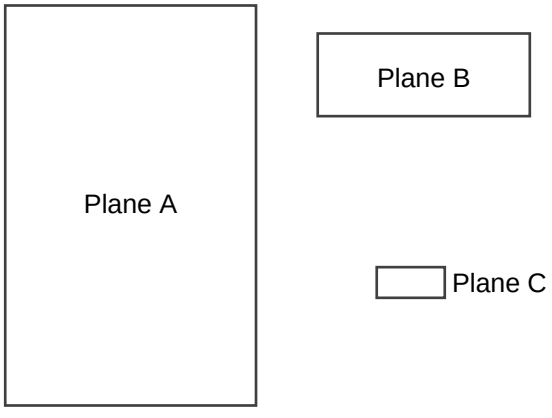

FIGURE 10.2
Three planes.

In Figure 10.2, Plane A is your destination surface. Planes B and C are the elements you want to superimpose (print) onto Plane A. Assume a coordinate system for each plane where the unit of measurement increases as you travel east along the X-axis and south along the Y-axis—that is, unless you live in Australia. Figure 10.3 depicts this coordinate system. The result of combining the planes is shown in Figure 10.4.

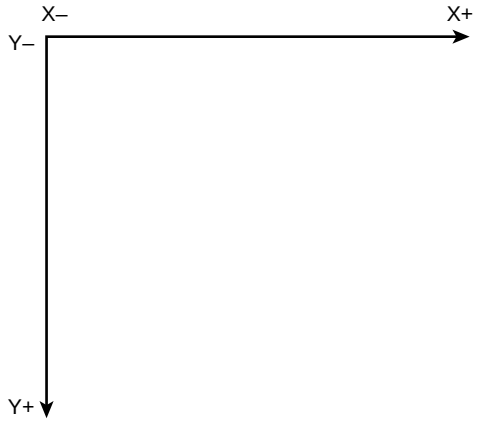

FIGURE 10.3
The Plane A, B, and C coordinate system.

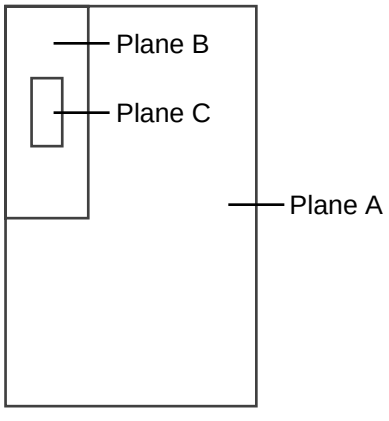

FIGURE 10.4
Planes B and C superimposed on Plane A.

Notice that Planes B and C were rotated by 90 degrees to achieve the final result. So far, this doesn't appear to be too bad. Given that your planes are measured using the same unit of measurement, you can easily draw out these rectangles to achieve the final result with some simple geometry. But what if they're not the same unit of measurement?

Suppose that Plane A represents a surface for which the measurements are given in pixels. Its dimensions are 2,550×3,300 pixels. Plane B is measured in inches: 6½×3¾ inches. Suppose

that you don't know the dimensions for Plane C; you do know, however, that it's measured in pixels, and you'll know its measurements later. These measurements are illustrated in Figure 10.5.

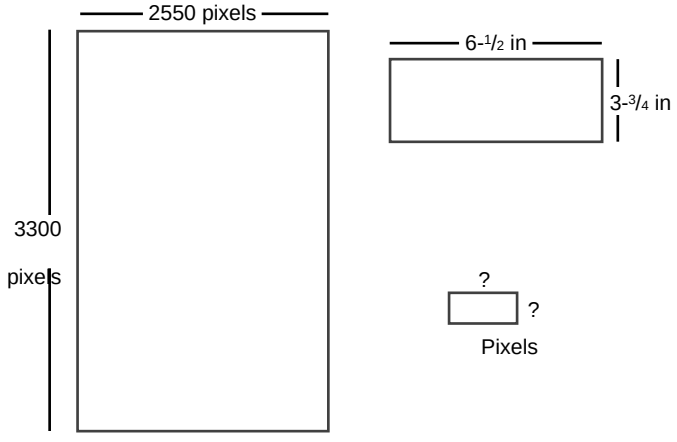

FIGURE 10.5
Plane measurements.

This abstraction illustrates the problem associated with printing. In fact, it illustrates the very task of printing an envelope. Plane A represents a printer's page size on a 300 dpi printer (at 300 dpi, 8½×11 inches equals 2,550×3,300 pixels). Plane B represents the envelope's size in inches, and Plane C represents the bounding rectangle for the text making up the address. Keep in mind, however, that this abstraction isn't tied to just envelopes. Planes B and C might represent `TImage` components measured in millimeters.

By looking at this task in its abstraction, you've achieved the first three steps to printing in Windows: Identify each element to print, identify the unit of measurement for the destination surface, and identify the units of measurement for each individual element to be plotted onto the destination surface.

Now consider another twist—literally. When you're printing an envelope in a vertical fashion, the text must rotate vertically.

A Step-by-Step Process for Printing

The following list summarizes the process you should follow when laying out your printed output in code:

1. Identify each element to be printed to the destination surface.
2. Identify the unit of measurement for the destination surface or printer canvas.

3. Identify the units of measurement for each individual element to be plotted onto the destination surface.
4. Decide on the common unit of measurement with which to perform all drawing routines. Almost always, this will be the printer canvas's units—pixels.
5. Write the translation routines to convert the other units of measurement to that of the common unit of measurement.
6. Write the routines to calculate the size for each element to print in the common unit of measurement. In Object Pascal, this can be represented by a `TPoint` structure. Keep in mind dependencies on other values. For example, the address's bounding rectangle is dependent on the envelope's position. Therefore, the envelope's data must be calculated first.
7. Write the routines to calculate the position of each element as it will appear on the printer canvas, based on the printer canvas's coordinate system and the sizes obtained from step 6. In Object Pascal, this can be represented by a `TRect` structure. Again, keep dependencies in mind.
8. Write your printing function, using the data gathered from the previous steps, to position items on the printed surface.

NOTE

Steps 5 and 6 can be achieved by using a technique of performing all drawing in a specific mapping mode. Mapping modes are discussed in Chapter 8, "Graphics Programming with GDI and Fonts."

Getting Down to Business

Given the step-by-step process, your task of printing an envelope should be much clearer. You'll see this in the envelope-printing project. The first step is to identify the elements to print or represent. The elements for the envelope example are the envelope, itself, and the address.

In this example, you learn how to print two standard envelope sizes: a size 10 and a size 6¾.

The following record holds the envelope sizes:

```
type

  TEnvelope = record
    Kind: string;   // Stores the envelope type's name
    Width: double;  // Holds the width of the envelope
    Height: double; // Holds the height of the envelope
  end;
```

```
const
  // This constant array stores envelope types
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4
```

Steps 2 and 3 are covered: You know that the destination surface is the `TPrinter.Canvas`, which is represented in pixels. The envelopes are represented in inches, and the address is represented in pixels. Step 4 requires you to select a common unit of measurement. For this project, you use pixels as the common unit of measurement.

For step 5, the only units you need to convert are from inches to pixels. The `GetDeviceCaps()` Win32 API function can return the amount of pixels per one inch along the horizontal and vertical axis for `Printer.Canvas`:

```
PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);
```

To convert the envelope's size to pixels, you just multiply the number of inches by `PixPerInX` or `PixPerInY` to get the horizontal or vertical measurement in pixels:

```
EnvelopeWidthInPixels := trunc(EnvelopeWidthValue * PixPerInX);
EnvelopeHeightInPixels := trunc(EnvelopeHeightValue * PixPerInY);
```

Because the envelope width or height can be a fractional value, it's necessary to use the `Trunc()` function to return the integer portion of the floating-point type rounded toward zero.

The sample project demonstrates how you would implement steps 6 and 7. The main form for this project is shown in Figure 10.6; Listing 10.4 shows the source code for the envelope-printing project.

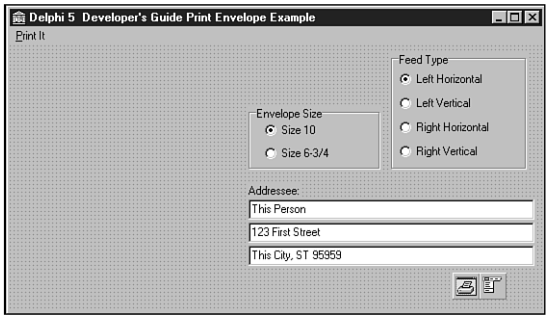

FIGURE 10.6
The main form for the envelope demo.

LISTING 10.4 Envelope Printing Demo

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, printers, StdCtrls, ExtCtrls, Menus, ComCtrls;

type

  TEnvelope = record
    Kind: string;   // Stores the envelope type's name
    Width: double;  // Holds the width of the envelope
    Height: double; // Holds the height of the envelope
  end;

const
  // This constant array stores envelope types
  EnvArray: array[1..2] of TEnvelope =
  ((Kind:'Size 10';Width:9.5;Height:4.125),     // 9-1/2 x 4-1/8
   (Kind:'Size 6-3/4';Width:6.5;Height:3.625)); // 6-1/2 x 3-3/4

type

  // This enumerated type represents printing positions.
  TFeedType = (epLHorz, epLVert, epRHorz, epRVert);

  TPrintPrevPanel = class(TPanel)
  public
    property Canvas; // Publicize the Canvas property
  end;

  TMainForm = class(TForm)
    gbEnvelopeSize: TGroupBox;
    rbSize10: TRadioButton;
    rbSize6: TRadioButton;
    mmMain: TMainMenu;
    mmiPrintIt: TMenuItem;
    lblAdressee: TLabel;
    edtName: TEdit;
    edtStreet: TEdit;
    edtCityState: TEdit;
    rgFeedType: TRadioGroup;
    PrintDialog: TPrintDialog;
```

continues

LISTING 10.4 Continued

```
    procedure FormCreate(Sender: TObject);
    procedure rgFeedTypeClick(Sender: TObject);
    procedure mmiPrintItClick(Sender: TObject);
  private
    PrintPrev: TPrintPrevPanel;  // Print  preview panel
    EnvSize: TPoint;     // Stores the envelope's size
    EnvPos: TRect;       // Stores the envelope's position
    ToAddrPos: TRect;    // Stores the address's position
    FeedType: TFeedType; // Stores the feed type from TEnvPosition
    function GetEnvelopeSize: TPoint;
    function GetEnvelopePos: TRect;
    function GetToAddrSize: TPoint;
    function GetToAddrPos: TRect;
    procedure DrawIt;
    procedure RotatePrintFont;
    procedure SetCopies(Copies: Integer);
  end;

var
  MainForm: TMainForm;

implementation
{$R *.DFM}

function TMainForm.GetEnvelopeSize: TPoint;
// Gets the envelope's size represented by a TPoint
var
  EnvW, EnvH: integer;
  PixPerInX,
  PixPerInY: integer;
begin
  // Pixels per inch along the horizontal axis
  PixPerInX := GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  // Pixels per inch along the vertical axis
  PixPerInY := GetDeviceCaps(Printer.Handle, LOGPIXELSY);

  // Envelope size differs depending on the user's selection
  if RBSize10.Checked then
  begin
    EnvW := trunc(EnvArray[1].Width * PixPerInX);
    EnvH := trunc(EnvArray[1].Height * PixPerInY);
  end
  else begin
    EnvW := trunc(EnvArray[2].Width * PixPerInX);
```

```
    EnvH := trunc(EnvArray[2].Height * PixPerInY);
  end;

  // return Result as a TPoint record
  Result := Point(EnvW, EnvH)
end;

function TMainForm.GetEnvelopePos: TRect;
{ Returns the envelope's position relative to its feed type. This
  function requires that the variable EnvSize be initialized }
begin
  // Determine feed type based on user's selection.
  FeedType := TFeedType(rgFeedType.ItemIndex);

  { Return a TRect structure indicating the envelope's
    position as it is ejected from the printer. }
  case FeedType of
   epLHorz:
     Result := Rect(0, 0, EnvSize.X, EnvSize.Y);
   epLVert:
     Result := Rect(0, 0, EnvSize.Y, EnvSize.X);
   epRHorz:
     Result := Rect(Printer.PageWidth - EnvSize.X, 0,
➥Printer.PageWidth, EnvSize.Y);
   epRVert:
     Result := Rect(Printer.PageWidth - EnvSize.Y, 0,
➥Printer.PageWidth, EnvSize.X);
  end; // Case
end;

function MaxLn(V1, V2: Integer): Integer;
// Returns the larger of the two. If equal, returns the first
begin
  Result := V1;    // Default result to V1 }
  if V1 < V2 then
    Result := V2
end;

function TMainForm.GetToAddrSize: TPoint;
var
  TempPoint: TPoint;
begin
  // Calculate the size of the longest line using the MaxLn() function
  TempPoint.x := Printer.Canvas.TextWidth(edtName.Text);
  TempPoint.x := MaxLn(TempPoint.x, Printer.Canvas.TextWidth(edtStreet.Text));
  TempPoint.x := MaxLn(TempPoint.x,
```

continues

LISTING 10.4 Continued

```
    Printer.Canvas.TextWidth(edtCityState.Text))+10;

  // Calculate the height of all the address lines
  TempPoint.y := Printer.Canvas.TextHeight(edtName.Text)+
    ➥Printer.Canvas.TextHeight(edtStreet.Text)+
    ➥Printer.Canvas.TextHeight(edtCityState.Text)+10;
  Result := TempPoint;
end;

function TMainForm.GetToAddrPos: TRect;
// This function requires that EnvSize, and EnvPos be initialized
Var
  TempSize: TPoint;
  LT, RB: TPoint;
begin
  // Determine the size of the Address bounding rectangle
  TempSize := GetToAddrSize;
  { Calculate two points, one representing the Left Top (LT) position
    and one representing the Right Bottom (RB) position of the
    address's bounding rectangle. This depends on the FeedType }
  case FeedType of
    epLHorz:
      begin
        LT := Point((EnvSize.x div 2) - (TempSize.x div 2),
          ((EnvSize.y div 2) - (TempSize.y div 2)));
        RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
      end;
    epLVert:
      begin
        LT := Point((EnvSize.y div 2) - (TempSize.y div 2),
          ((EnvSize.x div 2) - (TempSize.x div 2)));
        RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
      end;
    epRHorz:
    begin
      LT := Point((EnvSize.x div 2) - (TempSize.x div 2) + EnvPos.Left,
        ((EnvSize.y div 2) - (TempSize.y div 2)));
      RB := Point(LT.x + TempSize.x, LT.y + TempSize.Y);
    end;
    epRVert:
    begin
      LT := Point((EnvSize.y div 2) - (TempSize.y div 2) + EnvPos.Left,
        ((EnvSize.x div 2) - (TempSize.x div 2)));
      RB := Point(LT.x + TempSize.y, LT.y + TempSize.x);
    end;
```

```
  end; // End Case

  Result := Rect(LT.x, LT.y, RB.x, RB.y);
end;

procedure TMainForm.DrawIt;
// This procedure assumes that EnvPos and EnvSize have been initialized
begin
  PrintPrev.Invalidate;   // Erase contents of Panel
  PrintPrev.Update;
  // Set the mapping mode for the panel to MM_ISOTROPIC
  SetMapMode(PrintPrev.Canvas.Handle, MM_ISOTROPIC);
  // Set the TPanel's extent to match that of the printer boundaries.
  SetWindowExtEx(PrintPrev.Canvas.Handle,
      Printer.PageWidth, Printer.PageHeight, nil);
  // Set the viewport extent to that of the PrintPrev TPanel size.
  SetViewPortExtEx(PrintPrev.Canvas.Handle,
      PrintPrev.Width, PrintPrev.Height, nil);
  // Set the origin to the position at 0, 0
  SetViewportOrgEx(PrintPrev.Canvas.Handle, 0, 0, nil);
  PrintPrev.Brush.Style := bsSolid;

  with EnvPos do
   // Draw a rectangle to represent the envelope
    PrintPrev.Canvas.Rectangle(Left, Top, Right, Bottom);

  with ToAddrPos, PrintPrev.Canvas do
    case FeedType of
    epLHorz, epRHorz:
      begin
        Rectangle(Left, Top, Right, Top+2);
        Rectangle(Left, Top+(Bottom-Top) div 2, Right,
➥Top+(Bottom-Top) div 2+2);
        Rectangle(Left, Bottom, Right, Bottom+2);
      end;
    epLVert, epRVert:
      begin
        Rectangle(Left, Top, Left+2, Bottom);
        Rectangle(Left + (Right-Left)div 2, Top,
➥Left + (Right-Left)div 2+2, Bottom);
        Rectangle(Right, Top, Right+2, Bottom);
      end;
    end; // case
end;

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

continues

LISTING 10.4 Continued

```
var
  Ratio: double;
begin
  // Calculate a ratio of PageWidth to PageHeight
  Ratio := Printer.PageHeight / Printer.PageWidth;

  // Create a new TPanel instance
  with TPanel.Create(self) do
  begin
    SetBounds(15, 15, 203, trunc(203*Ratio));
    Color := clBlack;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    Parent := self;
  end;

  // Create a Print preview panel
  PrintPrev := TPrintPrevPanel.Create(self);

  with PrintPrev do
  begin
    SetBounds(10, 10, 200, trunc(200*Ratio));
    Color := clWhite;
    BevelInner := bvNone;
    BevelOuter := bvNone;
    BorderStyle := bsSingle;
    Parent := self;
  end;

end;

procedure TMainForm.rgFeedTypeClick(Sender: TObject);
begin
  EnvSize := GetEnvelopeSize;
  EnvPos := GetEnvelopePos;
  ToAddrPos := GetToAddrPos;
  DrawIt;
end;

procedure TMainForm.SetCopies(Copies: Integer);
var
  ADevice, ADriver, APort: String;
  ADeviceMode: THandle;
  DevMode: PDeviceMode;
```

```
begin
 SetLength(ADevice, 255);
 SetLength(ADriver, 255);
 SetLength(APort, 255);

 { If ADeviceMode is zero, a printer driver is not loaded. Therefore,
   setting PrinterIndex forces the driver to load. }
 if ADeviceMode = 0 then
 begin
   Printer.PrinterIndex := Printer.PrinterIndex;
   Printer.GetPrinter(PChar(ADevice), PChar(ADriver),
➥PChar(APort), ADeviceMode);
 end;

 if ADeviceMode <> 0 then
 begin
   DevMode := GlobalLock(ADeviceMode);
   try
     DevMode^.dmFields := DevMode^.dmFields or DM_Copies;
     DevMode^.dmCopies := Copies;
   finally
     GlobalUnlock(ADeviceMode);
   end;
 end
 else
   raise Exception.Create('Could not set printer copies');
end;

procedure TMainForm.mmiPrintItClick(Sender: TObject);
var
  TempHeight: integer;
  SaveFont: TFont;
begin
  if PrintDialog.Execute then
  begin
    // Set the number of copies to print
    SetCopies(PrintDialog.Copies);
    Printer.BeginDoc;
    try
      // Calculate a temporary line height
      TempHeight := Printer.Canvas.TextHeight(edtName.Text);
      with ToAddrPos do
      begin
        { When printing vertically, rotate the font such that it paints
          at a 90 degree angle. }
        if (FeedType = eplVert) or (FeedType = epRVert) then
```

continues

LISTING 10.4 Continued

```
        begin
          SaveFont := TFont.Create;
          try
            // Save the original font
            SaveFont.Assign(Printer.Canvas.Font);
            RotatePrintFont;
            // Write out the address lines to the printer's Canvas
            Printer.Canvas.TextOut(Left, Bottom, edtName.Text);
            Printer.Canvas.TextOut(Left+TempHeight+2, Bottom,
➥edtStreet.Text);
            Printer.Canvas.TextOut(Left+TempHeight*2+2, Bottom,
➥edtCityState.Text);
            // Restore the original font
            Printer.Canvas.Font.Assign(SaveFont);
          finally
            SaveFont.Free;
          end;
        end
        else begin
          { If the envelope is not printed vertically, then
            just draw the address lines normally. }
          Printer.Canvas.TextOut(Left, Top, edtName.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight+2, edtStreet.Text);
          Printer.Canvas.TextOut(Left, Top+TempHeight*2+2,
➥edtCityState.Text);
        end;
      end;
    finally
      Printer.EndDoc;
    end;
  end;
end;

procedure TMainForm.RotatePrintFont;
var
  LogFont: TLogFont;
begin
  with Printer.Canvas do
  begin
    with LogFont do
    begin
      lfHeight := Font.Height; // Set to Printer.Canvas.font.height
      lfWidth := 0;            // let font mapper choose width
```

```
      lfEscapement := 900;        // tenths of degrees so 900 = 90 degrees
      lfOrientation := lfEscapement; // Always set to value of lfEscapement
      lfWeight := FW_NORMAL;   // default
      lfItalic := 0;           // no italics
      lfUnderline := 0;        // no underline
      lfStrikeOut := 0;        // no strikeout
      lfCharSet := ANSI_CHARSET;  //default
      StrPCopy(lfFaceName, Font.Name); // Printer.Canvas's font's name
      lfQuality := PROOF_QUALITY;
      lfOutPrecision := OUT_TT_ONLY_PRECIS;   // force TrueType fonts
      lfClipPrecision := CLIP_DEFAULT_PRECIS; // default
      lfPitchAndFamily := Variable_Pitch;     // default
    end;
  end;
  Printer.Canvas.Font.Handle := CreateFontIndirect(LogFont);
end;

end.
```

When the user clicks one of the radio buttons in `gbEnvelopeSize` or `gbFeedType`, the `FeedTypeClick()` event handler is called. This event handler calls the routines to calculate the envelope's size and position based on the radio button choices.

The address rectangle's size and position also are calculated in these event handlers. This rectangle's width is based on the longest text width of the text in each of the three `TEdit` components. The rectangle's height consists of the combined height of the three `TEdit` components.

All calculations are based on `Printer.Canvas`'s pixels. `mmiPrintItClick()` contains logic to print the envelope based on the choices selected. Additional logic to handle font rotation when the envelope is positioned vertically is also provided. Additionally, a pseudo–print preview is created in the `FormCreate()` event handler. This print preview is updated as the user selects the radio buttons.

The `TFeedType` enumerated type represents each position of the envelope as it may feed out of the printer:

```
TFeedType = (epLHorz, epLVert, epRHorz, epRVert);
```

`TMainForm` contains variables to hold the envelope's size and position, the address's `TRect` size and position, and the current `TFeedType`.

`TMainForm` declares the methods `GetEnvelopeSize()`, `GetEnvelopePos()`, `GetToAddrSize()`, and `GetToAddrPos()` to determine the various measurements for elements to be printed, as specified in steps 6 and 7 of this chapter's model.

In `GetEnvelopeSize()`, the `GetDeviceCaps()` function is used to convert the envelope size in inches to pixels, based on the selection from `gbEnvelopeSize`. `GetEnvelopPos()` determines the position of the envelope on `TPrinter.Canvas`, based on `Printer.Canvas`'s coordinate system.

`GetToAddrSize()` calculates the size of the address's bounding rectangle, based on the measurements of text contained in the three `TEdit` components. Here, `Printer.Canvas`'s `TextHeight()` and `TextWidth()` methods are used to determine these sizes. The function `MaxLn()` is a helper function used to determine the longest text line of the three `TEdit` components, which is used as the rectangle's width. You can also use the `Max()` function from the `Math.pas` unit to determine the longest text line.

`GetToAddrPos()` calls `GetToAddrSize()` and uses the returned value to calculate the address's bounding rectangle's position on `Printer.Canvas`. Note that the envelope's size and placement are needed for this function to position the address rectangle properly.

The `mmiPrintItClick()` event handler performs the actual printing logic. First, it initializes printing with the `BeginDoc()` method. Then it calculates a temporary line height used for text positioning. It determines the `TFeedType`, and if it's one of the vertical types, saves the printer's font and calls the method `RotatePrintFont()`, which rotates the font 90 degrees. When it returns form `RotatePrintFont()`, it restores `Printer.Canvas`'s original font. If the `TFeedType` is one of the horizontal types, it performs the `TextOut()` calls to print the address. Finally, `mmiPrintItClick()` ends printing with the `EndDoc()` method.

`RotatePrintFont()` creates a `TLogFont` structure and initializes its various values obtained from `Printer.Canvas` and other default values. Notice the assignment to its `lfEscapement` member. Remember from Chapter 8, "Graphics Programming with GDI and Fonts," that `lfEscapement` specifies an angle in tenths of degrees at which the font is to be drawn. Here, you specify to print the font at a 90-degree angle by assigning `900` to `lfEscapement`. One thing to note here is that only TrueType fonts can be rotated.

A Simple Print Preview

Often, a good way to help your users not make a mistake by choosing the wrong selection is to enable them to view what the printed output would look like before actually printing. The project in this section contains a print preview panel. You did this by constructing a descendant class of `TPanel` and publicizing its `Canvas` property:

```
TPrintPrevPanel = class(TPanel)
public
  property Canvas; // Publicize this property
end;
```

The `FormCreate()` event handler performs the logic to instantiate a `TPrintPrevPanel`. The following line determines the ratio of the printer's width to its height:

```
Ratio := Printer.PageHeight / Printer.PageWidth;
```

This ratio is used to calculate the width and height for the `TPrintPrevPanel` instance.

Before the `TPrintPrevPanel` is created, however, a regular `TPanel` with a black color is created to serve as a shadow to the `TPrintPrevPanel` instance, `PrintPrev`. Its boundaries are adjusted so that they're slightly to the right of and below the `PrintPrev`'s boundaries. The effect is that it gives `PrintPrev` a three-dimensional look with a shadow behind it. `PrintPrev` is used primarily to show how the envelope would be printed. The routine `DrawIt()` performs this logic.

`TEnvPrintForm.DrawIt()` calls `PrintPrev.Invalidate` to erase its previous contents. Then it calls `PrintPrev.Update()` to ensure that the paint message is processed before executing the remaining code. It then sets `PrintPrev`'s mapping mode to `MM_ISOTROPIC` to allow it to accept arbitrary extents along the X- and Y-axes. `SetWindowExt()` sets `PrintPrev`'s windows' extents to those of `Printer.Canvas`, and `SetViewPortExt()` sets `PrintPrev`'s viewport extents to its own height and width (see Chapter 8, "Graphics Programming with GDI and Fonts," for a discussion on mapping modes).

This enables `DrawIt()` to use the same metric values used for the `Printer.Canvas`, the envelope, the address rectangle, and the `PrintPrev` panel. This routine also uses rectangles to represent text lines. The effect is shown in Figure 10.7.

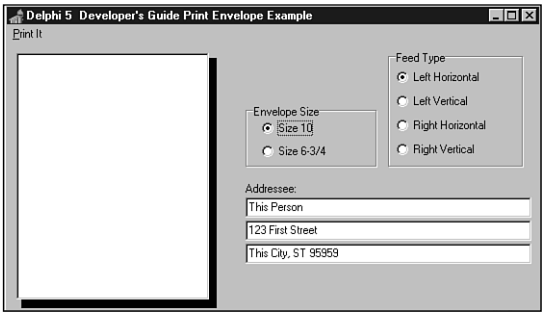

FIGURE 10.7
An envelope-printing form with a print preview feature.

NOTE

An alternative and better print preview can be created with metafiles. Create the metafile using the printer handle as the reference device, then draw into the metafile canvas just as you would the printer canvas, and then draw the metafile on the screen. No scaling or viewport extent tweaking is required.

Miscellaneous Printing Tasks

Occasionally, you'll need to perform a printing task that isn't available through the `TPrinter` object, such specifying the print quality of your print job. To perform these tasks, you must resort to the Win32 API method. However, this isn't too difficult. First, you must understand the `TDeviceMode` structure. The next section discusses this. The following sections show you how to use this structure to perform these various printing tasks.

The TDeviceMode Structure

The `TDeviceMode` structure contains information about a printer driver's initialization and environment data. Programmers use this structure to retrieve information about or set various attributes of the current printer. This structure is defined in the Windows.pas file.

You'll find definitions for each of the fields in Delphi's online help. The following sections cover some of the more common fields of this structure, but it would be a good idea to take a look at the online help and read what some of the other fields are used for. In some cases, you might need to refer to these fields, and some of them are used differently in Windows NT/2000 than in Windows 95/98.

To obtain a pointer to the current printer's `TDeviceMode` structure, you can first use `TPrinter.GetPrinter()` to obtain a handle to the memory block that the structure occupies. Then use the `GlobalLock()` function to retrieve a pointer to this structure. Listing 10.5 illustrates how to get the pointer to the `TDeviceMode` structure.

LISTING 10.5 Obtaining a Pointer to a `TDeviceMode` Structure

```
var
  ADevice, ADriver, APort: array [0..255] of Char;
  DeviceHandle: THandle;
  DevMode: PDeviceMode; // A Pointer to a TDeviceMode structure
begin
  { First obtain a handle to the TPrinter's DeviceMode structure }
  Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  { If DeviceHandle is still 0, then the driver was not loaded. Set
    the printer index to force the printer driver to load making the
    handle available }
  if DeviceHandle = 0 then
  begin
    Printer.PrinterIndex := Printer.PrinterIndex;
    Printer.GetPrinter(ADevice, ADriver, APort, DeviceHandle);
  end;
  { If DeviceHandle is still 0, then an error has occurred. Otherwise,
    use GlobalLock() to get a pointer to the TDeviceMode structure }
```

```
  if DeviceHandle = 0 then
    Raise Exception.Create('Could Not Initialize TDeviceMode structure')
  else
    DevMode := GlobalLock(DeviceHandle);
  { Code to use the DevMode structure goes here }
  { !!!! }
  if not DeviceHandle = 0 then
    GlobalUnlock(DeviceHandle);
end;
```

The comments in the preceding listing explain the steps required to obtain the pointer to the `TDeviceMode` structure. After you've obtained this pointer, you can perform various printer routines, as illustrated in the following sections. First, however, notice this comment in the preceding listing:

```
{ Code to use the DevMode structure goes here }
  { !!!! }
```

It's here that you place the code examples to follow.

Before you can initialize any of the members of the `TDeviceMode` structure, however, you must specify which member you're initializing by setting the appropriate bit in the `dmFields` bit flags. Table 10.3 lists the various bit flags of `dmFields` and also specifies to which `TDeviceMode` member they pertain.

TABLE 10.3 `TDeviceMode.dmFields` Bit Flags

dmField Value	*Corresponding Field*
`DM_ORIENTATION`	`dmOrientation`
`DM_PAPERSIZE`	`dmPaperSize`
`DM_PAPERLENGTH`	`dmPaperLength`
`DM_PAPERWIDTH`	`dmPaperWidth`
`DM_SCALE`	`dmScale`
`DM_COPIES`	`dmCopies`
`DM_DEFAULTSOURCE`	`dmDefaultSource`
`DM_PRINTQUALITY`	`dmPrintQuality`
`DM_COLOR`	`dmColor`
`DM_DUPLEX`	`dmDuplex`
`DM_YRESOLUTION`	`dmYResolution`
`DM_TTOPTION`	`dmTTOption`
`DM_COLLATE`	`dmCollate`

continues

TABLE 10.3 Continued

dmField Value	*Corresponding Field*
DM_FORMNAME	dmFormName
DM_LOGPIXELS	dmLogPixels
DM_BITSPERPEL	dmBitsPerPel
DM_PELSWIDTH	dmPelsWidth
DM_PELSHEIGHT	dmPelsHeight
DM_DISPLAYFLAGS	dmDisplayFlags
DM_DISPLAYFREQUENCY	dmDisplayFrequency
DM_ICMMETHOD	dmICMMethod (Windows 95 only)
DM_ICMINTENT	dmICMIntent (Windows 95 only)
DM_MEDIATYPE	dmMediaType (Windows 95 only)
DM_DITHERTYPE	dmDitherType (Windows 95 only)

In the examples that follow, you'll see how to set the appropriate bit flag as well as the corresponding TDeviceMode member.

Specifying Copies to Print

You can tell a print job how many copies to print by specifying the number of copies in the dmCopies field of the TDeviceMode structure. The following code illustrates how to do this:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_COPIES;
   dmCopies := Copies;
end;
```

First, you must set the appropriate bit flag of the dmFields field to indicate which member of the TDeviceMode structure has been initialized. The preceding code is what you would insert into the code in Listing 10.6 where specified. Then, whenever you start your print job, the number of copies specified should be sent to the printer. It's worth mentioning that although this examples illustrates how to set the copies to print using the TDeviceMode structure, the TPrinter.Copies property does the same.

Specifying Printer Orientation

Specifying printer orientation is similar to specifying copies except that you initialize a different TDeviceMode structure:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_ORIENTATION;
   dmOrientation := DMORIENT_LANDSCAPE;
end;
```

The two options for `dmOrientation` are `DMORIENT_LANDSCAPE` and `DMORIENT_PORTRAIT`. You might also look at the `TPrinter.Orientation` property.

Specifying Paper Size

To specify a paper size, you initialize `TDeviceMode`'s `dmPaperSize` member:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERSIZE;
   dmPaperSize := DMPAPER_LETTER;  // Letter, 8-1/2 by 11 inches
end;
```

Several predefined values exist for the `dmPaperSize` member, which you can look up in the online help under `TDeviceMode`. The `dmPaperSize` member can be set to zero if the paper size is specified by the `dmPaperWidth` and `dmPaperHeight` members.

Specifying Paper Length

You can specify the paper length in tenths of a millimeter for the printed output by setting the `dmPaperLength` field. This overrides any settings applied to the `dmPaperSize` field. The following code illustrates setting the paper length:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERLENGTH;
   dmPaperLength := SomeLength;
end;
```

Specifying Paper Width

Paper width is also specified in tenths of a millimeter. To set the paper width, you must initialize the `dmPaperWidth` field of the `TDeviceMode` structure. The following code illustrates this setting:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PAPERWIDTH;
   dmPaperWidth := SomeWidth;
end;
```

This also overrides the settings for the `dmPaperSize` field.

Specifying Print Scale

The *print scale* is the factor by which the printed output is scaled. Therefore, the resulting page size is scaled from the physical page size by a factor of `TDeviceMode.dmScale` divided by 100. Therefore, to shrink the printed output (graphics and text) by half their original size, you would assign the value of 50 to the `dmScale` field. The following code illustrates how to set the print scale:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_SCALE;
   dmScale := 50;
end;
```

Specifying Print Color

For printers that support color printing, you can specify whether the printer is to render color or monochrome printing by initializing the `dmColor` field, as shown here:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_COLOR;
   dmColor := DMCOLOR_COLOR;
end;
```

Another value that can be assigned to the `dmColor` field is `DMCOLOR_MONOCHROME`.

Specifying Print Quality

Print quality is the resolution at which the printer prints its output. Four predefined values exist for setting the print quality, as shown in the following list:

- `DMRES_HIGH`. High-resolution printing
- `DMRES_MEDIUM`. Medium-resolution printing
- `DMRES_LOW`. Low-resolution printing
- `DMRES_DRAFT`. Draft-resolution printing

To change the quality of print, you initialize the `dmPrintQuality` field of the `TDeviceMode` structure:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_PRINTQUALITY;
   dmPrintQuality := DMRES_DRAFT;
end;
```

Specifying Duplex Printing

Some printers are capable of duplex printing—printing on both sides of the paper. You can tell the printer to perform double-sided printing by initializing the `dmDuplex` field of the `TDeviceMode` structure to one of these values:

- `DMDUP_SIMPLEX`
- `DMDUP_HORIZONTAL`
- `DMDUP_VERTICAL`

Here's an example:

```
with DevMode^ do
begin
   dmFields := dmFields or DM_DUPLEX;
   dmDuplex := DMDUP_HORIZONTAL;
end;
```

Changing the Default Printer

Although it's possible to change the default printer by launching the printer folder, you might want to change the default printer at runtime. This is possible as illustrated in the sample project shown in Listing 10.6.

LISTING 10.6 Changing the Default Printer

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    cbPrinters: TComboBox;
    lblPrinter: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

continues

LISTING 10.6 Continued

```
var
  MainForm: TMainForm;

implementation
uses IniFiles, Printers;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Copy the printer names to the combobox and set the combobox to
    show the currently selected default printer }
  cbPrinters.Items.Assign(Printer.Printers);
  cbPrinters.Text := Printer.Printers[Printer.PrinterIndex];
  // Update the label to reflect the default printer
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
var
  IniFile: TIniFile;
  TempStr1, TempStr2: String;
begin
  with Printer do
  begin
    // Set the new printer based on the ComboBox's selected printer
    PrinterIndex := cbPrinters.ItemIndex;
    // Store the printer name into a temporary string
    TempStr1 := Printers[PrinterIndex];
    // Delete the unnecessary portion of the printer name
    System.Delete(TempStr1, Pos(' on ', TempStr1), Length(TempStr1));
    // Create a TIniFile class
    IniFile := TIniFile.Create('WIN.INI');
    try
      // Retrieve the device name of the selected printer
      TempStr2 := IniFile.ReadString('Devices', TempStr1, '');
      // Change the default printer to that chosen by the user
      IniFile.WriteString('windows', 'device', TempStr1 + ',' + TempStr2);
    finally
      IniFile.Free;
    end;
  end;
  // Update the label to reflect the new printer selection
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

end.
```

The preceding project consists of a main form with a `TComboBox` and a `TLabel` component. Upon form creation, the `TComboBox` component is initialized with the string list of printer names obtained from the `Printer.Printers` property. The `TLabel` component is then updated to reflect the currently selected printer. The `cbPrintersChange()` event handler is where we placed the code to modify the system-wide default printer. What this entails is changing the `[device]` entry in the `[windows]` section of the `WIN.INI` file, located in the `Windows` directory. The comments in the preceding code go on to explain the process of making these modifications.

Obtaining Printer Information

This section illustrates how you can retrieve information about a printer device such as physical characteristics (number of bins, paper sizes supported, and so on) as well as the printer's text- and graphics-drawing capabilities.

You might want to get information about a particular printer for several reasons. For example, you might need to know whether the printer supports a particular capability. A typical example is to determine whether the current printer supports banding. *Banding* is a process that can improve printing speed and disk space requirements for printers with memory limitations. To use banding, you must make API calls specific to this capability. On a printer that doesn't support this capability, these calls wouldn't function. Therefore, you can first determine whether the printer will support banding (and use it, if so); otherwise, you can avoid the banding API calls.

GetDeviceCaps() and DeviceCapabilities()

The Win32 API function `GetDeviceCaps()` allows you to obtain information about devices such as printers, plotters, screens, and so on. Generally, these are devices that have a device context. You use `GetDeviceCaps()` by supplying it a handle to a device context and an index that specifies the information you want to retrieve.

`DeviceCapabilities()` is specific to printers. In fact, the information obtained from `DeviceCapabilities()` is provided by the printer driver for a specified printer. Use `DeviceCapabilities()` by supplying it with strings identifying the printer device as well as an index specifying the data you want to retrieve. Sometimes two calls to `DeviceCapabilities()` are required to retrieve certain data. The first call is made to determine how much memory you must allocate for the data to be retrieved. The second call stores the data in the memory block you've allocated. This section illustrates how to do this.

One thing you should know is that most of the drawing capabilities that aren't supported by a particular printer will still work if you use them. For example, when `GetDeviceCaps()` or `DeviceCapabilities()` indicates that `BitBlt()`, `StretchBlt()`, or printing TrueType fonts isn't supported, you can still use any of these functions; GDI will simulate these functions for you. Note, however, that GDI cannot simulate `BitBlt()` on a device that doesn't support raster scanline pixels; `BitBlt()` will always fail on a pen plotter, for example.

Printer Information Sample Program

Figure 10.8 shows the main form for the sample program. This program contains eight pages, each of which lists different printer capabilities for the printer selected in the combo box.

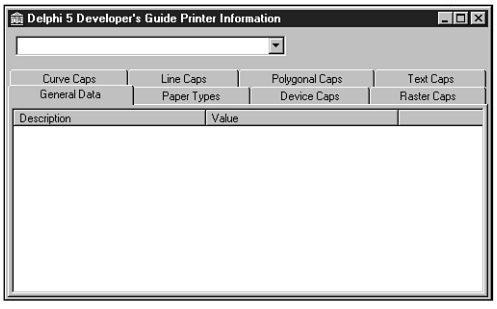

FIGURE 10.8
The main form for the printer information example.

Declaring the DeviceCapabilitiesA Function

If you attempt to use the function `DeviceCapabilities()` defined in `Windows.pas`, you won't be able to run your program because this function isn't defined in `GDI32.DLL` as `Windows.pas` indicates. Instead, this function in `GDI32.DLL` is `DeviceCapabilitiesEx()`. However, even if you define this function's prototype as follows, the function won't work as expected and returns erroneous results:

```
function DeviceCapabilitiesEx(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'Gdi32.dll';
```

It turns out that two functions—`DeviceCapabilitiesA()` for ANSI strings and `DeviceCapabilitiesW()` for wide strings—are defined in `WINSPOOL.DRV`, which is the Win32 print spooler interface. This function is the correct one to use as indicated in the Microsoft Developer's Network CD (MSDN). The correct definition for the function prototype that's used in the sample program in Listing 10.8 (shown in the following section) is as follows:

```
function DeviceCapabilitiesA(pDevice, pPort: Pchar; fwCapability: Word;
         pOutput: Pchar; DevMode: PdeviceMode):
         Integer; stdcall; external 'winspool.drv';
```

Note that the preceding declaration can be found in `WINSPOOL.PAS` in Delphi 5.

Sample Program Functionality

Listing 10.8 (shown at the end of this section) contains the source for the Printer Information sample program. The main form's `OnCreate` event handler simply populates the combo box

with the list of available printers on the system. The `OnChange` event handler for the combo box is the central point of the application where the methods to retrieve the printer information are called.

The first page on the form General Data contains general information about the printer device. You'll see that the printer's device name, driver, and port location are obtained by calling the `TPrinter.GetPrinter()` method. This method also retrieves a handle to a `TDeviceMode` structure for the currently selected printer. This information is then added to the General Data page. To retrieve the printer driver version, you use the `DeviceCapabilitiesA()` function and pass the `DC_DRIVER` index. The rest of the `PrinterComboChange` event handler calls the various routines to populate the list boxes on the various pages of the main form.

The `GetBinNames()` method illustrates how to use the `DeviceCapabilitiesA()` function to retrieve the bin names for the selected printer. This method first gets the number of bin names available by calling `DeviceCapabilitiesA()`, passing the `DC_BINNAMES` index, and passing `nil` as the `pOutput` and `DevMode` parameters. The result of this function call specifies how much memory must be allocated to hold the bin names. According to the documentation on `DeviceCapabilitiesA()`, each bin name is defined as an array of 24 characters. We defined a `TBinName` data type like this:

```
TBinName = array[0..23] of char;
```

We also defined an array of `TBinName`:

```
TBinNames = array[0..0] of TBinName;
```

This type is used to typecast a pointer as an array of `TBinName data types`. To access an element at some index into the array, you must disable range checking, because this array is defined to have a range of `0..0`, as illustrated in the `GetBinNames()` method. The bin names are added to the appropriate list box.

This same technique of determining the amount of memory required and allocating this memory dynamically is also used in the methods `GetDevCapsPaperNames()` and `GetResolutions()`.

The methods `GetDuplexSupport()`, `GetCopies()`, and `GetEMFStatus()` all use the `DeviceCapabilitiesA()` function to return a value of the requested information. For example, the following code determines whether the selected printer supports duplex printing by returning a value of `1` if duplex printing is supported or `0` if not:

```
DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
```

Also, the following statement returns the maximum number of copies the device can print:

```
DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
```

The remaining methods use the `GetDeviceCaps()` function to determine the various capabilities of the selected device. In some cases, `GetDeviceCaps()` returns the specific value requested. For example, the following statement returns the width, in millimeters, of the printer device:

```
GetDeviceCaps(Printer.Handle, HORZSIZE);
```

In other cases, `GetDeviceCaps()` returns an integer value whose bits are masked to determine a particular capability. For example, the `GetRasterCaps()` method first retrieves the integer value that contains the bitmasked fields:

```
RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
```

Then, to determine whether the printer supports banding, you must mask out the `RC_BANDING` field by performing an `AND` operation whose result should equal the value of `RC_BANDING`:

```
(RCaps and RC_BANDING) = RC_BANDING
```

This evaluation is passed to one of the helper functions, `BoolToYesNoStr()`, which returns the string `Yes` or `No`, based on the result of the evaluation. Other fields are masked in the same manner. This same technique is used in other areas where bitmasked fields are returned from `GetDeviceCaps()` as well as from the `DeviceCapabilitiesA()` function, such as in the `GetTrueTypeInfo()` method.

You'll find both functions, `DeviceCapabilties()` and `GetDeviceCaps()`, well documented in the online Win32 API help.

LISTING 10.7 Printer Information Sample Program

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    pgcPrinterInfo: TPageControl;
    tbsPaperTypes: TTabSheet;
    tbsGeneralData: TTabSheet;
    lbPaperTypes: TListBox;
    tbsDeviceCaps: TTabSheet;
    tbsRasterCaps: TTabSheet;
    tbsCurveCaps: TTabSheet;
    tbsLineCaps: TTabSheet;
    tbsPolygonalCaps: TTabSheet;
```

```
    tbsTextCaps: TTabSheet;
    lvGeneralData: TListView;
    lvCurveCaps: TListView;
    Splitter1: TSplitter;
    lvDeviceCaps: TListView;
    lvRasterCaps: TListView;
    pnlTop: TPanel;
    cbPrinters: TComboBox;
    lvLineCaps: TListView;
    lvPolyCaps: TListView;
    lvTextCaps: TListView;
    procedure FormCreate(Sender: TObject);
    procedure cbPrintersChange(Sender: TObject);
  private
    Device, Driver, Port: array[0..255] of char;
    ADevMode: THandle;
   public
    procedure GetBinNames;
    procedure GetDuplexSupport;
    procedure GetCopies;
    procedure GetEMFStatus;
    procedure GetResolutions;
    procedure GetTrueTypeInfo;
    procedure GetDevCapsPaperNames;
    procedure GetDevCaps;
    procedure GetRasterCaps;
    procedure GetCurveCaps;
    procedure GetLineCaps;
    procedure GetPolyCaps;
    procedure GetTextCaps;
  end;

var
  MainForm: TMainForm;

implementation
uses
  Printers, WinSpool;

const
  NoYesArray: array[Boolean] of String = ('No', 'Yes');
type

  // Types for holding bin names
  TBinName = array[0..23] of char;
  // Where used set $R- to prevent error
```

continues

LISTING 10.7 Continued

```
  TBinNames = array[0..0] of TBinName;

  // Types for holding paper names
  TPName = array[0..63] of char;

  // Where used set $R- to prevent error
  TPNames = array[0..0] of TPName;

  // Types for holding resolutions
  TResolution = array[0..1] of integer;
  // Where used set $R- to prevent error
  TResolutions = array[0..0] of TResolution;

  // Type for holding array of pages sizes (word types)
  TPageSizeArray = Array[0..0] of word;

var
 Rslt: Integer;

{$R *.DFM}
(*
function BoolToYesNoStr(aVal: Boolean): String;
// Returns the string "YES" or "NO" based on the boolean value
begin
  if aVal then
    Result := 'Yes'
  else
    Result := 'No';
end;
*)
procedure AddListViewItem(const aCaption, aValue: String; aLV: TListView);
// This method is used to add a TListItem to the TListView, aLV
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.SubItems.Add(aValue);
end;

procedure TMainForm.GetBinNames;
var
  BinNames: Pointer;
  i: integer;
```

```
begin
{$R-} // Range checking must be turned off here.
  // First determine how many bin names are available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_BINNAMES, nil, nil);
  if Rslt > 0 then
  begin
    { Each bin name is 24 bytes long. Therefore, allocate Rslt*24 bytes to hold
      the bin names. }
    GetMem(BinNames, Rslt*24);
    try
      // Now retrieve the bin names in the allocated block of memory.
      if DeviceCapabilitiesA(Device, Port, DC_BINNAMES, BinNames, nil) = -1
then
        raise Exception.Create('DevCap Error');
      //{ Add the information to the appropriate list box.
      AddListViewItem('BIN NAMES', EmptyStr, lvGeneralData);
      for i := 0 to Rslt - 1 do
      begin
        AddListViewItem(Format('   Bin Name %d', [i]),
          StrPas(TBinNames(BinNames^)[i]), lvGeneralData);
      end;
    finally
      FreeMem(BinNames, Rslt*24);
    end;
  end;
{$R+} // Turn range checking back on.
end;

procedure TMainForm.GetDuplexSupport;
begin
  { This function uses DeviceCapabilitiesA to determine whether or not the
    printer device supports duplex printing. }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_DUPLEX, nil, nil);
  AddListViewItem('Duplex Printing', NoYesArray[Rslt = 1], lvGeneralData);
end;

procedure TMainForm.GetCopies;
begin
  { This function determines how many copies the device can be set to print.
    If the result is not greater than 1 then the print logic must be
    executed multiple times }
  Rslt := DeviceCapabilitiesA(Device, Port, DC_COPIES, nil, nil);
  AddListViewItem('Copies that printer can print',
     InttoStr(Rslt), lvGeneralData);

end;
```

continues

LISTING 10.7 Continued

```
procedure TMainForm.GetEMFStatus;
begin
  // This function determines if the device supports the enhanced metafiles.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_EMF_COMPLIANT, nil, nil);
  AddListViewItem('EMF Compliant', NoYesArray[Rslt=1], lvGeneralData);
end;

procedure TMainForm.GetResolutions;
var
  Resolutions: Pointer;
  i: integer;
begin
{$R-}  // Range checking must be turned off.
  // Determine how many resolutions are available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS, nil, nil);
  if Rslt > 0 then begin
    { Allocate the memory to hold the different resolutions which are
      represented by integer pairs, ie: 300, 300 }
    GetMem(Resolutions, (SizeOf(Integer)*2)*Rslt);
    try
      // Retrieve the different resolutions.
      if DeviceCapabilitiesA(Device, Port, DC_ENUMRESOLUTIONS,
      Resolutions, nil) = -1 then
        Raise Exception.Create('DevCaps Error');
      // Add the resolution information to the appropriate list box.
      AddListViewItem('RESOLUTION CONFIGURATIONS', EmptyStr, lvGeneralData);

      for i := 0 to Rslt - 1 do
      begin
        AddListViewItem('   Resolution Configuration',
          IntToStr(TResolutions(Resolutions^)[i][0])+
          ' '+IntToStr(TResolutions(Resolutions^)[i][1]), lvGeneralData);
      end;
    finally
      FreeMem(Resolutions, SizeOf(Integer)*Rslt*2);
    end;
  end;
{$R+}  // Turn range checking back on.
end;

procedure TMainForm.GetTrueTypeInfo;
begin
  // Get the TrueType font capabilities of the device represented as bitmasks
  Rslt := DeviceCapabilitiesA(Device, Port, DC_TRUETYPE, nil, nil);
  if Rslt <> 0 then
```

```
    { Now mask out the individual TrueType capabilities and indicate the
      result in the appropriate list box. }
    AddListViewItem('TRUE TYPE FONTS', EmptyStr, lvGeneralData);
    with lvGeneralData.Items do
    begin
      AddListViewItem('   Prints TrueType fonts as graphics',
        NoYesArray[(Rslt and DCTT_BITMAP) = DCTT_BITMAP], lvGeneralData);

      AddListViewItem('   Downloads TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD) = DCTT_DOWNLOAD],
➥lvGeneralData);

      AddListViewItem('   Downloads outline TrueType fonts',
        NoYesArray[(Rslt and DCTT_DOWNLOAD_OUTLINE) =
➥DCTT_DOWNLOAD_OUTLINE],
        lvGeneralData);

      AddListViewItem('   Substitutes device for TrueType fonts',
        NoYesArray[(Rslt and DCTT_SUBDEV) = DCTT_SUBDEV], lvGeneralData);
    end;
end;

procedure TMainForm.GetDevCapsPaperNames;
{ This method gets the paper types available on a selected printer from the
  DeviceCapabilitiesA function. }
var
  PaperNames: Pointer;
  i: integer;
begin
{$R-}  // Range checking off.
  lbPaperTypes.Items.Clear;
  // First get the number of paper names available.
  Rslt := DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES, nil, nil);
  if Rslt > 0 then begin
    { Now allocate the array of paper names. Each paper name is 64 bytes.
      Therefore, allocate Rslt*64 of memory. }
    GetMem(PaperNames, Rslt*64);
    try
      // Retrieve the list of names into the allocated memory block.
      if DeviceCapabilitiesA(Device, Port, DC_PAPERNAMES,
      PaperNames, nil) = - 1 then
        raise Exception.Create('DevCap Error');
      // Add the paper names to the appropriate list box.
      for i := 0 to Rslt - 1 do
       lbPaperTypes.Items.Add(StrPas(TPNames(PaperNames^)[i]));
    finally
```

continues

LISTING 10.7 Continued

```
      FreeMem(PaperNames, Rslt*64);
    end;
  end;
{$R+} // Range checking back on.
end;

procedure TMainForm.GetDevCaps;
{ This method retrieves various capabilities of the selected printer device by
  using the GetDeviceCaps function. Refer to the Online API help for the
  meaning of each of these items. }
begin
  with lvDeviceCaps.Items do
  begin
    Clear;
    AddListViewItem('Width in millimeters',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZSIZE)), lvDeviceCaps);
    AddListViewItem('Height in millimeter',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTSIZE)), lvDeviceCaps);
    AddListViewItem('Width in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, HORZRES)), lvDeviceCaps);
    AddListViewItem('Height in pixels',
      IntToStr(GetDeviceCaps(Printer.Handle, VERTRES)), lvDeviceCaps);
    AddListViewItem('Pixels per horizontal inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSX)), lvDeviceCaps);
    AddListViewItem('Pixels per vertical inch',
      IntToStr(GetDeviceCaps(Printer.Handle, LOGPIXELSY)), lvDeviceCaps);
    AddListViewItem('Color bits per pixel',
      IntToStr(GetDeviceCaps(Printer.Handle, BITSPIXEL)), lvDeviceCaps);
    AddListViewItem('Number of color planes',
      IntToStr(GetDeviceCaps(Printer.Handle, PLANES)), lvDeviceCaps);
    AddListViewItem('Number of brushes',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMBRUSHES)), lvDeviceCaps);
    AddListViewItem('Number of pens',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMPENS)), lvDeviceCaps);
    AddListViewItem('Number of fonts',
      IntToStr(GetDeviceCaps(Printer.Handle, NUMFONTS)), lvDeviceCaps);
    Rslt := GetDeviceCaps(Printer.Handle, NUMCOLORS);
    if Rslt = -1 then
      AddListViewItem('Number of entries in color table', ' > 8', lvDeviceCaps)
    else AddListViewItem('Number of entries in color table',
        IntToStr(Rslt), lvDeviceCaps);
    AddListViewItem('Relative pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTX)), lvDeviceCaps);
    AddListViewItem('Relative pixel drawing height',
```

```
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTY)), lvDeviceCaps);
    AddListViewItem('Diagonal pixel drawing width',
      IntToStr(GetDeviceCaps(Printer.Handle, ASPECTXY)), lvDeviceCaps);
    if GetDeviceCaps(Printer.Handle, CLIPCAPS) = 1 then
      AddListViewItem('Clip to rectangle', 'Yes', lvDeviceCaps)
    else AddListViewItem('Clip to rectangle', 'No', lvDeviceCaps);
  end;
end;

procedure TMainForm.GetRasterCaps;
{ This method gets the various raster capabilities of the selected printer
  device by using the GetDeviceCaps function with the RASTERCAPS index. Refer
  to the online help for information on each capability. }
var
  RCaps: Integer;
begin
  with lvRasterCaps.Items do
  begin
    Clear;
    RCaps := GetDeviceCaps(Printer.Handle, RASTERCAPS);
    AddListViewItem('Banding',
      NoYesArray[(RCaps and RC_BANDING) = RC_BANDING], lvRasterCaps);
    AddListViewItem('BitBlt Capable',
      NoYesArray[(RCaps and RC_BITBLT) = RC_BITBLT], lvRasterCaps);
    AddListViewItem('Supports bitmaps > 64K',
      NoYesArray[(RCaps and RC_BITMAP64) = RC_BITMAP64], lvRasterCaps);
    AddListViewItem('DIB support',
      NoYesArray[(RCaps and RC_DI_BITMAP) = RC_DI_BITMAP], lvRasterCaps);
    AddListViewItem('Floodfill support',
      NoYesArray[(RCaps and RC_FLOODFILL) = RC_FLOODFILL], lvRasterCaps);
    AddListViewItem('Windows 2.0 support',
      NoYesArray[(RCaps and RC_GDI20_OUTPUT) = RC_GDI20_OUTPUT],
➥lvRasterCaps);
    AddListViewItem('Palette based device',
      NoYesArray[(RCaps and RC_PALETTE) = RC_PALETTE], lvRasterCaps);
    AddListViewItem('Scaling support',
      NoYesArray[(RCaps and RC_SCALING) = RC_SCALING], lvRasterCaps);
    AddListViewItem('StretchBlt support',
      NoYesArray[(RCaps and RC_STRETCHBLT) = RC_STRETCHBLT],
➥lvRasterCaps);
    AddListViewItem('StretchDIBits support',
      NoYesArray[(RCaps and RC_STRETCHDIB) = RC_STRETCHDIB],
➥lvRasterCaps);
  end;
end;
```

continues

LISTING 10.7 Continued

```
procedure TMainForm.GetCurveCaps;
{ This method gets the various curve capabilities of the selected printer
  device by using the GetDeviceCaps function with the CURVECAPS index. Refer
  to the online help for information on each capability. }
var
  CCaps: Integer;
begin
  with lvCurveCaps.Items do
  begin
    Clear;
    CCaps := GetDeviceCaps(Printer.Handle, CURVECAPS);

    AddListViewItem('Curve support',
      NoYesArray[(CCaps and CC_NONE) = CC_NONE], lvCurveCaps);

    AddListViewItem('Circle support',
      NoYesArray[(CCaps and CC_CIRCLES) = CC_CIRCLES], lvCurveCaps);

    AddListViewItem('Pie support',
      NoYesArray[(CCaps and CC_PIE) = CC_PIE], lvCurveCaps);

    AddListViewItem('Chord arc support',
      NoYesArray[(CCaps and CC_CHORD) = CC_CHORD], lvCurveCaps);

    AddListViewItem('Ellipse support',
      NoYesArray[(CCaps and CC_ELLIPSES) = CC_ELLIPSES], lvCurveCaps);

    AddListViewItem('Wide border support',
      NoYesArray[(CCaps and CC_WIDE) = CC_WIDE], lvCurveCaps);

    AddListViewItem('Styled border support',
      NoYesArray[(CCaps and CC_STYLED) = CC_STYLED], lvCurveCaps);

    AddListViewItem('Round rectangle support',
      NoYesArray[(CCaps and CC_ROUNDRECT) = CC_ROUNDRECT], lvCurveCaps);

  end;
end;

procedure TMainForm.GetLineCaps;
{ This method gets the various line drawing capabilities of the selected
printer device by using the GetDeviceCaps function with the LINECAPS index.
Refer to the online help for information on each capability. }
var
  LCaps: Integer;
```

```
begin
  with lvLineCaps.Items do
  begin
    Clear;
    LCaps := GetDeviceCaps(Printer.Handle, LINECAPS);

    AddListViewItem('Line support',
      NoYesArray[(LCaps and LC_NONE) = LC_NONE], lvLineCaps);

    AddListViewItem('Polyline support',
      NoYesArray[(LCaps and LC_POLYLINE) = LC_POLYLINE], lvLineCaps);

    AddListViewItem('Marker support',
      NoYesArray[(LCaps and LC_MARKER) = LC_MARKER], lvLineCaps);

    AddListViewItem('Multiple marker support',
      NoYesArray[(LCaps and LC_POLYMARKER) = LC_POLYMARKER], lvLineCaps);

    AddListViewItem('Wide line support',
      NoYesArray[(LCaps and LC_WIDE) = LC_WIDE], lvLineCaps);

    AddListViewItem('Styled line support',
      NoYesArray[(LCaps and LC_STYLED) = LC_STYLED], lvLineCaps);

    AddListViewItem('Wide and styled line support',
      NoYesArray[(LCaps and LC_WIDESTYLED) = LC_WIDESTYLED], lvLineCaps);

    AddListViewItem('Interior support',
      NoYesArray[(LCaps and LC_INTERIORS) = LC_INTERIORS], lvLineCaps);
  end;
end;

procedure TMainForm.GetPolyCaps;
{ This method gets the various polygonal capabilities of the selected printer
  device by using the GetDeviceCaps function with the POLYGONALCAPS index.
Refer to the online help for information on each capability. }
var
  PCaps: Integer;
begin
  with lvPolyCaps.Items do
  begin
    Clear;
    PCaps := GetDeviceCaps(Printer.Handle, POLYGONALCAPS);

    AddListViewItem('Polygon support',
      NoYesArray[(PCaps and PC_NONE) = PC_NONE], lvPolyCaps);
```

continues

LISTING 10.7 Continued

```
    AddListViewItem('Alternate fill polygon support',
      NoYesArray[(PCaps and PC_POLYGON) = PC_POLYGON], lvPolyCaps);

    AddListViewItem('Rectangle support',
      NoYesArray[(PCaps and PC_RECTANGLE) = PC_RECTANGLE], lvPolyCaps);

    AddListViewItem('Winding-fill polygon support',
      NoYesArray[(PCaps and PC_WINDPOLYGON) = PC_WINDPOLYGON], lvPolyCaps);

    AddListViewItem('Single scanline support',
      NoYesArray[(PCaps and PC_SCANLINE) = PC_SCANLINE], lvPolyCaps);

    AddListViewItem('Wide border support',
      NoYesArray[(PCaps and PC_WIDE) = PC_WIDE], lvPolyCaps);

    AddListViewItem('Styled border support',
      NoYesArray[(PCaps and PC_STYLED) = PC_STYLED], lvPolyCaps);

    AddListViewItem('Wide and styled border support',
      NoYesArray[(PCaps and PC_WIDESTYLED) = PC_WIDESTYLED], lvPolyCaps);

    AddListViewItem('Interior support',
      NoYesArray[(PCaps and PC_INTERIORS) = PC_INTERIORS], lvPolyCaps);
  end;
end;

procedure TMainForm.GetTextCaps;
{ This method gets the various text drawing capabilities of the selected
  printer device by using the GetDeviceCaps function with the TEXTCAPS index.
  Refer to the online help for information on each capability. }
var
  TCaps: Integer;
begin
  with lvTextCaps.Items do
  begin
    Clear;
    TCaps := GetDeviceCaps(Printer.Handle, TEXTCAPS);

    AddListViewItem('Character output precision',
      NoYesArray[(TCaps and TC_OP_CHARACTER) = TC_OP_CHARACTER], lvTextCaps);

    AddListViewItem('Stroke output precision',
      NoYesArray[(TCaps and TC_OP_STROKE) = TC_OP_STROKE], lvTextCaps);

    AddListViewItem('Stroke clip precision',
```

```
      NoYesArray[(TCaps and TC_CP_STROKE) = TC_CP_STROKE], lvTextCaps);

    AddListViewItem('90 degree character rotation',
      NoYesArray[(TCaps and TC_CR_90) = TC_CR_90], lvTextCaps);

    AddListViewItem('Any degree character rotation',
      NoYesArray[(TCaps and TC_CR_ANY) = TC_CR_ANY], lvTextCaps);

    AddListViewItem('Independent scale in X and Y direction',
      NoYesArray[(TCaps and TC_SF_X_YINDEP) = TC_SF_X_YINDEP], lvTextCaps);

    AddListViewItem('Doubled character for scaling',
      NoYesArray[(TCaps and TC_SA_DOUBLE) = TC_SA_DOUBLE], lvTextCaps);

    AddListViewItem('Integer multiples only for character scaling',
      NoYesArray[(TCaps and TC_SA_INTEGER) = TC_SA_INTEGER], lvTextCaps);

    AddListViewItem('Any multiples for exact character scaling',
      NoYesArray[(TCaps and TC_SA_CONTIN) = TC_SA_CONTIN], lvTextCaps);

    AddListViewItem('Double weight characters',
      NoYesArray[(TCaps and TC_EA_DOUBLE) = TC_EA_DOUBLE], lvTextCaps);

    AddListViewItem('Italicized characters',
      NoYesArray[(TCaps and TC_IA_ABLE) = TC_IA_ABLE], lvTextCaps);

    AddListViewItem('Underlined characters',
      NoYesArray[(TCaps and TC_UA_ABLE) = TC_UA_ABLE], lvTextCaps);

    AddListViewItem('Strikeout characters',
      NoYesArray[(TCaps and TC_SO_ABLE) = TC_SO_ABLE], lvTextCaps);

    AddListViewItem('Raster fonts',
      NoYesArray[(TCaps and TC_RA_ABLE) = TC_RA_ABLE], lvTextCaps);

    AddListViewItem('Vector fonts',
      NoYesArray[(TCaps and TC_VA_ABLE) = TC_VA_ABLE], lvTextCaps);

    AddListViewItem('Scrolling using bit-block transfer',
      NoYesArray[(TCaps and TC_SCROLLBLT) = TC_SCROLLBLT], lvTextCaps);
  end;
end;

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

continues

LISTING 10.7 Continued

```
  // Store the printer names in the combo box.
  cbPrinters.Items.Assign(Printer.Printers);
  // Display the default printer in the combo box.
  cbPrinters.ItemIndex := Printer.PrinterIndex;
  // Invoke the combo's OnChange event
  cbPrintersChange(nil);
end;

procedure TMainForm.cbPrintersChange(Sender: TObject);
begin
  Screen.Cursor := crHourGlass;
  try
    // Populate combo with available printers
    Printer.PrinterIndex := cbPrinters.ItemIndex;
    with Printer do
      GetPrinter(Device, Driver, Port, ADevMode);
    // Fill the general page with printer information
    with lvGeneralData.Items do
    begin
      Clear;
      AddListViewItem('Port', Port, lvGeneralData);
      AddListViewItem('Device', Device, lvGeneralData);

      Rslt := DeviceCapabilitiesA(Device, Port, DC_DRIVER, nil, nil);
      AddListViewItem('Driver Version', IntToStr(Rslt), lvGeneralData);
    end;

    // The functions below make use of the GetDeviceCapabilitiesA function.
    GetBinNames;
    GetDuplexSupport;
    GetCopies;
    GetEMFStatus;
    GetResolutions;
    GetTrueTypeInfo;

    // The functions below make use of the GetDeviceCaps function.
    GetDevCapsPaperNames;
    GetDevCaps;    // Fill Device Caps page.
    GetRasterCaps; // Fill Raster Caps page.
    GetCurveCaps;  // Fill Curve Caps page.
    GetLineCaps;   // Fill Line Caps page.
    GetPolyCaps;   // Fill Polygonal Caps page.
    GetTextCaps;   // Fill Text Caps page.
  finally
```

```
    Screen.Cursor := crDefault;
  end;
end;

end.
```

Summary

This chapter teaches the techniques you need to know in order to program any type of custom printing, from simple printing to more advanced techniques. You also learned a methodology that you can apply to any printing task. Additionally, you learned about the `TDeviceMode` structure and how to perform common printing tasks. You'll use more of this knowledge in upcoming chapters, where you build even more powerful printing methods.

Porting to Delphi 5

CHAPTER

15

IN THIS CHAPTER

If you're upgrading to Delphi 5 from a previous version, this chapter is written for you. The first section of this chapter discusses the issues involved in moving from any version of Delphi to Delphi 5. In the second, third, and fourth sections, you learn about the often subtle differences between the various 32-bit versions of Delphi and how to take these differences into account as you migrate applications to Delphi 5. The fourth section of this chapter is intended to help those migrating 16-bit Delphi 1.0 applications to the 32-bit world of Delphi 5. Although Borland makes a concerted effort to ensure that your code is compatible between versions, it's understandable that some changes have to be made in the name of progress, and certain situations require code changes if applications are to compile and run properly under the latest version of Delphi.

New to Delphi 5

In general, the more recent the version of Delphi you're coming from, the easier it will be for you to port to Delphi 5. However, whether you're migrating from Delphi 1, 2, 3, or 4, this section provides the information necessary for moving up to Delphi 5.

Which Version?

Although most Delphi code will compile for all versions of the compiler, in some instances language or VCL differences require that you write slightly differently to accomplish a given task for each product version. Occasionally, you might need to be able to compile for multiple versions of Delphi from one code base. For this purpose, each version of the Delphi compiler contains a `VER`*xxx* conditional define for which you can test in your source code. Because Borland C++Builder also ships with new versions of the Delphi compiler, these editions also contain this conditional define. Table 15.1 shows the conditional defines for the various versions of the Delphi compiler.

TABLE 15.1 Conditional Defines for Compiler Versions

Product	*Conditional Define*
Delphi 1	`VER80`
Delphi 2	`VER90`
C++Builder 1	`VER95`
Delphi 3	`VER100`
C++Builder 3	`VER110`
Delphi 4	`VER120`
C++Builder 4	`VER120`
Delphi 5	`VER130`

Using these defines, the source code you must write in order to compile for different compiler versions would look something like this:

```
{$IFDEF VER80}
  Delphi 1 code goes here
{$ENDIF}
{$IFDEF VER90}
  Delphi 2 code goes here
{$ENDIF}
{$IFDEF VER95}
  C++Builder 1 code goes here
{$ENDIF}
{$IFDEF VER100}
  Delphi 3 code goes here
{$ENDIF}
{$IFDEF VER110}
  C++Builder 3 code goes here
{$ENDIF}
{$IFDEF VER120}
  Delphi 4 and C++Builder 4 code goes here
{$ENDIF}
{$IFDEF VER130}
  Delphi 5 code goes here
{$ENDIF}
```

NOTE

If you're wondering why the Delphi 1.0 compiler is considered version 8, Delphi 2 version 9, and so on, it's because Delphi 1.0 is considered version 8 of Borland's Pascal compiler. The last Turbo Pascal version was 7.0, and Delphi is the evolution of that product line.

Just as you'll have to deal with differences in the language and VCL between Delphi versions, you'll also have to deal with differences in the Windows API. If you need to cope with differences in the 16-bit and 32-bit APIs from a single code base, you can take advantage of additional defines intended for this purpose. The 16-bit Delphi compiler defines `WINDOWS`, whereas the Win32 Delphi compilers define `WIN32`. The following code example demonstrates how to take advantage of these defines:

```
{$IFDEF WINDOWS}
  16-bit Windows-specific code goes here
{$ENDIF}
{$IFDEF WIN32}
  Win32-specific code goes here
{$ENDIF}
```

Units, Components, and Packages

Delphi 5 compiled units (DCU files) differ from those of all previous versions of Delphi (and C++Builder). You must have the source code to any units used in your application in order to build your application under any particular version of Delphi. This, of course, means that you won't be able to use any components used in your application—your own components or third-party components—unless you have the source to these components. If you don't have the source code to a particular third-party component, contact your vendor for a version of the component specific to your version of Delphi.

NOTE

This issue of compiler version versus unit file version is not a new situation and is the same as C++ compiler object file versioning. If you distribute (or buy) components without source code, you must understand that what you're distributing/buying is a compiler version–specific binary file that will probably need to be revised to keep up with subsequent compiler releases.

What's more, the issue of DCU versioning isn't necessarily a compiler-only issue. Even if the compiler weren't changed between versions, changes and enhancements to core VCL would probably still make it necessary that units be recompiled from source.

Delphi 3 also introduced *packages*, the idea of multiple units stored in a single binary file. Starting with Delphi 3, the component library became a collection of packages rather than one massive component library DLL. Like units, packages are not compatible across product versions, so you'll need to rebuild your packages for Delphi 5, and you'll need to contact the vendors of your third-party components for updated packages.

Migrating from Delphi 4

There are only a handful of migration issues as you take your Delphi 4 applications into Delphi 5. In many cases, you can simply load your project into Delphi 5 and hit the compile key. However, if you do run into problems, this section discusses the migration speed bumps you may face for getting things rolling in Delphi 5.

IDE Issues

Problems with IDE are likely the first you'll encounter as you migrate your applications. Here are a few of the issues you may encounter on the way:

- Delphi 4 debugger symbol files (RSM) are not compatible with Delphi 5. You'll know you're having this problem when you see the message "Error reading symbol file." If this happens, the fix is simple: Rebuild the application.

- Delphi 5 now defaults to storing form files in text mode. If you need to maintain DFM compatibility with earlier versions of Delphi, you'll need to save the forms files in binary instead. You can do this by unchecking New Forms As Text on the Preferences page of the Environment Options dialog.
- Code generation when importing and generating type libraries has been changed. In addition to some minor changes, the new generator has been enhanced to allow you to map symbol names; you can customize type library–to–Pascal symbol name mapping by editing the `tlibimp.sym` file. For directions, see the "Mapping Symbol Names in the Type Library" topic in the online help.

RTL Issues

The only issue you're likely to come across here deals with the setting of the floating-point unit (FPU) control word in DLLs. In previous versions of Delphi, DLLs would set the FPU control word, thereby changing the setting established by the host application. Now, DLL startup code no longer sets the FPU control word. If you need to set the control word to ensure some specific behavior by the FPU, you can do it manually using the `Set8087CW()` function in the `System` unit.

VCL Issues

There are a number of VCL issues that you may come across, but most involve some simple edits as a means to get your project on track in Delphi 5. Here's a list of these issues:

- The type of properties that represent an index into an image list has changed from `Integer` to `TImageIndex` type. `TImageIndex` is a strongly typed `Integer` defined in the `ImgList` unit as

  ```
  TImageIndex = type Integer;
  ```

  This should only cause problems in cases where exact type matching matters, such as when you're passing `var` parameters.
- `TCustomTreeview.CustomDrawItem()` has a new `var` parameter called `PaintImages` of type `Boolean`. If your application overrides this method, you'll need to add this parameter in order for it to compile in Delphi 5.
- The `CoInitFlags` variable in `ComObj`, which holds the flags passed to `CoInitializeEx()` in the `ComServ` unit, has been changed to properly support initialization of multithreaded COM servers. Now either the `COINIT_MULTITHREADED` or `COINIT_APARTMENTTHREADED` flags will be added when appropriate.
- If you're invoking pop-up menus in response to `WM_RBUTTONUP` messages or `OnMouseUp` events, you may exhibit "double" pop-up menus or no pop-up menus at all when compiling with Delphi 5. Delphi 5 now uses the `WM_CONTEXT` menu message to invoke pop-up menus.

Internet Development Issues

If you're developing applications with Internet support, we have some bad news and some good news:

- The `TWebBrowser` component, which encapsulates the Microsoft Internet Explorer ActiveX control, has replaced the `THTML` component from `Netmasters`. Although the `TWebBrowser` control is much more feature rich, you're faced with a good deal of rewrite if you used `THTML` because the interface is totally different. If you don't want to rewrite your code, you can go back to the old control by importing the `HTML.OCX` file from the `\Info\Extras\NetManage` directory on the Delphi 5 CD-ROM.
- Packages are now supported when building ISAPI and NSAPI DLLs. You can take advantage of this new support by replacing HTTPApp in your `uses` clause with WebBroker.

Database Issues

There are a few database issues that may trip you up as you migrate to Delphi 5. These involve some renaming of existing symbols and the new architecture of MIDAS:

- The type of the `TDatabase.OnLogin` event has been renamed `TDatabaseLoginEvent` from `TLoginEvent`. This is unlikely to cause problems, but you may run into troubles if you're creating and assigning to `OnLogin` in code.
- The global `FMTBCDToCurr()` and `CurrToFMTBCD()` routines have been replaced by the new `BCDToCurr` and `CurrToBCD` routines (and the corresponding protected methods on `TDataSet` have been replaced by the protected and undocumented `DataConvert` method).
- MIDAS has undergone some significant changes between Delphi 4 and 5. See Chapter 32, "MIDAS Development," for information on the changes, new features, and how to port your MIDAS applications to Delphi 5.

Migrating from Delphi 3

Although there aren't a great deal of compatibility issues between Delphi 3 and later versions, the few issues that do exist can be potentially more problematic than porting from any other previous version of Delphi to the next. Most of these issues revolve around new types and the changing behavior of certain existing types.

Unsigned 32-Bit Integers

Delphi 4 introduced the `LongWord` type, which is an unsigned 32-bit integer. In previous versions of Delphi, the largest integer type was a signed 32-bit integer. Because of this, many of

the types that you would expect to be unsigned, such as `DWORD`, `UINT`, `HResult`, `HWND`, `HINSTANCE`, and other handle types, were defined simply as `Integers`. In Delphi 4 and later, these types are redefined as `LongWords`. Additionally, the `Cardinal` type, which was previously a subrange type of `0..MaxInt`, is now also a `LongWord`. Although all this `LongWord` business won't cause problems in most circumstances, there are several problematic cases you should know about:

- `Integer` and `LongWord` are not `var`-parameter compatible. Therefore, you cannot pass a `LongWord` in a `var Integer` parameter, and vice versa. The compiler will give you an error in this case, so you'll need to change the parameter or variable type or typecast to get around this problem.
- Literal constants having the value of `$80000000` through `$FFFFFFFF` are considered `LongWords`. You must typecast such a literal to an `Integer` if you wish to assign it to an `Integer` type. Here's an example:

```
var
  I: Integer;
begin
  I := Integer($FFFFFFFF);
```

- Similarly, any literal having a negative value is out of range for a `LongWord`, and you'll need to typecast to assign a negative literal to a `LongWord`. Here's an example:

```
var
  L: LongWord;
begin
  L := LongWord(-1);
```

- If you mix signed and unsigned integers in arithmetic or comparison operations, the compiler will automatically promote each operand to `Int64` in order to perform the arithmetic or comparison. This can cause some very difficult-to-find bugs. Consider the following code:

```
var
  I: Integer;
  D: DWORD;
begin
  I := -1;
  D := $FFFFFFFF;
  if I = D then DoSomething;
```

  Under Delphi 3, *`DoSomething`* would execute because `-1` and `$FFFFFFFF` are the same value when contained in an `Integer`. However, because Delphi 4 and later will promote each operand to `Int64` in order to perform the most accurate comparison, the generated code ends up comparing `$FFFFFFFFFFFFFFFF` against `$00000000FFFFFFFF`, which is definitely not what's intended. In this case, *`DoSomething`* will not execute.

TIP

The compiler in Delphi 4 and later generates a number of new hints, warnings, and errors that deal with these type compatibility problems and implicit type promotions. Make sure you turn on hints and warnings when compiling in order to let the compiler help you write clean code.

64-Bit Integer

Delphi 4 also introduced a new type called `Int64`, which is a signed 64-bit integer. This new type is now used in the RTL and VCL where appropriate. For example, the `Trunc()` and `Round()` standard functions now return `Int64`, and there are new versions of `IntToStr()`, `IntToHex()`, and related functions that deal with `Int64`.

The Real Type

Starting with Delphi 4, the `Real` type became an alias for the `Double` type. In previous versions of Delphi and Turbo Pascal, `Real` was a six-byte floating-point type. This shouldn't pose any problems for your code unless you have `Real`s written to some external storage (such as a `file of record`) with an earlier version or you have code that depends on the organization of the `Real` in memory. You can force `Real` to be the old 6-byte type by including the `{$REALCOMPATIBILITY ON}` directive in the units you want to use the old behavior. If all you need to do is force a limited number of instances of the `Real` type to use the old behavior, you can use the `Real48` type instead.

Migrating from Delphi 2

You'll find that a high degree of compatibility between Delphi 2 and the later versions means a smooth transition into a more up-to-date Delphi version. However, some changes have been made since Delphi 2, both in the language and in VCL, that you'll need to be aware of to migrate to the latest version and take full advantage of its power.

Changes to Boolean Types

The implementation of the Delphi 2 Boolean types (`Boolean`, `ByteBool`, `WordBool`, `LongBool`) dictated that `True` was ordinal value `1` and `False` ordinal value `0`. To provide better compatibility with the Win32 API, the implementations of `ByteBool`, `WordBool`, and `LongBool` have changed slightly; the ordinal value of `True` is now `-1` (`$FF`, `$FFFF`, and `$FFFFFFFF`, respectively). Note that no change was made to the `Boolean` type. These changes have the potential to cause problems in your code—but only if you depend on the ordinal values of these types. For example, consider the following declaration:

```
var
  A: array[LongBool] of Integer;
```

This code is quite harmless under Delphi 2; it declares an `array[False..True]` (or `[0..1]`) `of Integer`, for a total of three elements. Under Delphi 3 and later, however, this declaration can cause some very unexpected results. Because `True` is defined as `$FFFFFFFF` for a `LongBool`, the declaration boils down to `array[0..$FFFFFFFF] of Integer`, or an array of 4 billion `Integers`! To avoid this problem, use the `Boolean` type as the array index.

Ironically, this change was necessary because a disturbing number of ActiveX controls and control containers (such Visual Basic) test `BOOLs` by checking for `-1` rather than testing for a zero or nonzero value.

TIP

To help ensure portability and to avoid bugs, never write code like this:

```
if BoolVar = True then ...
```

Instead, always test Boolean types like this:

```
if BoolVar then ...
```

ResourceString

If your application uses string resources, consider taking advantage of `ResourceStrings` as described in Chapter 2, "The Object Pascal Language." Although this won't improve the efficiency of your application in terms of size or speed, it will make language translation easier. `ResourceStrings` and the related topic of resource DLLs are required to be able to write applications displaying different language strings but have them all running on the same core VCL package.

RTL Changes

Several changes made to the runtime library (RTL) after Delphi 2 might cause problems as you migrate your applications. First, the meaning of the `HInstance` global variable has changed slightly: `HInstance` contains the instance handle of the current DLL, EXE, or package. Use the new `MainInstance` global variable when you want to obtain the instance handle of the main application.

The second significant change pertains to the `IsLibrary` global. In Delphi 2, you could check the value of `IsLibrary` to determine whether your code was executing within the context of a DLL or EXE. `IsLibrary` isn't package aware, however, so you can no longer depend on `IsLibrary` to be accurate, depending on whether it's called from an EXE, DLL, or a module

within a package. Instead, you should use the `ModuleIsLib` global, which returns `True` when called within the context of a DLL or package. You can use this in combination with the `ModuleIsPackage` global to distinguish between a DLL and a package.

TCustomForm

The Delphi 3 VCL introduced a new class between `TScrollingWinControl` and `TForm` called `TCustomForm`. In itself, that shouldn't pose a problem for you in migrating your applications from Delphi 2; however, if you have any code that manipulates instances of `TForm`, you might need to update it so that it manipulates `TCustomForms` instead of `TForms`. Some examples of these are calls to `GetParentForm()`, `ValidParentForm()`, and any usage of the `TDesigner` class.

CAUTION

The semantics for `GetParentForm()`, `ValidParentForm()`, and other VCL methods that return `Parent` pointers have changed slightly from Delphi 2. These routines can now return `nil`, even though your component has a parent window context in which to draw. For example, when your component is encapsulated as an ActiveX control, it may have a `ParentWindow`, but not a `Parent` control. This means you must watch out for Delphi 2 code that does this:

```
with GetParentForm(xx) do ...
```

`GetParentForm()` can now return `nil` depending on how your component is being contained.

GetChildren()

Component writers, be aware that the declaration of `TComponent.GetChildren()` has changed to read as follows:

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

The new `Root` parameter holds the component's root owner—that is, the component obtained by walking up the chain of the component's owners until `Owner` is `nil`.

Automation Servers

The code required for automation has changed significantly from Delphi 2. Chapter 23, "COM-based Technologies," describes the process of creating Automation servers in Delphi 5. Rather than describe the details of the differences here, suffice it to say that you should never mix the Delphi 2 style of creating Automation servers with the more recent style found in Delphi 3 and later.

In Delphi 2, automation is facilitated through the infrastructure provided in the `OleAuto` and `Ole2` units. These units are present in later releases of Delphi only for backward compatibility, and you shouldn't use them for new projects. Now the same functionality is provided in the `ComObj`, `ComServ`, and `ActiveX` units. You should never mix the former units with the latter in the same project.

Migrating from Delphi 1

Most of the changes required when porting Delphi 1 applications to a later version are necessary because of the nature of programming under a new operating system. Other changes are required because of enhancements in VCL and the Object Pascal language. Some people would prefer that Delphi 1 applications simply ran without modification under Delphi 5. If that's your opinion, keep in mind that sometimes it's necessary to leave a little behind to move ahead. You should find that each new version of Delphi strikes an excellent balance in this regard.

In addition to explaining what you need to know to migrate Delphi 1 applications, this section also provides you with information about optimizing your project for 32-bit Delphi and maintaining code that's compatible with both 16- and 32-bit Delphi.

Strings and Characters

In response to customer demand for a more flexible string, Borland introduced a new string type in Delphi 2 known as `AnsiString`. Among other benefits, `AnsiString` supports the creation of virtually unlimited-length strings. Delphi 2 also introduced new character and null-terminated string types to fully support application internationalization using the Unicode double-byte format. Delphi 3 took double-byte support even further with the introduction of the `WideString` type. By far the most common issues likely to arise as you migrate from Delphi 1 are those dealing with the use and manipulation of strings.

New Character Types

Strings, of course, are made up of characters, so it's important that you understand the behavior of character types in Delphi 4 before learning about the new string types. The most important change to this portion of the language is the new character type `WideChar`, introduced in Delphi 2 to support Unicode (or "wide") character and string types. In addition to `WideChar`, Delphi 3 introduced a new type name, `AnsiChar`, which specifies a normal single-byte character.

The `AnsiChar` type is the same as the Delphi 1 `Char` type. It's a one-byte value that can contain any of 256 different values. Use `AnsiChar` only when you know that the value in question will always be one byte in size.

Use `WideChar` for a character value that's two bytes in size. `WideChar` exists for compatibility with the Unicode character standard adopted by the Win32 API to support local-language

strings. Because of its two-byte size, a `WideChar` can contain any one of 65,536 possible values, enough for even the largest alphabets.

> The purpose of Unicode is to support use of local-language strings across the entire system (and across the global network) without loss of information. There are one-byte character sets for most languages (except Far Eastern languages), but converting between language character sets is not always reversible, so conversion introduces loss of information. Unicode solves that by eliminating the need to convert between character set encodings. It also addresses the Far Eastern language issue by providing a very large character set base for encoding pictograph-per-word languages such as Chinese.

Of course, the `char` type is still valid in Delphi. Currently, the `char` and `AnsiChar` types are equivalent. However, Borland reserves the right to change the definition of `char` to `WideChar` in some future version of Delphi; you should never depend on the size of a `char` being a certain length in your code—always use `SizeOf()` to determine its actual size.

New String Types

Listed here are the string types supported in Delphi 5:

- `AnsiString` (also referred to as *long string* and *huge string*) is the new default string type for Object Pascal. It's composed of `AnsiChar` characters and allows for lengths of up to 1GB. This string type is also compatible with null-terminated strings. This string is always dynamically allocated and lifetime managed.
- `ShortString` is synonymous with the standard string type in Delphi 1.0. The capacity of `ShortString` is limited to 255 characters.
- `WideString` is comprised of `WideChar` characters, and, like `AnsiString`, it's automatically allocated and lifetime managed. Chapter 2, "The Object Pascal Language," contains a complete rundown on this and other string types.
- `PAnsiChar` is a pointer to a null-terminated `AnsiChar` string.
- `PWideChar` is a pointer to a null-terminated string of `WideChar` characters, making up a Unicode, or double-byte, string.
- `PChar` is a pointer to a null-terminated `Char` string, which is fully compatible with C-style strings used in Windows API functions. This type hasn't changed from version 1.0 and is currently defined as `PAnsiChar`.

By default, strings defined in Delphi 2 and later are `AnsiStrings`. So if you declare a `string`, as shown here, the compiler assumes that you're creating an `AnsiString`:

```
var
  S: String;  // S is an AnsiString
```

Alternatively, you can cause variables declared as `String` to instead be of type `ShortString` by using the `$H` compiler directive. When the value of the `$H` compiler directive is negative, `String` variables are `ShortStrings`; when the value of the directive is positive (the default), `String` variables are `AnsiStrings`. The following code demonstrates this behavior:

```
var
  {$H-}
  S1: String;  // S1 is a ShortString
  {$H+}
  S2: String;  // S2 is an AnsiString
```

The exception to the `$H` rule is that a `String` declared with an explicit size (limited to a maximum of 255 characters) is always a `ShortString`:

```
var
  S: String[63];  // A ShortString of up to 63 characters
```

CAUTION

Be careful when passing strings declared in units with `$H+` to functions and procedures defined in units with `$H-`, and vice versa. These types of errors can introduce some hard-to-find bugs into your applications.

Setting String Length

In Delphi 1, you could set the length of a string by assigning a value to the 0, byte as shown here:

```
S[0] := 23;   { sets the length byte of a short string }
```

This was possible because the maximum length of a short string (255) could be stored in the leading byte. Because the maximum length of a long string is 1GB, the size obviously can't fit in one byte; the length is therefore stored differently. Because of this issue, Delphi 2 introduced a new standard procedure called `SetLength()` that you should use to set the length of a string. `SetLength()` is defined as follows:

```
procedure SetLength(var S: String; NewLength: Integer};
```

`SetLength()` can be used with short and long strings. If you want to maintain one set of source code for 16-bit Delphi 1 projects and 32-bit Delphi projects, you can define a `SetLength()` function, as follows, for 16-bit Delphi 1 projects:

```
{$IFDEF WINDOWS}
{ for 16-bit Delphi 1 projects }
```

```
procedure SetLength(var S: String; NewLength: Integer);
begin
  S[0] := Char(NewLength);
end;
{$ENDIF}
```

TIP

For more information on the physical layout of the `AnsiString` type, see Chapter 2, "The Object Pascal Language."

Dynamically Allocated Strings

In Delphi 1, it's possible to use variables of type `PString` to implement dynamically allocated strings by allocating memory with the `NewStr()`, `GetMem()`, or `AllocMem()` standard procedure. In 32-bit Delphi, because long strings are automatically allocated dynamically from the heap, there's no need to use such techniques. Change your `PString` references to `String` (the code that dynamically creates and frees memory). You must also remove any dereferencing of the `PString` variable that appears in your code. Consider the following block of Delphi 1 code:

```
var
  S1, S2: PString;
begin
  S1 := AllocMem(SizeOf(S1^));
  S1^ := 'Give up the rock.';
  S2 := NewStr(S1^);
  FreeMem(S1, SizeOf(S1^));
  Edit1.Text := S2^;
  DisposeStr(S2);
end;
```

This code can be enormously simplified (and optimized) simply by taking advantage of long strings, as shown here:

```
var
  S1, S2: string;
begin
  S1 := 'Give up the rock.';
  S2 := S1;
  Edit1.Text := S2;
end;
```

Indexing Strings as Arrays

Sometimes you want to access a certain character in a string by indexing the string as an array. For example, the following line of code sets the fifth character in the string to *A*:

```
S[5] := 'A';
```

This type of operation is still perfectly legitimate with long strings, but there's one caveat: Because long strings are dynamically allocated, you must ensure that the length of the string is greater than or equal to the character element you attempt to index. For example, the following code is invalid:

```
var
  S: string;
begin
  S[5] := 'A';  // Space for S has not yet been allocated!!
end;
```

However, this code is quite valid:

```
var
  S: string;
begin
  S := 'Hello';  // allocates enough room for the string
  S[5] := 'A';
end;
```

This code is also valid:

```
var
  S: string;
begin
  SetLength(S, 5); // allocate 5 characters for S
  S[5] := 'A';
end;
```

CAUTION

You should not assume that a character index into a string is the same thing as the byte offset into the string. For example, `WideStringvVar[5]` accesses the fifth character (at byte offset 10).

Null-Terminated Strings

When calling Windows 3.1 API functions in Delphi 1, programmers had to be aware of the difference between the Pascal `String` type and the C-style `PChar` (the null-terminated string used in Windows). Long strings make it much easier to call Win32 API functions. Long strings are both heap allocated and guaranteed to be null terminated. For these reasons, you can simply typecast a long string variable when you need to use it as a null-terminated `PChar` in a Win32 API function call. Imagine that you have procedure `Foo()`, defined as follows:

```
procedure Foo(P: PChar);
```

In Delphi 1, you would typically call this function like this:

```
var
  S: string;            { Pascal short string }
  P: PChar;             { null terminated string }
begin
  S := 'Hello world';   { initialize S }
  P := AllocMem(255);   { allocate P }
  StrPCopy(P, S);       { copy S to P }
  Foo(P);               { call Foo with P }
  FreeMem(P, 255);      { dispose P }
end;
```

Using a 32-bit version of Delphi, you can call `Foo()` using a long string variable with the following syntax:

```
var
  S: string;       // a long string is null terminated
begin
  S := 'Hello World';
  Foo(PChar(S));   // fully compatible with PChar type
end;
```

This means that you can optimize your 32-bit code by removing unnecessary temporary buffers to hold null-terminated strings.

Null-Terminated Strings as Buffers

A common use for `PChar` variables is as a buffer to be passed to an API function that fills the buffer string with information. A classic example is the `GetWindowsDirectory()` API function, defined in the Win32 API as follows:

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

If your goal is to store the Windows directory in a string variable, a common shortcut under Delphi 1 is to pass the address of the first element of the string as shown here:

```
var
  S: string;
begin
  GetWindowsDirectory(@S[1], 254); { 254 = room for null }
  S[0] := Chr(StrLen(@S[1]));      { adjust length }
end;
```

This technique doesn’t work with long strings for two reasons. First, as mentioned earlier, you must give the string an initial length before any space is allocated. Second, because a long

string is already a pointer to heap space, using the `@` operator effectively passes a pointer to a pointer to a character—definitely not what you intended! With long strings, this technique is streamlined by typecasting the string to a `PChar`:

```
var
  S: string;
begin
  SetLength(S, MAX_PATH + 1);      // allocate space
  GetWindowsDirectory(PChar(S), MAX_PATH);
  SetLength(S, StrLen(PChar(S))); // adjust length
end;
```

PChars as Strings

Because long strings can be used as `PChars`, it's only fair that the reverse hold true. Null-terminated strings are assignment compatible to long strings. In Delphi 1, the following code requires a call to `StrPCopy()`:

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := StrPas(P);
  StrDispose(P);
end;
```

Now you can accomplish the same thing with a simple assignment using long strings:

```
var
  S: string;
  P: PChar;
begin
  P := StrNew('Object Pascal');
  S := P;
  StrDispose(P);
end;
```

Similarly, you can also pass null-terminated strings to functions and procedures that expect `String` parameters. Suppose that procedure `Bar()` is defined as follows:

```
procedure Bar(S: string);
```

You can call `Bar()` using a `PChar` as follows:

```
var
  P: PChar;
begin
  P := StrNew('Hello');
```

```
  Bar(P);
  StrDispose(P);
end;
```

However, this technique doesn't work with procedures and functions that accept `String`s by reference. Suppose that procedure `Bar()` were instead defined like this:

```
procedure Bar(var S: string);
```

The sample code just presented couldn't be used to call `Bar()`. Instead, you have to define a temporary string to pass to `Bar()`, as shown here:

```
var
  P: PChar;
  TempStr: string;
begin
  P := StrNew('Hello');
  TempStr := P;
  Bar(TempStr);
  StrDispose(P);
  P := PChar(TempStr);
end;
```

Delphi 2 introduced a standard procedure called `SetString()` that allows you to copy only a portion of a `PChar` into a string variable. `SetString()` has an advantage in that it works with both long and short strings. The definition of `SetString()` is given here:

```
procedure SetString(var S: string: Buffer: PChar; Len: Integer);
```

CAUTION

Be wary of assigning a string variable to a `PChar` variable when the lifetime of the `PChar` variable is greater than that of the string. Because the string will be deallocated when it leaves scope, the `PChar` variable will point to garbage after the string leaves scope. The following code illustrates this problem:

```
var
  P: PChar;

procedure Bar(var P: PChar);
var
  S: String;
begin
  S := 'Hola Mundo';
  P := PChar(S);   // P is valid here
end;               // S is freed here
```

```
procedure Foo;
begin
  Bar(P);
  ShowMessage(P);  // DANGER! P is now invalid
end;
```

Variable Size and Range

Another issue that might arise as you migrate your Delphi code is that some types change size (and therefore range) when they move from 16-bit to 32-bit environments. Tables 15.2 and 15.3 show the differences with regard to these types.

TABLE 15.2 Variable Size Differences

Type	*16-Bit Size*	*32-Bit Size*
`Integer`	Two bytes	Four bytes
`Cardinal`	Two bytes	Four bytes
`String`	256 bytes	Four bytes

TABLE 15.3 Variable Range Differences

Type	*16-Bit Range*	*32-Bit Range*
`Integer`	-32,768..32,767	-2,147,483,648..2,147,483,647
`Cardinal`	0..65,536	0..2,147,483,647
`String`	255 characters	1GB of characters

For the most part, these new variable sizes have no effect on your applications. In those areas where you depend on type sizes, make sure you use the `SizeOf()` function. Also, if you've written any of these types to binary files or BLOBs in 16-bit Delphi, you must take into account the change in size as you read the data back in with 32-bit Delphi. For this purpose, Table 15.4 indicates which 32-bit Delphi types share binary compatibility with the Delphi 1 types.

TABLE 15.4 Variable Type Compatibility

16-Bit Delphi Type	*Compatible 32-Bit Delphi Type*
`Integer`	`SmallInt`
`Cardinal`	`Word`
`string`	`ShortString`

Record Alignment

By default in 32-bit Delphi, records are padded so that they're aligned properly; 32-bit data (such as `Integer`) is aligned on 32-bit (`DWORD`) boundaries, and 16-bit data (such as `Word`) is aligned on addresses that are even multiples of 16.

```
type
  TX = record
    B: Byte;
    L: Longint;
  end;
```

NOTE

Data is aligned on boundaries in order to optimize processor performance when accessing memory.

With the default compiler settings, the Delphi 1 `SizeOf(TX)` function returns `5`; under Delphi 2 and later, `SizeOf(TX)` function returns `8`. This isn't normally an issue; however, it can be an issue if you don't use `SizeOf()` to determine the size of the record in your code or if you have records written to a binary file.

The reason that the 32-bit Delphi compiler aligns record elements on `DWORD` boundaries is that doing so enables the compiler to generate more optimized code. If there's a reason you want to block this behavior, you can use the new `packed` modifier in the `type` declaration:

```
type
  TX = packed record
    B: Byte;
    L: Longint;
  end;
```

With `TX` defined as a *packed record*, as shown here, `SizeOf(TX)` now returns `5` under Delphi 2 and later. You can make packed records the default by using the `$A-` compiler directive.

TIP

If you can reorder the fields in a record to make fields start on their natural boundaries (for example, reorder four `Byte` fields so they all come before an `Integer` field, instead of two on either side of the `Integer`), you can get optimum packing without incurring the performance cost of unaligned data.

32-Bit Math

A much more subtle issue regarding variable size is that the 32-bit Delphi compiler automatically performs optimized 32-bit math on all operands in an expression (Delphi 1 used 16-bit math). Consider the following Object Pascal code:

```
var
  L: longint;
  w1, w2: word;
begin
  w1 := $FFFE;
  w2 := 5;
  L := w1 + w2;
end;
```

Under Delphi 1.0, the value of `L` at the end of this routine is `3` because the calculation of `w1 + w2` is stored as a 16-bit value, and the operation causes the result to wrap. Under Delphi 3, the value of `L` at the end of this routine is `$10003` because the `w1 + w2` calculation is performed using 32-bit math. The repercussion of the new functionality is that if you use and depend on the compiler's range-checking logic to catch "errors" such as these in Delphi 1.0, you must use some other method for finding those errors in 32-bit versions of Delphi, because a range-check error won't occur.

The TDateTime Type

To maintain compatibility with OLE and the Win32 API, the zero value of a `TDateTime` variable has changed. Date values start at `00/00/0000` under Delphi 1; they start at `12/30/1899` under 32-bit Delphi. Although this change won't affect dates stored in a database field, it will affect binary dates stored in a binary file or database BLOB field.

Unit Finalization

Delphi 1 provides a procedure called `AddExitProc()` and a pointer called `ExitProc` that enable you to define a procedure as containing "exit code" for a particular unit. Under 32-bit Delphi, the process of adding an exit procedure to a unit is greatly simplified with the addition of the unit's `finalization` section. Intended as a counterpart to the unit's `initialization` section, the code in a `finalization` section is guaranteed to be called when the application closes. Although this type of change isn't necessary to compile your application under 32-bit Delphi, it does make for much cleaner code.

NOTE

Conversion of `ExitProcs` to `finalization` blocks is mandatory for packages. Packages can be dynamically loaded and unloaded multiple times at design time, and `ExitProcs` are not called when a package is dynamically unloaded by the IDE. Therefore, your cleanup code must go in `finalization` sections.

Consider the following Delphi 1 `initialization` section and exit code:

```
procedure MyExitProc;
begin
  MyGlobalObject.Free;
end;
initialization
  AddExitProc(MyExitProc);
  MyGlobalObject := TGlobalObject.Create;
end.
```

This code can be simplified using the `finalization` section in 32-bit Delphi, as shown here:

```
initialization
  MyGlobalObject := TGlobalObject.Create;
finalization
  MyGlobalObject.Free;
end.
```

Assembly Language

Because assembly language is highly dependent on the platform for which it is written, the 16-bit built-in assembly language in Delphi 1 applications doesn't work in 32-bit Delphi. You must rewrite such routines using 32-bit assembly language.

Additionally, certain interrupts might not be supported under Win32. An example of interrupts no longer supported under Win32 is the suite of DOS Protected Mode Interface (DPMI) functions provided under interrupt `$31`. In some cases, Win32 API functions and procedures take the place of interrupts (the new Win32 file I/O functions are an example). If your application makes use of interrupts, refer to the Win32 documentation to check the alternatives in your specific case.

Additionally, inline hexadecimal code is no longer supported in the 32-bit Delphi compiler. If you have any routines that use inline code, replace them with 32-bit assembly language routines.

Calling Conventions

Delphi 1 can use either the `cdecl` or `pascal` calling convention for parameter passing and stack cleanup for function and procedure calls. The default calling convention for Delphi 1 is `pascal`.

Delphi 2 introduced directives representing two new calling conventions: `register` and `stdcall`. The `register` calling convention is the default for Delphi 2 and 3, offering faster performance. This method dictates that the first three 32-bit parameters be passed in the `eax`, `edx`, and `ecx` registers, respectively. Remaining parameters use the `pascal` calling convention. The `stdcall` calling convention is a hybrid of `pascal` and `cdecl` in that the parameters are passed using the `cdecl` convention but the stack is cleaned up using the `pascal` convention.

Delphi 3 introduced a new procedure directive called `safecall`. `safecall` follows the `stdcall` convention for parameter passing and also allows COM errors to be handled in a more Delphi-like manner. Most COM functions return `HRESULT` values as errors, whereas the preferred manner of error handling in Delphi is through the use of structured exception handling. When you call a `safecall` function from Delphi, the `HRESULT` return value of the function is converted into an exception that you may handle. When implementing a `safecall` function in Delphi, any exceptions raised in the function will be converted into an `HRESULT` value, which is returned to the caller.

NOTE

Although functions and procedures in the 16-bit Windows API use the `pascal` calling convention, Win32 API functions and procedures use the `stdcall` convention. Consequently, if you have any callback functions in your code, those also use the `stdcall` calling convention. Consider the following callback, intended for use with the `EnumWindows()` API function under 16-bit Windows:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; export;
```

It's defined as follows for 32-bit Windows:

```
function EnumWindowsProc(Handle: hwnd; lParam: Longint): BOOL; stdcall;
```

Dynamic Link Libraries (DLLs)

The creation and use of DLLs work very much the same in 32-bit Delphi as in Delphi 1, although there are a few minor differences. Some of these issues are listed here:

- Because of Win32's flat memory model, the `export` directive (necessary for callback and DLL functions in Delphi 1) is unnecessary in later versions. It's simply ignored by the compiler.

- If you're writing a DLL you intend to share with executables written using other development tools, it's a good idea to use the `stdcall` directive for maximum compatibility.
- The preferred way to export functions in a Win32 DLL is by name (instead of by ordinal). The following example exports functions by ordinal, the Delphi 1 way:

```
function SomeFunction: integer; export;
begin
.
.
.
end;

procedure SomeProcedure; export;
begin
.
.
.
end;

exports
  SomeFunction index 1,
  SomeProcedure index 2;
```

  Here are the same functions exported by name, the 32-bit Delphi way:

```
function SomeFunction: integer; stdcall;
begin
.
.
.
end;

procedure SomeProcedure; stdcall;
begin
.
.
.
end;

exports
  SomeFunction name 'SomeFunction';
  SomeProcedure name 'SomeProcedure';
```

- Exported names are case sensitive. You must use proper case when importing functions by name and when calling `GetProcAddress()`.
- When you import a function or procedure and specify the library name after the external directive, the file extension can be included. If no extension is specified, `.DLL` is assumed.

- Under Windows 3.*x*, a DLL in memory has only one data segment that's shared by all instances of the DLL. Therefore, if applications A and B both load DLL C, changes made to global variables in DLL C from application A are visible to application B, and the reverse is also true. Under Win32, each DLL receives its own data segment, so changes made to global DLL data from one program aren't visible to another program.

TIP

See Chapter 9, "Dynamic Link Libraries," for more information on the behavior of DLLs under Win32.

Windows Operating System Changes

In several areas, changes in the 32-bit architecture of Windows can have an impact on code written in Delphi. These include changes resulting from the 32-bit memory model, changes in resource formats, unsupported features, and changes to the Windows API itself.

32-Bit Address Space

Win32 provides a 4GB flat address space for your application. The term *flat* means that all segment registers hold the same value and that the definition of a pointer is an offset into that 4GB space. Because of this, any code in your Delphi 1 applications that depends on the concept of a pointer consisting of a selector and offset must be rewritten to accommodate the new architecture.

The following elements of the Delphi 1 runtime library are 16-bit pointer specific and are not in the 32-bit Delphi runtime library: `DSeg`, `SSeg`, `CSeg`, `Seg`, `Ofs`, and `SPtr`.

Because of the way in which Win32 uses a hard disk paging file to simulate RAM on demand, the Delphi 1.0 `MemAvail()` and `MaxAvail()` functions are no longer useful for gauging available memory. If you have to obtain this information in 32-bit Delphi, use the `GetHeapStatus()` Delphi RTL function, which is defined as follows:

```
function GetHeapStatus: THeapStatus;
```

The `THeapStatus` record is designed to provide information (in bytes) on the status of the heap for your process. This record is defined as follows:

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
```

```
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

Again, because the nature of Win32 is such that the amount of "free" memory has little meaning, most users will find the `TotalAllocated` field (which indicates how much heap memory has been allocated by the current process) most useful for debugging purposes.

NOTE

For more information on the internals of the Win32 operating system, see Chapter 3, "The Win32 API."

32-Bit Resources

If you have any resources (RES or DCR files) that you link into your application or use with a component, you must create 32-bit versions of these files before you can use them with 32-bit Delphi. Typically, this is a simple matter of using the included Image Editor or a separate resource editor (such as Resource Workshop) to save the resource file in a 32-bit-compatible format.

VBX Controls

Because Microsoft doesn't support VBX controls (which are inherently 16-bit controls) in 32-bit applications for Windows 95 and Windows NT, they aren't supported in 32-bit Delphi. ActiveX controls (OCXs) effectively replace VBX controls in 32-bit platforms. If you want to migrate a Delphi 1.0 application that uses VBX controls, contact your VBX vendor to get an equivalent 32-bit ActiveX control.

Changes to the Windows API Functions

Some Windows APIs or features have changed from Windows 3.1 to Win32. Some 16-bit API functions no longer exist in Win32, some functions are obsolete but continue to exist for the sake of compatibility, and some accept different parameters or return different types or values. Tables 15.5 and 15.6 list these functions. For complete documentation on these functions, see the Win32 API online help that comes with Delphi.

TABLE 15.5 Obsolete Windows 3.*x* API Functions

Windows 3.x Function	*Win32 Replacement*
`OpenComm()`	`CreateFile()`
`CloseComm()`	`CloseHandle()`
`FlushComm()`	`PurgeComm()`
`GetCommError()`	`ClearCommError()`
`ReadComm()`	`ReadFile()`
`WriteComm()`	`WriteFile()`
`UngetCommChar()`	N/A
`DlgDirSelect()`	`DlgDirSelectEx()`
`DlgDirSelectComboBox()`	`DlgDirSelectComboBoxEx()`
`GetBitmapDimension()`	`GetBitmapDimensionEx()`
`SetBitmapDimension()`	`SetBitmapDimensionEx()`
`GetBrushOrg()`	`GetBrushOrgEx()`
`GetAspectRatioFilter()`	`GetAspectRatioFilterEx()`
`GetTextExtent()`	`GetTextExtentPoint()`
`GetViewportExt()`	`GetViewportExtEx()`
`GetViewportOrg()`	`GetViewportOrgEx()`
`GetWindowExt()`	`GetWindowExtEx()`
`GetWindowOrg()`	`GetWindowOrgEx()`
`OffsetViewportOrg()`	`OffsetViewportOrgEx()`
`OffsetWindowOrg()`	`OffsetWindowOrgEx()`
`ScaleViewportExt()`	`ScaleViewportExtEx()`
`ScaleWindowExt()`	`ScaleWindowExtEx()`
`SetViewportExt()`	`SetViewportExtEx()`
`SetViewportOrg()`	`SetViewportOrgEx()`
`SetWindowExt()`	`SetWindowExtEx()`
`SetWindowOrg()`	`SetWindowOrgEx()`
`GetMetafileBits()`	`GetMetafileBitsEx()`
`SetMetafileBits()`	`SetMetafileBitsEx()`
`GetCurrentPosition()`	`GetCurrentPositionEx()`
`MoveTo()`	`MoveToEx()`
`DeviceCapabilities()`	`DeviceCapabilitiesEx()`
`DeviceMode()`	`DeviceModeEx()`

continues

TABLE 15.5 Continued

Windows 3.x Function	*Win32 Replacement*
`ExtDeviceMode()`	`ExtDeviceModeEx()`
`FreeSelector()`	N/A
`AllocSelector()`	N/A
`ChangeSelector()`	N/A
`GetCodeInfo()`	N/A
`GetCurrentPDB()`	`GetCommandLine()` and/or `GetEnvironmentStrings()`
`GlobalDOSAlloc()`	N/A
`GlobalDOSFree()`	N/A
`SwitchStackBack()`	N/A
`SwitchStackTo()`	N/A
`GetEnvironment()`	(Win32 file I/O functions)
`SetEnvironment()`	(Win32 file I/O functions)
`ValidateCodeSegments()`	N/A
`ValidateFreeSpaces()`	N/A
`GetInstanceData()`	N/A
`GetKBCodePage()`	N/A
`GetModuleUsage()`	N/A
`Yield()`	`WaitMessage()` and/or `Sleep()`
`AccessResource()`	N/A
`AllocResource()`	N/A
`SetResourceHandler()`	N/A
`AllocDSToCSAlias()`	N/A
`GetCodeHandle()`	N/A
`LockData()`	N/A
`UnlockData()`	N/A
`GlobalNotify()`	N/A
`GlobalPageLock()`	`VirtualLock()`

TABLE 15.6 Win32 API Compatibility Function

Windows 3.x Function	*Win32 Replacement*
`DefineHandleTable()`	N/A
`MakeProcInstance()`	N/A

Windows 3.x Function	*Win32 Replacement*
`FreeProcInstance()`	N/A
`GetFreeSpace()`	`GlobalMemoryStatus()`
`GlobalCompact()`	N/A
`GlobalFix()`	N/A
`GlobalUnfix()`	N/A
`GlobalWire()`	N/A
`GlobalUnwire()`	N/A
`LocalCompact()`	N/A
`LocalShrink()`	N/A
`LockSegment()`	N/A
`UnlockSegment()`	N/A
`SetSwapAreaSize()`	N/A

Concurrent 16-Bit and 32-Bit Projects

This section gives you some guidelines for developing projects that compile under 16-bit Delphi 1 or any 32-bit version of Delphi. Although you can follow the directions outlined in this chapter for source code compatibility, here are some further pointers to help you along:

- The `WINDOWS` conditional is defined by the compiler under Delphi 1; the `WIN32` conditional is defined under 32-bit versions of Delphi. You can use these defines to perform conditional compilation with the `{$IFDEF WINDOWS}` and `{$IFDEF WIN32}` directives.
- Avoid the use of any component or feature in 32-bit Delphi that isn't supported by Windows 3.1 or Delphi 1 if you want to recompile with Delphi 1 for a 16-bit application. For example, avoid the use of Win95 components and features such as multithreading that aren't available in Windows 3.1. The easiest way to ensure compatibility with projects intended for both Delphi 1 and 32-bit versions of Delphi is to develop the project in Delphi 1 and recompile it with 32-bit Delphi for optimized 32-bit performance.
- Be wary of differences between the APIs. If you have to use an API procedure or function that's implemented differently in the different platforms, make use of the `WINDOWS` and `WIN32` conditional defines.
- Each new version often adds more properties or different properties than those found in the previous version. This means, for example, that when version 5 components are saved to a DFM file, these new properties are written as well. Although it's often possible to "ignore" the errors that occur when loading projects with these properties in Delphi 1, it's often a more favorable solution to maintain two separate sets of DFM files, one for each platform.

Win32s

One other option for leveraging a single code base into both 16- and 32-bit Windows is to attempt to run your 32-bit Delphi applications under Win32s. Win32s is an add-on to Windows 3.*x*, which enables a subset of the Win32 API to function on 16-bit Windows. One serious drawback to this method is that many Win32 features, such as threads, are not available under Win32s (this precludes your use of the Borland Database Engine in this circumstance because the BDE makes use of threads). If you choose this route, you should also bear in mind that Win32s is not an officially supported platform for 32-bit Delphi, so you're on your own if things don't quite function as you expect.

Summary

Armed with the information provided by this chapter, you should be able to migrate your projects smoothly from any previous version of Delphi to Delphi 5. Also, with a bit of work, you'll be able to maintain projects that work with multiple versions of Delphi.

MDI Applications

CHAPTER

16

IN THIS CHAPTER

The Multiple Document Interface, otherwise known as *MDI*, was introduced to Windows 2.0 in the Microsoft Excel spreadsheet program. MDI gave Excel users the ability to work on more than one spreadsheet at a time. Other uses of MDI included the Windows 3.1 Program Manager and File Manager programs. Borland Pascal for Windows is another MDI application.

During the development of Windows 95, many developers were under the impression that Microsoft was going to eliminate MDI capabilities. Much to their surprise, Microsoft kept MDI as part of Windows 95 and there has been no further word about Microsoft's intention to get rid of it.

CAUTION

Microsoft has acknowledged that the Windows MDI implementation is flawed. It advised developers against continuing to build apps in the MDI model. Since then, Microsoft has returned to building MS apps in the MDI model but does so without using the Windows MDI implementation. You can still use MDI, but be forewarned that the Windows MDI implementation is still flawed, and Microsoft has no plans to fix those problems. What we present in this chapter is a safe implementation of the MDI model.

Handling events simultaneously between multiple forms might seem difficult. In traditional Windows programming, you had to have knowledge of the Windows class `MDICLIENT`, MDI data structures, and the additional functions and messages specific to MDI. With Delphi 5, creating MDI applications is greatly simplified. When you finish this chapter, you'll have a solid foundation for building MDI applications, which you can easily expand to include more advanced techniques.

Creating the MDI Application

To create MDI applications, you need familiarity with the form styles `fsMDIForm` and `fsMDIChild` and a bit of MDI programming methodology. The following sections present some basic concepts regarding MDI and show how MDI works with special MDI child forms.

Understanding MDI Basics

To understand MDI applications, first you must understand how they're constructed. Figure 16.1 shows an MDI application similar to one you'll build in this chapter.

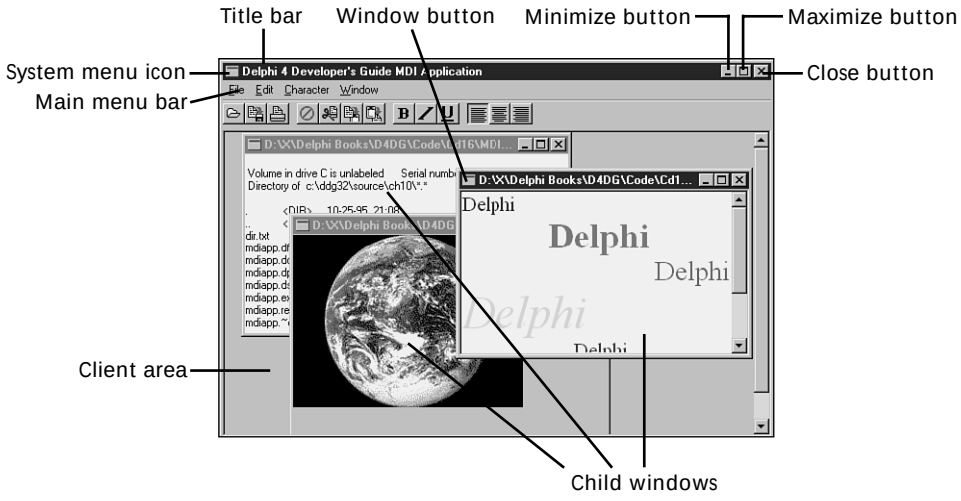

FIGURE 16.1
The structure of an MDI application.

Here are the windows involved with an MDI application:

- *Frame window.* The application's main window. It has a caption, menu bar, and system menu. Minimize, maximize, and close buttons appear in its upper-right corner. The blank space inside the frame window is known as its *client area* and is actually the client window.
- *Client window.* The manager for MDI applications. The client window handles all MDI-specific commands and manages the child windows that reside on its surface—including the drawing of MDI child windows. The client is created automatically by the *Visual Component Library* (VCL) when you create a frame window.
- *Child windows.* MDI child windows are your actual documents—text files, spreadsheets, bitmaps, and other document types. Child windows, like frame windows, have a caption, system menu, minimize, maximize, and close buttons, and possibly a menu. It's possible to place a help button on a child window. A child window's menu is combined with the frame window's menu. Child windows never move outside the client area.

Delphi 5 does not require you to be familiar with the special MDI window's messages. The client window is responsible for managing MDI functionality, such as cascading and tiling child windows. To cascade child windows using the traditional method, for example, use the Windows API function `SendMessage()` to send a `WM_MDICASCADE` message to the client window:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
```

```
  SendMessage(ClientHandle, WM_MDICASCADE, 0, 0);
end;
```

In Delphi 5, just call the `Cascade()` method:

```
procedure TFrameForm.Cascade1Click(Sender: TObject);
begin
  cascade;
end;
```

The following sections show you a complete MDI application whose child MDI windows have the functionality of a text editor, a bitmap file viewer, and a rich text format editor. The purpose of this application is to show you how to build MDI applications whose child windows each display and edit different types of information. For example, the text editor allows you to edit any text-based file. The rich text editor allows you to edit rich text–formatted (`.rtf`) files. Finally, the bitmap viewer allows you to view any Windows bitmapped file.

We also show you how to perform some advanced MDI techniques using the Win32 API. These techniques mainly have to do with managing MDI child forms in an MDI-based application. First, we'll discuss the building of the child forms and their functionality. Then we'll talk about the main form.

The Child Form

As mentioned earlier, this MDI application contains three types of child forms: `TMdiEditForm`, `TMdiRTFForm`, and `TMdiBMPForm`. Each of these three types descends from `TMDIChildForm`, which serves as a base class. The following section describes the `TMDIChildForm` base class. The sections after that talk about the three child forms used in the MDI application.

The TMDIChildForm Base Class

The child forms used in the MDI application have some common functionality. They all have the same File menu and their `FormStyle` property is set to `fsMDIChild`. Additionally, they all make use of a `TToolBar` component. By deriving each child form from a base form class, you can avoid having to redefine these settings for each form. We defined a base form, `TMDIChildForm`, as shown in `MdiChildFrm.pas` (refer to Listing 16.1).

LISTING 16.1 `MdiChildFrm.pas`: A Unit Defining `TMDIChildForm`

```
unit MdiChildFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, ComCtrls, ToolWin, ImgList;
```

```
type

  TMDIChildForm = class(TForm)
  (* Component list removed, refer to online source. *)
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormDeactivate(Sender: TObject);
end;

var
  MDIChildForm: TMDIChildForm;

implementation
uses MainFrm,  Printers;

{$R *.DFM}

procedure TMDIChildForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  { Reassign the toolbar parent }
  tlbMain.Parent := self;
  { If this is the last child form being displayed, then make the main form's
    toolbar visible }
  if (MainForm.MDIChildCount = 1) then
    MainForm.tlbMain.Visible := True
end;

procedure TMDIChildForm.mmiExitClick(Sender: TObject);
begin
  MainForm.Close;
end;

procedure TMDIChildForm.mmiCloseClick(Sender: TObject);
begin
  Close;
end;

procedure TMDIChildForm.mmiOpenClick(Sender: TObject);
begin
  MainForm.mmiOpenClick(nil);
end;
```

continues

LISTING 16.1 Continued

```
procedure TMDIChildForm.mmiNewClick(Sender: TObject);
begin
  MainForm.mmiNewClick(nil);
end;

procedure TMDIChildForm.FormActivate(Sender: TObject);
begin
  { When the form becomes active, hide the main form's toolbar and assign
    this child form's toolbar to the parent form.  Then display this
    child form's toolbar. }
  MainForm.tlbMain.Visible := False;
  tlbMain.Parent := MainForm;
  tlbMain.Visible := True;
end;

procedure TMDIChildForm.FormDeactivate(Sender: TObject);
begin
  { The child form becomes inactive when it is either destroyed or when another
    child form becomes active.  Hide this form's toolbar so that the next
form's
    toolbar will be visible. }
  tlbMain.Visible := False;
end;

end.
```

NOTE

Note that we have removed the component declarations for the `TMdiChildForm` base class from the printed text for space reasons.

`TMDIChildForm` contains event handlers for the menu items for its main menu as well as for some common tool buttons. Actually, the tool buttons are simply wired to the event handler of their corresponding menu item. Some of these event handlers call methods on the main form. For example, notice that the `mmiNewClick()` event handler calls the `MainForm.mmiNewClick()` event handler. `TMainForm.mmiNewClick()` contains functionality for creating a new MDI child form. You'll notice that there are other event handlers such as `mmiOpenClick()` and `mmiExitClick()` that call the respective event handlers on the main form. We'll cover `TMainForm`'s functionality later in the section "The Main Form."

Because each MDI child needs to have the same functionality, it makes sense to put this functionality into a base class from which the MDI child forms can descend. This way, the MDI

child forms do not have to define these same methods. They will inherit the main menu as well as the toolbar components that you see on the main form.

Notice in the `TMDIChildForm.FormClose()` event handler that you set the `Action` parameter to `caFree` to ensure that the `TMDIChildForm` instance is destroyed when closed. You do this because MDI child forms don't close automatically when you call their `Close()` method. You must specify, in the `OnClose` event handler, what you want done with the child form when its `Close()` method is called. The child form's `OnClose` event handler passes in a variable `Action`, of type `TCloseAction`, to which you must assign one of four possible values:

- `caNone`. Do nothing.
- `caHide`. Hide the form but don't destroy it.
- `caFree`. Free the form.
- `caMinimize`. Minimize the form (this is the default).

`TCloseAction` is an enumerated type.

When a form becomes active, its `OnActivate` event handler is called. You must perform some specific logic whenever a child form becomes active. Therefore, in the `TMdiChildForm.FormActivate()` event handler, you'll see that we make the main form's toolbar invisible while setting the child form's toolbar to visible. We also assign the main form as the parent to the child form's toolbar so that the toolbar appears on the main form and not on the child form. This is one way you might give the main form a different toolbar when a different type of MDI child form is active. The `OnDeactivate` event handler simply makes the child form's toolbar invisible. Finally, the `OnClose` event reassigns the child form as the parent to the toolbar, and if the current child form is the only child form, it makes the main form's toolbar visible. The effect is that the main form has a single toolbar with buttons that change depending on the type of active child form.

The Text Editor Form

The text editor form enables the user to load and edit any text file. This form, `TMdiEditForm`, is inherited from `TMDIChildForm`. `TMdiEditForm` contains a client-aligned `TMemo` component.

`TMdiEditForm` also contains `TPrintDialogs`, `TSaveDialog` and `TFontDialog` components.

`TMdiEditForm` is not an autocreated form and is removed from the list of autocreated forms in the Project Options dialog box.

NOTE

None of the forms, except for `TMainForm`, in the MDI project are automatically created and therefore have been removed from the list of autocreated forms. These forms are created dynamically in the project's source code.

`TMdiEditForm`'s source code is given in Listing 16.2.

LISTING 16.2 `MdiEditFrm.pas`: A Unit Defining `TMdiEditForm`

```
unit MdiEditFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Menus, ExtCtrls, Buttons, ComCtrls,
  ToolWin, MdiChildFrm, ImgList;

type

  TMdiEditForm = class(TMDIChildForm)
    memMainMemo: TMemo;
    SaveDialog: TSaveDialog;
    FontDialog: TFontDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiDelete: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;
    PrintDialog: TPrintDialog;

    { File Event Handlers }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);
```

```
    { Edit Event Handlers }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);

    { Character Event Handlers }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Form Event Handlers }
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure mmiPrintClick(Sender: TObject);
  public
    { User Defined Methods }
    procedure OpenFile(FileName: String);
    procedure SetButtons;
  end;

var
  MdiEditForm: TMdiEditForm;

implementation

uses Printers;

{$R *.DFM}
{ File Event Handlers }

procedure TMdiEditForm.mmiSaveClick(Sender: TObject);
begin
  inherited;
  { If there isn't a caption, then there isn't already a filename.
    Therefore, call mmiSaveAsClick since it gets a filename. }
  if Caption = '' then
    mmiSaveAsClick(nil)
  else begin
    { Save to the file specified by the form's Caption. }
    memMainMemo.Lines.SaveToFile(Caption);
    memMainMemo.Modified := false; // Set to false since the text is saved.
```

continues

LISTING 16.2 Continued

```
  end;
end;

procedure TMdiEditForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    { Set caption to filename specified by SaveDialog1 since this
      may have changed. }
    Caption := SaveDialog.FileName;
    mmiSaveClick(nil); // Save the file.
  end;
end;

{ Edit Event Handlers }

procedure TMdiEditForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CutToClipBoard;
end;

procedure TMdiEditForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  memMainMemo.CopyToClipBoard;
end;

procedure TMdiEditForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.PasteFromClipBoard;
end;

procedure TMdiEditForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  memMainMemo.ClearSelection;
end;

procedure TMdiEditForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
```

```
  memMainMemo.Perform(EM_UNDO, 0, 0);
end;

procedure TMdiEditForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  memMainMemo.SelectAll;
end;

{ Character Event Handlers }
procedure TMdiEditForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

{ TAlignment is defined by VCL as:

  TAlignment = (taLeftJustify, taRightJustify, taCenter);

  Therefore each of the menu items contains the appropriate Tag property
  whose value represents one of the TAlignment values: 0, 1, 2 }

  { If the menu invoked this event handler, set it to checked and
    set the alignment for the memo }
  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    memMainMemo.Alignment := TAlignment(TMenuItem(Sender).Tag);
  end
  { If a TToolButton from the main form invoked this event handler,
    set the memo's alignment and then check the appropriate TMenuItem. }
  else if Sender is TToolButton then
  begin
    memMainMemo.Alignment := TAlignment(TToolButton(Sender).Tag);
    case memMainMemo.Alignment of
      taLeftJustify:   mmiLeft.Checked := True;
      taRightJustify:  mmiRight.Checked := True;
      taCenter:        mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;

procedure TMdiEditForm.mmiBoldClick(Sender: TObject);
```

continues

LISTING 16.2 Continued

```
begin
  inherited;
  if not mmiBold.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsBold]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsBold];
  SetButtons;
end;

procedure TMdiEditForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsItalic]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsItalic];
  SetButtons;
end;

procedure TMdiEditForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    memMainMemo.Font.Style := memMainMemo.Font.Style + [fsUnderline]
  else
    memMainMemo.Font.Style := memMainMemo.Font.Style - [fsUnderline];
  SetButtons;
end;

procedure TMdiEditForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with memMainMemo do
  begin
    WordWrap := not WordWrap;
    { Remove scrollbars if Memo1 is wordwrapped since they're not
      required. Otherwise, make sure scrollbars are present. }
    if WordWrap then
      ScrollBars := ssVertical
   else
      ScrollBars := ssBoth;
    mmiWordWrap.Checked := WordWrap;
  end;
end;

procedure TMdiEditForm.mmiFontClick(Sender: TObject);
```

```
begin
  inherited;
  FontDialog.Font := memMainMemo.Font;
  if FontDialog.Execute then
    memMainMemo.Font := FontDialog.Font;
end;

{ Form Event Handlers }
procedure TMdiEditForm.FormCloseQuery(Sender: TObject;
     var CanClose: Boolean);
{ This procedure ensures that the user has saved the contents of the
  memo if it was modified since the last time the file was saved. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if memMainMemo.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]), mtConfirmation,
mbYesNoCancel, 0);
    case MsgVal of
      mrYes:    mmiSaveClick(Self);
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiEditForm.OpenFile(FileName: string);
begin
  memMainMemo.Lines.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiEditForm.SetButtons;
{ This procedure ensures that menu items and buttons on the main form
  accurately reflect various settings for the memo. }
begin
  mmiBold.Checked := fsBold in memMainMemo.Font.Style;
  mmiItalic.Checked := fsItalic in memMainMemo.Font.Style;
  mmiUnderLine.Checked := fsUnderline in memMainMemo.Font.Style;

  tbBold.Down      := mmiBold.Checked;
```

continues

LISTING 16.2 Continued

```
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderLine.Checked;
  tbLAlign.Down    := mmiLeft.Checked;
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiEditForm.mmiPrintClick(Sender: TObject);
var
  i: integer;
  PText: TextFile;
begin
  inherited;
  if PrintDialog.Execute then
  begin
    AssignPrn(PText);
    Rewrite(PText);
    try
      Printer.Canvas.Font := memMainMemo.Font;
      for i := 0 to memMainMemo.Lines.Count -1 do
        writeln(PText, memMainMemo.Lines[i]);
    finally
      CloseFile(PText);
    end;
  end;
  end;

end.
```

Most of the methods for `TMdiEditForm` are event handlers for the various menus in `TMdiEditForm`'s main menu, the same menu inherited from `TMdiChildForm`. Also notice that additional menu items have been added to the main menu that apply specifically to `TMdiEditForm`.

You'll notice that there are no event handlers for the File, New, File, Open, File, Close, and File, Exit menus because they're already linked to `TMdiChildForm`'s event handlers.

The event handlers for the Edit menu items are all single-line methods that interact with the `TMemo` component. For example, you'll notice that the event handlers `mmiCutClick()`, `mmiCopyClick()`, and `mmiPasteClick()` interact with the Windows Clipboard in order to perform cut, copy, and paste operations. The other edit event handlers perform various editing functions on the memo component that have to do with deleting, clearing, and selecting text.

The Character menu applies various formatting attributes to the memo.

Notice that we stored a unique value in the `Tag` property of the `TToolButton` components for setting text alignment. This `Tag` value represents a value in the `TAlignment` enumerated type. This value is extracted from the `Tag` value of the `TToolButton` component that invoked the event handler to set the appropriate alignment for the memo component.

All menu items and tool buttons that set text alignment are wired to the `CharAlignClick()` event handler. This is why you have to check and respond appropriately in the event handler depending on whether a `TMenuItem` or `TToolButton` component invoked the event.

`CharAlignClick()` calls the `SetButtons()` method, which sets various menu items and components accordingly based on the memo's attributes.

The `mmiWordWrapClick()` event handler simply toggles the memo's wordwrap attribute and then the `Checked` property for the menu item. This method also specifies whether the memo component contains scrollbars based on its word-wrapping capability.

The `mmiFontClick()` event handler invokes a `TFontDialog` component and applies the selected font to the memo. Notice that before launching the `FontDialog` component, the `Font` property is set to reflect the memo's font so that the correct font is displayed in the dialog box.

The `mmiSaveAsClick()` event handler invokes a `TSaveDialog` component to get a filename from the user to which the memo's contents will be saved. When the file is saved, the `TMdiEditForm.Caption` property is set to reflect the new filename.

The `mmiSaveClick()` event handler calls the `mmiSaveAsClick()` event handler if a filename doesn't exist. This is the case if the user creates a new file instead of opening an existing one. Otherwise, the memo's contents are saved to the existing file specified by the `MdiEditForm.Caption` property. Notice that this event handler also sets `memMainMemo.Modified` to `False`. `Modified` is automatically set to `True` whenever the user changes the contents of a `TMemo` component. However, it's not set to `False` automatically whenever its contents are saved.

The `FormCloseQuery()` method is the event handler for the `OnCloseQuery` event. This event handler evaluates the `memMainMemo.Modified` property when the user attempts to close the form. If the memo has been modified, the user is notified and asked whether he or she wants to save the contents of the memo.

The public method `TMdiEditForm.OpenFile()` loads the file specified by the `FileName` parameter and places the file's contents into the `memMainMemo.Lines` property and then sets the form's `Caption` to reflect this filename.

That completes the functionality for `TMdiEditForm`. The other forms are somewhat similar in functionality.

The Rich Text Editor Form

The rich text editor enables the user to load and edit rich text–formatted files. This form, `TMdiRtfForm`, is derived from `TMDIChild`. It contains a client-aligned `TRichEdit` component.

`TMdiRtfForm` and `TMdiEditForm` are practically identical except that `TMdiRtfForm` contains a `TRichEdit` component as its editor; `TMdiEditForm` uses a `TMemo` component. `TMdiRtfForm` differs from the text editor in that the text attributes applied to the `TRichEdit` component affect paragraphs or selected text in the `TRichEdit` component; they affect the entire text with the `TMemo` component.

The source code for `TMdiRtfForm` is shown in Listing 16.3.

LISTING 16.3 `MdiRtfFrm.pas`: A Unit Defining `TMdiRtfForm`

```
unit MdiRtfFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MdiChildFrm, StdCtrls, ComCtrls,
  ExtCtrls, Buttons, Menus, ToolWin, ImgList;

type

  TMdiRtfForm = class(TMDIChildForm)
    reMain: TRichEdit;
    FontDialog: TFontDialog;
    SaveDialog: TSaveDialog;
    mmiEdit: TMenuItem;
    mmiSelectAll: TMenuItem;
    N7: TMenuItem;
    mmiPaste: TMenuItem;
    mmiCopy: TMenuItem;
    mmiCut: TMenuItem;
    mmiCharacter: TMenuItem;
    mmiFont: TMenuItem;
    N8: TMenuItem;
    mmiWordWrap: TMenuItem;
    N9: TMenuItem;
    mmiCenter: TMenuItem;
    mmiRight: TMenuItem;
    mmiLeft: TMenuItem;
    mmiUndo: TMenuItem;
    mmiDelete: TMenuItem;
```

```
    N4: TMenuItem;
    mmiBold: TMenuItem;
    mmiItalic: TMenuItem;
    mmiUnderline: TMenuItem;

    { File Event Handlers }
    procedure mmiSaveClick(Sender: TObject);
    procedure mmiSaveAsClick(Sender: TObject);

    { Edit Event Handlers }
    procedure mmiCutClick(Sender: TObject);
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiDeleteClick(Sender: TObject);
    procedure mmiUndoClick(Sender: TObject);
    procedure mmiSelectAllClick(Sender: TObject);

    { Character Event Handlers }
    procedure CharAlignClick(Sender: TObject);
    procedure mmiBoldClick(Sender: TObject);
    procedure mmiItalicClick(Sender: TObject);
    procedure mmiUnderlineClick(Sender: TObject);
    procedure mmiWordWrapClick(Sender: TObject);
    procedure mmiFontClick(Sender: TObject);

    { Form Event Handlers }
    procedure FormShow(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure reMainSelectionChange(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    { User-Defined Functions. }
    procedure OpenFile(FileName: String);
    function GetCurrentText: TTextAttributes;
    procedure SetButtons;
  end;

var
  MdiRtfForm: TMdiRtfForm;

implementation
{$R *.DFM}
{ File Event Handlers }

procedure TMdiRtfForm.mmiSaveClick(Sender: TObject);
begin
```

continues

LISTING 16.2 Continued

```
  inherited;
  reMain.Lines.SaveToFile(Caption);
end;

procedure TMdiRtfForm.mmiSaveAsClick(Sender: TObject);
begin
  inherited;
  SaveDialog.FileName := Caption;
  if SaveDialog.Execute then
  begin
    Caption := SaveDialog.FileName;
    mmiSaveClick(Sender);
  end;
end;

{ Edit Event Handlers }

procedure TMdiRtfForm.mmiCutClick(Sender: TObject);
begin
  inherited;
  reMain.CutToClipBoard;
end;

procedure TMdiRtfForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  reMain.CopyToClipBoard;
end;

procedure TMdiRtfForm.mmiPasteClick(Sender: TObject);
begin
  inherited;
  reMain.PasteFromClipBoard;
end;

procedure TMdiRtfForm.mmiDeleteClick(Sender: TObject);
begin
  inherited;
  reMain.ClearSelection;
end;

procedure TMdiRtfForm.mmiUndoClick(Sender: TObject);
begin
  inherited;
  reMain.Perform(EM_UNDO, 0, 0);
```

```
end;

procedure TMdiRtfForm.mmiSelectAllClick(Sender: TObject);
begin
  inherited;
  reMain.SelectAll;
end;

{ Character Event Handlers }

procedure TMdiRtfForm.CharAlignClick(Sender: TObject);
begin
  inherited;
  mmiLeft.Checked := false;
  mmiRight.Checked := false;
  mmiCenter.Checked := false;

  { If a TMenuItem invoked this event handler, set its checked
    property to true and set the attribute to RichEdit1's current
    paragraph. }
  if Sender is TMenuItem then
  begin
    TMenuItem(Sender).Checked := true;
    with reMain.Paragraph do
      if mmiLeft.Checked then
        Alignment := taLeftJustify
      else if mmiRight.Checked then
        Alignment := taRightJustify
      else if mmiCenter.Checked then
        Alignment := taCenter;
  end
  { If one of the main form's tool buttons invoked this event handler
    set the attribute to reMain's current paragraph and set the
    alignment menu items accordingly. }
  else if Sender is TSpeedButton then
  begin
    reMain.Paragraph.Alignment :=
       TAlignment(TSpeedButton(Sender).Tag);
    case reMain.Paragraph.Alignment of
      taLeftJustify: mmiLeft.Checked := True;
      taRightJustify: mmiRight.Checked := True;
      taCenter: mmiCenter.Checked := True;
    end;
  end;
  SetButtons;
end;
```

continues

LISTING 16.2 Continued

```
procedure TMdiRtfForm.mmiBoldClick(Sender: TObject);
begin
  inherited;
  if not mmiBold.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsBold]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsBold];
end;

procedure TMdiRtfForm.mmiItalicClick(Sender: TObject);
begin
  inherited;
  if not mmiItalic.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsItalic]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsItalic];
end;

procedure TMdiRtfForm.mmiUnderlineClick(Sender: TObject);
begin
  inherited;
  if not mmiUnderline.Checked then
    GetCurrentText.Style := GetCurrentText.Style + [fsUnderline]
  else
    GetCurrentText.Style := GetCurrentText.Style - [fsUnderline];
end;

procedure TMdiRtfForm.mmiWordWrapClick(Sender: TObject);
begin
  inherited;
  with reMain do
  begin
    { Remove scrollbars if Memo1 is wordwrapped since they're not
      required. Otherwise, make sure scrollbars are present. }
    WordWrap := not WordWrap;    if WordWrap then
      ScrollBars := ssVertical
    else
      ScrollBars := ssNone;
    mmiWordWrap.Checked := WordWrap;
  end;
end;

procedure TMdiRtfForm.mmiFontClick(Sender: TObject);
begin
  inherited;
  FontDialog.Font.Assign(reMain.SelAttributes);
```

```
  if FontDialog.Execute then
    GetCurrentText.Assign(FontDialog.Font);
  reMain.SetFocus;
end;

{ Form Event Handlers }

procedure TMdiRtfForm.FormShow(Sender: TObject);
begin
  inherited;
  reMainSelectionChange(nil);
end;


procedure TMdiRtfForm.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
{ This procedure ensures that the user has saved the contents of
  reMain if it was modified since the last time the file was saved. }
const
  CloseMsg = '''%s'' has been modified, Save?';
var
  MsgVal: integer;
  FileName: string;
begin
  inherited;
  FileName := Caption;
  if reMain.Modified then
  begin
    MsgVal := MessageDlg(Format(CloseMsg, [FileName]), mtConfirmation,
mbYesNoCancel, 0);
    case MsgVal of
      mrYes: mmiSaveClick(Self);
      mrCancel: CanClose := false;
    end;
  end;
end;

procedure TMdiRtfForm.reMainSelectionChange(Sender: TObject);
begin
  inherited;
  SetButtons;
end;

procedure TMdiRtfForm.OpenFile(FileName: String);
begin
  reMain.Lines.LoadFromFile(FileName);
```

continues

LISTING 16.2 Continued

```
  Caption := FileName;
end;

function TMdiRtfForm.GetCurrentText: TTextAttributes;
{ This procedure returns the text attributes of the current paragraph
  or based on the selected text of reMain.}
begin
  if reMain.SelLength > 0 then
    Result := reMain.SelAttributes
  else
    Result := reMain.DefAttributes;
end;

procedure TMdiRtfForm.SetButtons;
{ Ensures that the controls on the form reflect the
  current attributes of the paragraph by looking at the paragraph
  attributes themselves and setting the controls accordingly. }
begin

  with reMain.Paragraph do
  begin
    mmiLeft.Checked := Alignment = taLeftJustify;
    mmiRight.Checked := Alignment = taRightJustify;
    mmiCenter.Checked := Alignment = taCenter;
  end;

  with reMain.SelAttributes do
  begin
    mmiBold.Checked := fsBold in Style;
    mmiItalic.Checked := fsItalic in Style;
    mmiUnderline.Checked := fsUnderline in Style;
  end;
  mmiWordWrap.Checked := reMain.WordWrap;

  tbBold.Down      := mmiBold.Checked;
  tbItalic.Down    := mmiItalic.Checked;
  tbUnderline.Down := mmiUnderline.Checked;
  tbLAlign.Down    := mmiLeft.Checked;
  tbRAlign.Down    := mmiRight.Checked;
  tbCAlign.Down    := mmiCenter.Checked;
end;

procedure TMdiRtfForm.mmiPrintClick(Sender: TObject);
begin
```

```
  inherited;
  reMain.Print(Caption);
end;

end.
```

Like TMdiEditForm, most of TMdiRtfForm's methods are event handlers for the various menu items and tool buttons. These event handlers are similar to TMdiEditForm's event handlers.

TMdiRtfForm's File menu items invoke the File menu items of the TMdiChildForm base class. Recall that TMdiChildForm is the ancestor to TMdiRtfForm. The event handlers, mmiSaveClick() and mmiSaveAsClick(), both call reMain.Lines.SaveToFile() to save reMain's contents.

The event handlers for TMdiRtfForm's Edit menu items are single-line methods similar to the TMdiEditForm's Edit menu event handlers except that these event handlers call methods applicable to reMain. The method names are the same as the memo methods that perform the same operations.

TMdiRtfForm's Character menu items modify the alignment of *paragraphs* or selected text within the TRichEdit component (as opposed to the text within the entire component, as is the behavior with a TMemo component). Whether these attributes are applied to a paragraph or to selected text depends on the return value of the GetCurrentText() function. GetCurrentText() determines whether any text is selected by looking at the value of TRichEdit.SelLength. A zero value indicates that no text is selected. The TRichEdit.SelAttributes property refers to any selected text in the TRichEdit component. TRichEdit.DefAttributes refers to the current paragraph of the TRichEdit component.

The mmiFontClick() event handler allows the user to specify font attributes for a paragraph. Note that a paragraph can also refer to selected text.

Word wrapping is handled the same with the TRichEdit component as with the TMemo component in the text editor.

The TRichEdit.OnSelectionChange event handler is available to allow the programmer to provide some functionality whenever the selection of the component has changed. When the user moves the caret within the TRichEdit component, the component's SelStart property value changes. Because this action causes the OnSelectionChange event handler to be called, code was added to change the status of the various TMenuItem and TSpeedButton components on the main form to reflect the attributes of the text as the user scrolls through text in the TRichEdit component. This is necessary because text attributes in the TRichEdit component can differ; this is not the case with a TMemo component because attributes applied to a TMemo component apply to the entire component.

In functionality, the rich text editor form and the text editor form are, for the most part, very similar. The main difference is that the rich text editor allows users to change the attributes for separate paragraphs or selected text; the text editor is incapable of doing this.

The Bitmap Viewer—The Third MDI Child Form

The bitmap viewer enables the user to load and view Windows bitmap files. Like the other two MDI child forms, the bitmap viewer form, `TMdiBmpForm`, is derived from the `TMDIChildForm` base class. It contains a client-aligned `TImage` component.

`TMdiBmpForm` contains only its inherited `TMainMenu` component. Listing 16.4 shows the source code that defines `TMdiBmpForm`.

LISTING 16.4 `MdiBmpFrm.pas`: A Unit Defining `TMdiBmpForm`

```
unit MdiBmpFrm;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  MdiChildFrm, ExtCtrls, Menus, Buttons, ComCtrls, ToolWin, ImgList;

type
  TMdiBMPForm = class(TMDIChildForm)
    mmiEdit: TMenuItem;
    mmiCopy: TMenuItem;
    mmiPaste: TMenuItem;
    imgMain: TImage;
    procedure mmiCopyClick(Sender: TObject);
    procedure mmiPasteClick(Sender: TObject);
    procedure mmiPrintClick(Sender: TObject);
  public
    procedure OpenFile(FileName: string);
  end;

var
  MdiBMPForm: TMdiBMPForm;

implementation

uses ClipBrd, Printers;

{$R *.DFM}

procedure TMdiBMPForm.OpenFile(FileName: String);
```

```
begin
  imgMain.Picture.LoadFromFile(FileName);
  Caption := FileName;
end;

procedure TMdiBMPForm.mmiCopyClick(Sender: TObject);
begin
  inherited;
  ClipBoard.Assign(imgMain.Picture);
end;

procedure TMdiBMPForm.mmiPasteClick(Sender: TObject);
{ This method copies the contents from the clipboard into imgMain }
begin
  inherited;
  // Copy clipboard content to imgMain
  imgMain.Picture.Assign(ClipBoard);
  ClientWidth := imgMain.Picture.Width;
  { Adjust clientwidth to adjust the scollbars }
  VertScrollBar.Range := imgMain.Picture.Height;
  HorzScrollBar.Range := imgMain.Picture.Width;
end;

procedure TMdiBMPForm.mmiPrintClick(Sender: TObject);
begin
  inherited;

  with ImgMain.Picture.Bitmap do
  begin
      Printer.BeginDoc;
      Printer.Canvas.StretchDraw(Canvas.ClipRect, imgMain.Picture.Bitmap);
      Printer.EndDoc;
  end; { with }
end;

end.
```

There is not as much code for `TMdiBmpForm` as there was for the two previous forms. The File menu items invoke the `TMDIChildForm`'s event handlers just as the `TMdiEditForm` and `TMdiRtfForm` File menu items do. The Edit menu items copy and paste the bitmap to and from the Windows Clipboard, respectively. Before calling the `TImage.Picture.Assign()` method to assign the Clipboard data to the `TImage` component. The `TImage` component recognizes both the `CF_BITMAP` and `CF_PICTURE` formats as bitmaps.

The Windows Clipboard

The Clipboard provides the easiest way for two applications to share information. It's nothing more than a global memory block that Windows maintains for any application to access through a specific set of Windows functions.

The Clipboard supports several standard formats, such as text, OEM text, bitmaps, and metafiles; it also supports other specialized formats. Additionally, you can extend the Clipboard to support application-specific formats.

Delphi 5 encapsulates the Windows Clipboard with the global variable `Clipboard` of type `TClipBoard`, making it much easier for you to use. The `TClipBoard` class is covered in detail in Chapter 17, "Sharing Information with the Clipboard."

The Main Form

The main form is the form with which the user initially works to create or switch between MDI child forms. This form is appropriately named `MainForm`. `MainForm` serves as the parent to the text editor, bitmap viewer, and RTF editor MDI child forms.

`TMainForm` is not a descendant of `TMDIChildForm` as are the other forms discussed so far in this chapter. `TMainForm` has the `FormStyle` of `fsMDIForm` (the other three forms inherited the style `fsMDIChild` from `TMDIChild`). `TMainForm` contains a `TMainMenu` component and a `TOpenDialog` component. `TMainForm` also contains a toolbar that contains only one button. `TMainForm`'s source code is shown in Listing 16.5.

LISTING 16.5 `MdiMainForm.pas`: A Unit Defining `TMainForm`

```
unit MainFrm;

interface

uses
  WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Messages, Dialogs, SysUtils, ComCtrls,
  ToolWin, ExtCtrls, Buttons, ImgList;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    OpenDialog: TOpenDialog;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    N3: TMenuItem;
    mmiOpen: TMenuItem;
```

```
    mmiNew: TMenuItem;
    mmiWindow: TMenuItem;
    mmiArrangeIcons: TMenuItem;
    mmiCascade: TMenuItem;
    mmiTile: TMenuItem;
    mmiCloseAll: TMenuItem;
    tlbMain: TToolBar;
    ilMain: TImageList;
    tbFileOpen: TToolButton;

    { File Event Handlers }
    procedure mmiNewClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);

    { Window Event Handlers }
    procedure mmiTileClick(Sender: TObject);
    procedure mmiArrangeIconsClick(Sender: TObject);
    procedure mmiCascadeClick(Sender: TObject);
    procedure mmiCloseAllClick(Sender: TObject);
  public
    { User defined methods }
    procedure OpenTextFile(EditForm: TForm; Filename: string);
    procedure OpenBMPFile(FileName: String);
    procedure OpenRTFFile(RTFForm: TForm; FileName: string);
  end;

var
  MainForm: TMainForm;

implementation
uses MDIBmpFrm, MdiEditFrm, MdiRtfFrm, FTypForm;

const
  { Define constants to represent file name extensions }
  BMPExt      = '.BMP';  // Bitmapped file
  TextExt     = '.TXT';  // Text file
  RTFExt      = '.RTF';  // Rich Text Format file

{$R *.DFM}

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  { Determine the file type the user wishes to open by calling the
    GetFileType function. Call the appropriate method based on the
    retrieved file type. }
```

continues

LISTING 16.5 Continued

```
  case GetFileType of
    mrTXT: OpenTextFile(nil, ''); // Open a text file.
    mrRTF: OpenRTFFile(nil, '');  // Open an RTF file.
    mrBMP:
     begin
        { Set the default filter for OpenDialog1 for BMP files. }
        OpenDialog.FilterIndex := 2;
        mmiOpenClick(nil);
     end;
  end;
end;

procedure TMainForm.mmiOpenClick(Sender: TObject);
var
  Ext: string[4];
begin
  { Call the appropriate method based on the file type of the file
    selected from OpenDialog1 }
  if OpenDialog.Execute then
  begin
    { Get the file's extension and compare it to determine the
      file type the user is opening. Call the appropriate method and
      pass in the file name. }
    Ext := ExtractFileExt(OpenDialog.FileName);
    if CompareStr(UpperCase(Ext), TextExt) = 0 then
      OpenTextFile(ActiveMDIChild, OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), BMPExt) = 0 then
      OpenBMPFile(OpenDialog.FileName)
    else if CompareStr(UpperCase(Ext), RTFExt) = 0 then
      OpenRTFFile(ActiveMDIChild, OpenDialog.FileName);
  end;
end;

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

{ Window Event Handlers }

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  Tile;
end;

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

```
begin
   ArrangeIcons;
end;

procedure TMainForm.mmiCascadeClick(Sender: TObject);
begin
  Cascade;
end;


procedure TMainForm.mmiCloseAllClick(Sender: TObject);
var
  i: integer;
begin
  { Close all forms in reverse order as they appear in the
    MDIChildren property. }
  for i := MdiChildCount - 1 downto 0 do
    MDIChildren[i].Close;
end;

{ User Defined Methods }
procedure TMainForm.OpenTextFile(EditForm: TForm; FileName: string);
begin
  { If EditForm is of a TEditForm type, then give the user the option
   of loading the file contents into this form. Otherwise, create a
   new TEditForm instance and load the file into that instance }
  if (EditForm <> nil) and (EditForm is TMdiEditForm) then
    if MessageDlg('Load file into current form?', mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
    begin
      TMdiEditForm(EditForm).OpenFile(FileName);
       Exit;
    end;
  { Create a new TEditForm and call its OpenFile() method }
  with TMdiEditForm.Create(self) do
    if FileName <> '' then
      OpenFile(FileName)
end;

procedure TMainForm.OpenRTFFile(RTFForm: TForm; FileName: string);
begin
  { If RTFForm is of a TRTFForm type, then give the user the option
   of loading the file contents into this form. Otherwise, create a
   new TRTFForm instance and load the file into that instance }
  if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
    if MessageDlg('Load file into current form?', mtConfirmation,
```

continues

LISTING 16.5 Continued

```
      [mbYes, mbNo], 0) = mrYes then begin
        (RTFForm as TMdiRTFForm).OpenFile(FileName);
      Exit;
    end;
    { Create a new TRTFForm and call its OpenFile() method }
    with TMdiRTFForm.Create(self) do
      if FileName <> '' then
        OpenFile(FileName);
end;

procedure TMainForm.OpenBMPFile(FileName: String);
begin
  { Create a new TBMPForm instances and load a BMP file into it. }
  with TMdiBmpForm.Create(self) do
    OpenFile(FileName);
end;

end.
```

`TMainForm` uses another form, `FileTypeForm`, of the type `TFileTypeForm`. Listing 16.6 shows the source code for this form.

LISTING 16.6 The `FTYPFORM.PAS` Unit Defining `TFileTypeForm`

```
unit FTypForm;

interface

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

const
  mrTXT = mrYesToAll+1;
  mrBMP = mrYesToAll+2;
  mrRTF = mrYesToAll+3;


type

  TFileTypeForm = class(TForm)
    rgFormType: TRadioGroup;
    btnOK: TButton;
    procedure btnOkClick(Sender: TObject);
  end;
```

```
var
  FileTypeForm: TFileTypeForm;


function GetFileType: Integer;

implementation

function GetFileType: Integer;
{ This function returns the file type selected by the user as
  represented by one of the above defined constants. }
begin
  FileTypeForm := TFileTypeForm.Create(Application);
  try
    Result := FileTypeForm.ShowModal;
  finally
    FileTypeForm.Free;
  end;
end;

{$R *.DFM}

procedure TFileTypeForm.btnOkClick(Sender: TObject);
begin
  { Return the correct modal result based on the selected file type }
  case rgFormType.ItemIndex of
   0: ModalResult := mrTXT;
   1: ModalResult := mrRTF;
   2: ModalResult := mrBMP;
  end;
end;

end.
```

`TFileTypeForm` is used to prompt the user for a file type to create. This form returns the `ModalResult` based on which `TRadioButton` the user selected to indicate the type of file. The `GetFileType()` function takes care of creating, showing, and freeing the `TFileTypeForm` instance. This function returns the `TFileTypeForm.ModalResult` property. This form is not automatically created and has been removed from the list of autocreated forms for the project.

`TMainForm`'s toolbar contains only one button, which is used to open the initial child form. When a child form becomes active, its toolbar replaces the main form's toolbar. This logic is handled by the `OnActivate` event of the child form. `TMainForm`'s public methods `OpenTextFile()`, `OpenRTFFile()`, and `OpenBMPFile()` are called from the event handler `TMainForm.mmiOpenClick()`, which is invoked whenever the user selects the File, Open menu.

`OpenTextFile()` takes two parameters: a `TForm` instance and a filename. The `TForm` instance represents the currently active form for the application. The reason for passing this `TForm` instance to the `OpenTextFile()` method is so that the method can determine whether the `TForm` passed to it is of the `TMdiEditForm` class. If so, it's possible that the user is opening a text file in the existing `TMdiEditForm` instance rather than creating a new `TMdiEditForm` instance. If a `TMdiEditForm` instance is passed to this method, the user is prompted whether he or she wants the text file to be placed into this `TForm` parameter. If the user replies no or the `TMdiEditForm` parameter is `nil`, a new `TMdiEditForm` instance is created.

`OpenRTFFile()` operates the same as `OpenTextFile()` except that it checks for a `TRFTForm` class as the currently active form represented by the `TForm` parameter. The functionality is the same.

`OpenBMPFile()` always assumes that the user is opening a new file. This is because the `TMdiBmpForm` is only a viewer and not an editor. If the form allowed the user to edit a bitmapped image, the `OpenBMPFile()` method would function as do `OpenTextFile()` and `OpenRTFFile()`.

The `mmiNewClick()` event handler calls the `GetFileType()` function to retrieve a file type from the user. It then calls the appropriate `Open`*`XXX`*`File()` method based on the return value. If the file is a `.bmp` file, the `OpenDialog.Filter` property is set to the BMP filter by default and the `mmiOpenClick()` method is invoked because the user is not creating a new `.bmp` file but is opening an existing one.

The `mmiOpenClick()` event handler invokes `OpenDialog` and calls the appropriate `Open`*`XXX`*`File()` method. Notice that `OpenTextFile()` and `OpenRTFFile()` are passed the `TMainForm.ActiveMDIChild` property as the first parameter. `ActiveMDIChild` is the MDI child that currently has focus. Recall that both these methods determine whether the user wants to open a file into an existing MDI child form. If no forms are active, `ActiveMDIChild` is `nil`. If `ActiveMDIChild` is pointing to a `TMdiRTFForm` and `OpenTextFile()` is called, `OpenTextFile()` still functions correctly because of this statement:

```
if (RTFForm <> nil) and (RTFForm is TMdiRTFForm) then
```

This statement determines whether `ActiveMDIChild` points to a `TMdiRtfForm`. If not, a new form is created.

The event handler, `mmiExitClick()`, calls `TMainForm.Close()`; this method not only closes the main form, it also terminates the application. If there are any child forms open at the time this event handler is invoked, the child forms are also closed and destroyed.

The Window menu event handlers are single-line methods that affect how the MDI child forms are arranged on the main form's client area. Figures 16.2 and 16.3 show tiled and cascaded forms, respectively.

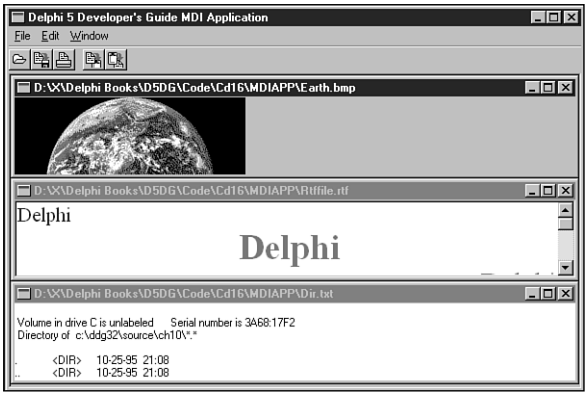

FIGURE 16.2
Tiled child forms.

FIGURE 16.3
Cascaded child forms.

The `mmiArrangeIconsClick()` method simply rearranges the icons in the main form's client area so that they're evenly spaced and do not overlap.

The `mmiCloseAllClick()` event handler closes all open MDI child forms. The loop that closes the child forms loops through all child forms in reverse order as they appear in the `MDIChildren` array property. The `MDIChildren` property is a zero-based array property of all MDI children active in an application. The `MDIChildCount` property is the number of children that are active.

This completes the discussion of the functionality of the MDI application. The following sections discuss some techniques and some of the components used with the various forms in the application.

Working with Menus

Using menus in MDI applications is no more difficult than using them in any other type of application. However, there are some differences in how menus work in MDI applications. The following sections show how an MDI application allows its child forms to share the same menu bar using a method called *menu merging*. You also learn how to make non-MDI applications share the same menu bar.

Merging Menus with MDI Applications

Take a look at the `TMainMenu` for the `TMdiEditForm`. By double-clicking the `TMainMenu` icon, you bring up the menu editor.

`TMdiEditForm`'s main menu contains three menu items along the menu bar. These items are File, Edit, and Character. Each of these menu items has a `GroupIndex` property that shows up in the Object Inspector as you click a menu item in the menu editor. Notice that the File menu item has a `GroupIndex` value of `0`. The Edit and Character menu items both have `GroupIndex` values of `1`.

Notice that `TMainForm`'s main menu has two menu items along its menu bar: File and Window. Like `TMdiEditForm`, `TMainForm`'s File menu item has a `GroupIndex` value of `0`. The Window menu item's `GroupIndex` property, on the other hand, has a value of `9`.

Also notice that the File menu for `TMainForm` and the File menu for `TMdiEditForm` have different submenu items. `TMdiEditForm`'s File menu has more submenu items than does `TMainForm`'s File menu.

The `GroupIndex` property is important because it allows menus of forms to be "merged." This means that when the main form launches a child form, the child form's main menu is merged with the main form's main menu. The `GroupIndex` property determines how the menus are ordered and which menus of the main form are replaced by menus of the child form. Note that menu merging applies only to menu items along the menu bar of a `TMainMenu` component and not to submenus.

Whenever a `GroupIndex` property for a child form's menu item has the same value as the `GroupIndex` property for a menu item on the main form, the child form's menu item replaces the main form's menu item. The remaining menus are arranged along the menu bar in the order specified by the `GroupIndex` properties of all combined menu items. When `MdiEditForm` is the active form in the project, the menu items that appear along the main form's menu bar are File, Edit, Character, and Window, in that order. Note that the File menu is `TMdiEditForm`'s File menu because both File menus have `GroupIndex` property values of `0`. Therefore, `TMdiChildForm`'s File menu replaces `TMainForm`'s File menu. The order of these menus directly reflects the order of the `GroupIndex` properties for each menu item along the menu bar: `0`, `1`, `1`, `9`.

NOTE

Although we don't use them here, there are certain numbering guidelines that you should follow so that your applications will better integrate with OLE Container's menu merging. These guidelines are explained in the "Borland Delphi Library Reference Guide."

This behavior is the same with the other forms in the MDI application. Whenever a form becomes active, the menu along the main menu bar changes to reflect the merging of menus for both the main form and child form. When you run the project, the menu bar changes depending on which child form is active.

Merging menus with MDI applications is automatic. As long as the values of the menu items' `GroupIndex` property is set in the order you specify, your menus merge correctly when you invoke MDI child forms.

For non-MDI applications, the process is just as easy but requires an extra step. We gave a quick example in Chapter 4, "Application Frameworks and Design Concepts," on merging menus in non-MDI applications when we discussed the `TNavStatForm`. However, in that application, we based this merging on child forms that were actually child windows to a control, other than the main form, and had to explicitly call the `Merge()` and `Unmerge()` functions. For merging menus with non–MDI-based applications in general, this process is not automatic, as it is with MDI applications. You must set the `AutoMerge` property to `True` for the `TMainMenu` on the form whose menus are to be merged with the main form. A sample project that shows menu merging for non-MDI forms can be found in the project `NonMDI.dpr` on the CD.

Adding a List of Open Documents to the Menu

To add a list of open documents to the Window menu, set the `WindowMenu` property of the main form to the menu item's instance that is to hold the list of open documents. For example, the `TMainForm.WindowMenu` property in the sample MDI application is set to `mmiWindow`, which refers to the Window menu along the menu bar. The selection you choose for this property must be a menu item that appears on the menu bar—it cannot be a submenu. The application displays a list of open documents in the Window menu.

Miscellaneous MDI Techniques

The following sections show various common techniques applicable to MDI applications.

Drawing a Bitmap in the MDI Client Window

When designing an MDI application, you might want to place a background image, such as a company logo, on the client area of an MDI application's main form. For regular (non-MDI) forms, this procedure is simple. You just place a `TImage` component on the form, set its `Align` property to `alClient`, and you're done (refer back to the bitmap viewer in the MDI sample application, earlier in this chapter). Placing an image on the main form of an MDI application, however, is a different story.

Recall that the client window of an MDI application is a separate window from the main form. The client window has many responsibilities of carrying out MDI-specific tasks, including the drawing of MDI child windows.

Think of it as though the main form is a transparent window over the client window. Whenever you place components such as `TButton`, `TEdit`, and `TImage` over the client area of the main form, these components are actually placed on the main form's transparent window. When the client window performs its drawing of child windows—or rather child forms—the forms are drawn *underneath* the components that appear on the main form, much like placing stickers on the glass of a picture frame (see Figure 16.4).

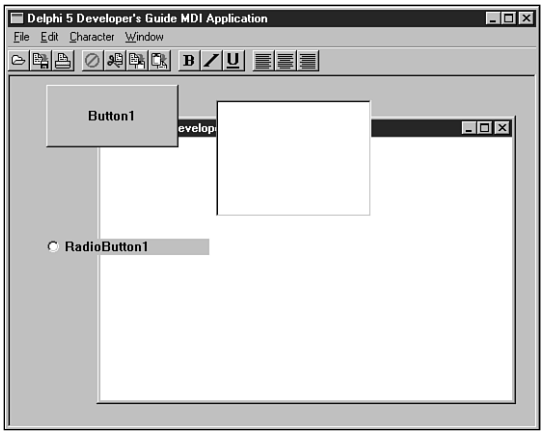

FIGURE 16.4
Client forms drawn underneath the main form's components.

So how do you go about drawing on the client window? Because Delphi 5 doesn't provide a VCL encapsulation of the client window, you must use the Win32 API. The method used is to subclass the client window and capture the message responsible for painting the client window's background—`WM_ERASEBKGND`. There, you take over the default behavior and perform your own custom drawing.

The following code is from the project `MdiBknd.dpr` on the CD. This project is an MDI application with a `TImage` component that contains a bitmap. From the menu, you can specify how to draw the image on the MDI client window—centered, tiled, or stretched, as shown respectively in Figures 16.5, 16.6, and 16.7.

FIGURE 16.5
The MDI client window with a centered image.

FIGURE 16.6
The MDI client window with a tiled image.

FIGURE 16.7
The MDI client window with a stretched image.

Listing 16.7 shows the unit code that performs the drawing.

LISTING 16.7 Drawing Images on the MDI Client Window

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, JPeg;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiImage: TMenuItem;
    mmiTile: TMenuItem;
    mmiCenter: TMenuItem;
    mmiStretch: TMenuItem;
    imgMain: TImage;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiTileClick(Sender: TObject);
```

```
  private
    FOldClientProc,
    FNewClientProc: TFarProc;
    FDrawDC: hDC;
    procedure CreateMDIChild(const Name: string);
    procedure ClientWndProc(var Message: TMessage);
    procedure DrawStretched;
    procedure DrawCentered;
    procedure DrawTiled;
  protected
    procedure CreateWnd; override;
  end;

var
  MainForm: TMainForm;

implementation

uses MdiChildFrm;

{$R *.DFM}

procedure TMainForm.CreateWnd;
begin
  inherited CreateWnd;
  // Turn the ClientWndProc method into a valid window procedure
  FNewClientProc := MakeObjectInstance(ClientWndProc);
  // Get a pointer to the original window procedure
  FOldClientProc := Pointer(GetWindowLong(ClientHandle, GWL_WNDPROC));
  // Set ClientWndProc as the new window procedure
  SetWindowLong(ClientHandle, GWL_WNDPROC, LongInt(FNewClientProc));
end;

procedure TMainForm.DrawCentered;
{ This procedure centers the image on the form's client area }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
   with imgMain do
     BitBlt(FDrawDC, ((CR.Right - CR.Left) - Picture.Width) div 2,
            ((CR.Bottom - CR.Top) - Picture.Height) div 2,
            Picture.Graphic.Width, Picture.Graphic.Height,
            Picture.Bitmap.Canvas.Handle, 0, 0, SRCCOPY);
end;
```

continues

LISTING 16.7 Continued

```
procedure TMainForm.DrawStretched;
{ This procedure stretches the image on the form's client area }
var
  CR: TRect;
begin
  GetWindowRect(ClientHandle, CR);
  StretchBlt(FDrawDC, 0, 0, CR.Right, CR.Bottom,
             imgMain.Picture.Bitmap.Canvas.Handle, 0, 0,
             imgMain.Picture.Width, imgMain.Picture.Height, SRCCOPY);
end;

procedure TMainForm.DrawTiled;
{ This procedure tiles the image on the form's client area }
var
  Row, Col: Integer;
  CR, IR: TRect;
  NumRows, NumCols: Integer;
begin
  GetWindowRect(ClientHandle, CR);
  IR := imgMain.ClientRect;
  NumRows := CR.Bottom div IR.Bottom;
  NumCols := CR.Right div IR.Right;
  with imgMain do
    for Row := 0 to NumRows+1 do
      for Col := 0 to NumCols+1  do
        BitBlt(FDrawDC, Col * Picture.Width, Row * Picture.Height,
               Picture.Width, Picture.Height, Picture.Bitmap.Canvas.Handle,
               0, 0, SRCCOPY);
end;

procedure TMainForm.ClientWndProc(var Message: TMessage);
begin
  case Message.Msg of
    // Capture the WM_ERASEBKGND messages and perform the client area drawing
    WM_ERASEBKGND:
      begin
        CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
Message.wParam,
          Message.lParam);
        FDrawDC := TWMEraseBkGnd(Message).DC;
        if mmiStretch.Checked then
          DrawStretched
        else if mmiCenter.Checked then
          DrawCentered
        else DrawTiled;
```

```
        Message.Result := 1;
      end;
    { Capture the scrolling messages and ensure the client area
      is redrawn by calling InvalidateRect }
    WM_VSCROLL, WM_HSCROLL:
      begin
        Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
Message.Msg,
          Message.wParam, Message.lParam);
        InvalidateRect(ClientHandle, nil, True);
      end;
    else
    // By Default, call the original window procedure
      Message.Result := CallWindowProc(FOldClientProc, ClientHandle,
Message.Msg,
        Message.wParam, Message.lParam);
  end; { case }
end;

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

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

procedure TMainForm.mmiTileClick(Sender: TObject);
begin
  mmiTile.Checked := false;
```

continues

LISTING 16.7 Continued

```
  mmiCenter.Checked := False;
  mmiStretch.Checked := False;
  { Set the Checked property for the menu item which invoked }
  { this event handler to Checked                            }
  if Sender is TMenuItem then
    TMenuItem(Sender).Checked := not TMenuItem(Sender).Checked;
  { Redraw the client area of the form }
  InvalidateRect(ClientHandle, nil, True);
end;

end.
```

To paint the image to the client window of the MDI application, you must use a technique called *subclassing.* Subclassing is discussed in Chapter 5, "Understanding Messages." To subclass the client window, you must store the client window's original window procedure so that you can call it. You must also have a pointer to the new window procedure. The form variable `FOldClientProc` stores the original window procedure, and the variable `FNewClientProc` points to the new window procedure.

The procedure `ClientWndProc()` is the procedure to which `FNewClientProc` points. Actually, because `ClientWndProc()` is a method of `TMainForm`, you must use the `MakeObjectInstance()` function to return a pointer to a window procedure created from the method `MakeObjectInstance()`, as discussed in Chapter 13, "Hard-core Techniques."

The `TMainForm.CreateWnd()` method was overridden when the main form's client window was subclassed by using the `GetWindowLong()` and `SetWindowLong()` Win32 API functions. `ClientWndProc()` is the new window procedure.

`TMainForm` contains three private methods: `DrawCentered()`, `DrawTiled()`, and `DrawStretched()`. Each of these methods uses Win32 API functions to perform the GDI drawing routines to paint the bitmap. Win32 API functions are used because the client window's device context isn't encapsulated by `TCanvas`, so you can't normally use the built-in Delphi 5 methods. Actually, it's possible to assign the device context to a `TCanvas.Handle` property. You would have to instantiate a `TCanvas` instance in order to do this, but it is possible.

You must capture three messages to perform the background drawing: `WM_ERASEBKGND`, `WM_VSCROLL`, and `WM_HSCROLL`. The `WM_ERASEBKGND` message is sent to a window when it's to be erased. This is an opportune time to perform the specialized drawing of the image. In the procedure, you determine which drawing procedure to call based on which menu item is selected. The `WM_VSCROLL` and `WM_HSCROLL` messages are captured to ensure that the background image

is properly drawn when the user scrolls the main form. Finally, all other messages are sent to the original window procedure with this statement:

```
Message.Result := CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
        Message.wParam, Message.lParam);
```

This example not only demonstrates how you can visually enhance your applications; it also shows how you can perform API-level development with techniques not provided by the VCL.

Creating a Hidden MDI Child Form

Delphi 5 returns an error if you ever attempt to hide an MDI child form using a statement such as this one:

```
ChildForm.Hide;
```

The error indicates that hiding an MDI child form is not allowed. The reason for this is because the Delphi developers found that in the Windows implementation of MDI, hiding MDI child forms corrupts the z-order of the child windows. Unless you're extremely careful about when you use such a technique, trying to hide an MDI child form can wreak havoc with your application. Nevertheless, you might have the need to hide a child form. There are two ways in which you can hide MDI child forms. Just be aware of the anomaly and use these techniques with caution.

One way to hide an MDI child form is to prevent the client window from drawing the child form altogether. Do this by using the `LockWindowUpdate()` Win32 API function to disable drawing to the MDI client window. This technique is useful if you want to create an MDI child form but don't want to show that form to the user unless some process has completed successfully. For example, such a process might be a database query; if the process fails, you might want to free the form. Unless you use some method to hide the form, you'll see a flicker on the screen as the form is created before you have an opportunity to destroy it. The `LockWindowUpdate()` function disables drawing to a window's canvas. Only one window can be locked at any given time. Passing `0` to `LockWindowUpdate` reenables drawing to the window's canvas.

The other method of hiding an MDI child form is to actually hide the child form by using the Win32 API function `ShowWindow()`. You hide the form by specifying the `SW_HIDE` flag along with the function. You must then use the `SetWindowPos()` function to restore the child window. You can use this technique to hide the MDI child form if it's already created and displayed to the user.

Listing 16.8 illustrates the techniques just described and is the main form for the project `MdiHide.dpr` on the CD.

LISTING 16.8 A Unit Showing MDI Child Form–Hiding Techniques

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, MdiChildFrm;

type
  TMainForm = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiHide: TMenuItem;
    mmiShow: TMenuItem;
    mmiHideForm: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiHideClick(Sender: TObject);
    procedure mmiShowClick(Sender: TObject);
    procedure mmiHideFormClick(Sender: TObject);
  private
    procedure CreateMDIChild(const Name: string);
  public
    HideForm: TMDIChildForm;
    Hidden: Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
```

```
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

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

procedure TMainForm.mmiHideClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    ShowWindow(HideForm.Handle, SW_HIDE);
  Hidden := True;
end;

procedure TMainForm.mmiShowClick(Sender: TObject);
begin
  if Assigned(HideForm) then
    SetWindowPos(HideForm.handle, HWND_TOP, 0, 0, 0, 0, SWP_NOSIZE
       or SWP_NOMOVE or SWP_SHOWWINDOW);
  Hidden := False;
end;

procedure TMainForm.mmiHideFormClick(Sender: TObject);
begin
  if not Assigned(HideForm) then
  begin
    if MessageDlg('Create Hidden?', mtConfirmation, [mbYes, mbNo], 0) = mrYes
then
    begin
       LockWindowUpdate(Handle);
      try
        HideForm := TMDIChildForm.Create(Application);
        HideForm.Caption := 'HideForm';
        ShowMessage('Form created and hidden. Press OK to show form');
      finally
```

continues

LISTING 16.8 Continued

```
        LockWindowUpdate(0);
      end;
    end
    else begin
      HideForm := TMDIChildForm.Create(Application);
      HideForm.Caption := 'HideForm';
    end;
  end
  else if not Hidden then
      HideForm.SetFocus;
end;

end.
```

The project is a simple MDI application. The event handler `mmiHideFormClick()` creates a child form that can either be created and hidden or hidden by the user after it's displayed.

When `mmiHideFormClick()` is invoked, it checks whether an instance of `THideForm` has been created. If so, it displays only the `THideForm` instance, provided that it has not been hidden by the user. If there is no instance of `THideForm` present, the user is prompted whether it should be created and hidden. If the user responds affirmatively, drawing to the client window is disabled before the form is created. If drawing to the client window is not disabled, the form is displayed as it's created. The user is then shown a message box indicating that the form is created. When the user closes the message box, drawing to the client window is reenabled and the child form is displayed by forcing the client window to repaint itself. You can replace the message box telling the user that the form is created with some lengthy process that requires the child form to be created but not displayed. If the user chooses not to create the form as hidden, it's created normally.

The second method used to hide the form after it has already been displayed calls the Win32 API function `ShowWindow()` and passes the child form's handle and the `SW_HIDE` flag. This effectively hides the form. To redisplay the form, call the Win32 API function `SetWindowPos()`, using the child form's handle and the flags specified in the listing. `SetWindowPos()` is used to change a window's size, position, or z-order. In this example, `SetWindowPos()` is used to redisplay the hidden window by setting its z-order; in this case, the z-order of the hidden form is set to be the top window by specifying the `HWND_TOP` flag.

Minimizing, Maximizing, and Restoring All MDI Child Windows

Often, you need to perform a task across all active MDI forms in the project. Changing the form's `WindowState` property is a typical example of a process to be performed on every

instance of an MDI child form. This task is quite simple and only requires that you walk through the forms using the main form's `MDIChildren` array property. The main form's `MDIChildren` property holds the number of active MDI child forms. Listing 16.9 shows the event handlers that minimize, maximize, and restore all MDI child windows in an application. This project can be found on the CD as the `Min_Max.dpr` project.

LISTING 16.9 Minimizing, Maximizing, and Restoring All MDI Child Forms

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
  StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
    mmiClose: TMenuItem;
    mmiWindow: TMenuItem;
    N1: TMenuItem;
    mmiExit: TMenuItem;
    mmiMinimizeAll: TMenuItem;
    mmiMaximizeAll: TMenuItem;
    mmiRestoreAll: TMenuItem;
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiMinimizeAllClick(Sender: TObject);
    procedure mmiMaximizeAllClick(Sender: TObject);
    procedure mmiRestoreAllClick(Sender: TObject);
  private
    { Private declarations }
    procedure CreateMDIChild(const Name: string);
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses MdiChildFrm;
```

continues

LISTING 16.9 Continued

```
{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  Child: TMDIChildForm;
begin
  Child := TMDIChildForm.Create(Application);
  Child.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('NONAME' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

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

procedure TMainForm.mmiMinimizeAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := MDIChildCount - 1 downto 0 do
    MDIChildren[i].WindowState := wsMinimized;
end;

procedure TMainForm.mmiMaximizeAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsMaximized;
end;

procedure TMainForm.mmiRestoreAllClick(Sender: TObject);
var
```

```
  i: integer;
begin
  for i := 0 to MDIChildCount - 1 do
    MDIChildren[i].WindowState := wsNormal;
end;

end.
```

Summary

This chapter showed you how to build MDI applications in Delphi 5. You also learned some advanced techniques specific to MDI applications. With the foundation you received in this chapter, you should be well on your way to creating professional-looking MDI applications.

Multimedia Programming with Delphi

CHAPTER

18

IN THIS CHAPTER

Delphi's `TMediaPlayer` component is proof that good things come in small packages. In the guise of this little component, Delphi encapsulates a great deal of the functionality of the Windows *Media Control Interface* (MCI)—the portion of the Windows API that provides control for multimedia devices.

Delphi makes multimedia programming so easy that the traditional and boring "Hello World" program may be a thing of the past. Why write `Hello World` to the screen when it's almost as easy to play a sound or video file that offers its greetings?

In this chapter, you learn how to write a simple yet powerful media player, and you even construct a fully functional audio CD player. This chapter explains the uses and nuances of the `TMediaPlayer` component. Of course, your computer must be equipped with multimedia devices, such as a sound card and CD-ROM, for this chapter to be of real use to you.

Creating a Simple Media Player

The best way to learn is by doing. This application demonstrates how quickly you can create a media player by placing `TMediaPlayer`, `TButton`, and `TOpenDialog` components on a form. This form is shown in Figure 18.1.

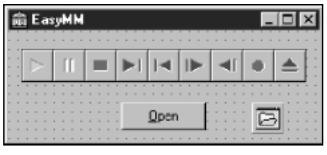

FIGURE 18.1
The `EasyMM` *Media Player.*

The `EasyMM` Media Player works like this: After you click `Button1`, the `OpenDialog` dialog box appears, and you choose a file from it. The Media Player prepares itself to play the file you chose in `OpenDialog`. You then can click the Play button on the Media Player to play the file. The following code belongs to the button's `OnClick` method, and it opens the Media Player with the file you chose:

```
procedure TMainForm.BtnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;
```

This code executes the `OpenDialog1` dialog box, and if a filename is chosen, `OpenDialog1`'s `FileName` property is copied to `MediaPlayer1`'s `FileName` property. The `MediaPlayer`'s `Open` method is then called to prepare it to play the file.

You might also want to limit the files to browse through with the `OpenDialog` dialog box to only multimedia files. `TMediaPlayer` supports a whole gaggle of multimedia device types, but for now, you'll just browse WAV, AVI, and MIDI files. This capability exists in the `TOpenDialog` component, and you take advantage of it by selecting `OpenDialog1` in the Object Inspector, choosing the `Mask` property, and clicking the ellipsis to the right of this item to invoke the Filter Editor. Fill in the `.WAV`, `.AVI`, and `.MID` extensions, as shown in Figure 18.2.

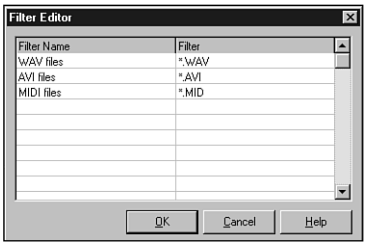

FIGURE 18.2
The Filter Editor.

The project is saved as `EasyMM.dpr` and the main unit as `Main.pas`. The Media Player is now ready to run. Run the program, and try it out using one of the multimedia files on your hard disk. Other people might have convinced you—or perhaps you had convinced yourself—that multimedia programming is difficult, but now you have firsthand proof that this just isn't true.

Using WAV Files in Your Applications

WAV files (pronounced *wave*, which is short for *waveform*) are the standard file format for sharing audio in Windows. As the name implies, WAV files store sounds in a binary format that resembles a mathematical wave. The great thing about WAV files is that they have gained industry acceptance, and you can find them everywhere. The bad thing about WAV files is that they tend to be bulky, and just a few of those Homer Simpson WAV files can take up a hefty chunk of hard disk space.

The `TMediaPlayer` component enables you to easily integrate WAV sounds into your applications. As just illustrated, playing WAV files in your program is no sweat—just feed a `TMediaPlayer` component a filename, open it, and play it. A little audio capability can be just the thing your applications need to go from neat to way cool.

If playing WAV files is all you want to do, you might not need the overhead of a `TMediaPlayer` component. Instead, you can use the `PlaySound()` API function found in the `MMSystem` unit. `PlaySound()` is defined this way:

```
function PlaySound(pszSound: PChar; hmod: HMODULE;
  fdwSound: DWORD): BOOL; stdcall;
```

`PlaySound()` has the capability to play a WAV sound from a file, from memory, or from a resource file linked into the application. `PlaySound()` accepts three parameters:

- The first parameter, `pszSound`, is a `PChar` variable that represents a filename, alias name, resource name, Registry entry, entry from the `[sounds]` section of your `WIN.INI` file, or pointer to a WAV sound located somewhere in memory.
- The second parameter, `hmod`, represents the handle of the executable file that contains the resource to be loaded. This parameter must be zero unless `snd_Resource` is specified in the `fdwSound` parameter.
- The third parameter, `fdwSound`, contains flags that describe how the sound should be played. These flags can contain a combination of any of the following values:

Flag	*Description*
`SND_APPLICATION`	The sound is played using an application-specific association.
`SND_ALIAS`	The `pszSound` parameter is a system-event alias in the Registry or the `WIN.INI` file. Don't use this flag with either `SND_FILENAME` or `SND_RESOURCE`, because they're mutually exclusive.
`SND_ALIAS_ID`	The `pszSound` parameter is a predefined sound identifier.
`SND_FILENAME`	The `pszSound` parameter is a filename.
`SND_NOWAIT`	This flag indicates that if the driver is busy, it returns immediately without playing the sound.
`SND_PURGE`	All sounds are stopped for the calling task. If `pszSound` is not zero, all instances of the specified sound are stopped. If `pszSound` is zero, all sounds invoked by the current task are stopped. You must also specify the proper instance handle to stop `SND_RESOURCE` events.
`SND_RESOURCE`	The `pszSound` parameter is a resource identifier. When you're using this flag, the `hmod` parameter must contain the instance that contains the specified resource.
`SND_ASYNC`	Plays the sound asynchronously and returns the function almost immediately. This achieves a background music effect.
`SND_LOOP`	Plays the sound over and over until you make it stop or you go insane. `SND_ASYNC` also must be specified when using this flag.

`SND_MEMORY`	Plays the WAV sound in the memory area pointed to by the `pszSound` parameter.
`SND_NODEFAULT`	If the sound can't be found, `PlaySound()` returns immediately without playing the default sound, as specified in the Registry.
`SND_NOSTOP`	Plays the sound only if it isn't already playing. `PlaySound()` returns `True` if the sound is played and `False` if the sound is not played. If this flag is not specified, Win32 will stop any currently playing sound before attempting to play the sound specified in `pszSound`.
`SND_SYNC`	Plays the sound synchronously and doesn't return from the function until the sound finishes playing.

TIP

To terminate a WAV sound currently playing asynchronously, call `PlaySound()` and pass `Nil` or zero for all parameters, as follows:

```
PlaySound(Nil, 0, 0);  // stop currently playing WAV
```

To terminate even non-waveform sounds for a given task, add the `snd_Purge` flag:

```
PlaySound(Nil, 0, snd_Purge);  // stop all currently playing sounds
```

NOTE

The Win32 API still supports the `sndPlaySound()` function, which was a part of the Windows 3.*x* API. This function is only supported for backward compatibility, however, and it might not be available in future implementations of the Win32 API. Use the Win32 `PlaySound()` function rather than `sndPlaySound()` for future compatibility.

Playing Video

AVI (short for *audio-video interleave)* is one of the most common file formats used to exchange audio and video information simultaneously. In fact, you'll find a couple of AVI files in the `\Runimage\Delphi50\Demos\Coolstuf` directory of the CD-ROM that contains your copy of Delphi 5.

You can use the simple multimedia player program you wrote earlier in this chapter to display AVI files. Simply select an AVI file when `OpenDialog1` is invoked and click the Play button. Note that the AVI file plays in its own window.

Showing the First Frame

You might want to display the first frame of an AVI file in a window before you actually play the file. This achieves a sort of freeze-frame effect. To do this after opening the `TMediaPlayer` component, just set the `Frames` property of `TMediaPlayer` to `1` and then call the `Step()` method. The `Frames` property tells `TMediaPlayer` how many frames to move when `Step()` and `Back()` methods are called. `Step()` advances the `TMediaPlayer` frames and displays the current frame. This is the code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      Frames := 1;
      Step;
      Notify := True;
    end;
end;
```

Using the Display Property

You can assign a value to `TMediaPlayer`'s `Display` property to cause the AVI file to play to a specific window, instead of creating its own window. To do this, you add a `TPanel` component to your Media Player, as shown in Figure 18.3. After adding the panel, you can save the project in a new directory as `DDGMPlay.dpr`.

FIGURE 18.3
The `DDGMPlay` *main window.*

Click the drop-down arrow button for `MediaPlayer1`'s `Display` property and notice that all the components in this project appear in the list box. Set the value of the `Display` property to `Panel1`.

Now notice that when you run the program and select and play an AVI file, the AVI file output appears in the panel. Also notice that the AVI file doesn't take up the whole area of the panel; the AVI file has a certain default size programmed into it.

Using the DisplayRect Property

`DisplayRect` is a property of type `TRect` that determines the size of the AVI file output window. You can use the `DisplayRect` property to cause your AVI file's output to stretch or shrink to a certain size. If you want the AVI file to take up the whole area of `Panel1`, for example, you can assign `DisplayRect` to the size of the panel:

```
MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
```

You can add this line of code to the `OnClick` handler for `Button1`, like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
  end;
end;
```

CAUTION

You can set the `DisplayRect` property only after the `TMediaPlayer`'s `Open()` method is called.

Understanding TMediaPlayer Events

`TMediaPlayer` has two unique events: `OnPostClick` and `OnNotify`.

The `OnPostClick` event is very similar to `OnClick`, but `OnClick` occurs as soon as the component is clicked, and `OnPostClick` executes only after some action occurs that was caused by a click. If you click the Play button on `TMediaPlayer` at runtime, for example, an `OnClick` event is generated, but an `OnPostClick` event is generated only after the media device is done playing.

The `OnNotify` event is a little more interesting. The `OnNotify` event executes whenever the `TMediaPlayer` completes a media-control method (such as `Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `Resume`, `Rewind`, `StartRecording`, `Step`, or `Stop`) and only when `TMediaPlayer`'s `Notify` property is set to `True`. To illustrate `OnNotify`, add a handler for this event to the DDGMPlay project. In the event handler method, you cause a message dialog box to appear after a command executes:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg('Media control method executed', mtInformation, [mbOk], 0);
end;
```

Don't forget to also set the `Notify` property to `True` in `Button1`'s `OnClick` handler after opening the Media Player:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;
```

TIP

Notice that you moved the code dealing with `MediaPlayer1` into a `with..do` construct. As you learned in earlier chapters, this construct offers advantages in code clarity and performance over simply qualifying each property and method name.

Viewing the Source Code for DDGMPlay

By now, you should know the basics of how to play WAV and AVI files. Listings 18.1 and 18.2 show the complete source code for the `DDGMPlay` project.

LISTING 18.1 The Source Code for DDGMPlay.dpr

```
program DDGMPlay;

uses
  Forms,
  Main in 'MAIN.PAS' {MainForm};

{$R *.RES}

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

LISTING 18.2 The Source Code for Main.pas

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    MediaPlayer1: TMediaPlayer;
    OpenDialog1: TOpenDialog;
    Button1: TButton;
    Panel1: TPanel;
    procedure Button1Click(Sender: TObject);
    procedure MediaPlayer1Notify(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;

procedure TMainForm.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg('Media control method executed', mtInformation, [mbOk], 0);
```

continues

LISTING 18.2 Continued

```
end;

end.
```

Device Support

`TMediaPlayer` supports the vast array of media devices supported by MCI. The type of device that `TMediaPlayer` controls is determined by its `DeviceType` property. Table 18.1 describes the different values of the `DeviceType` property.

TABLE 18.1 Values of `TMediaPlayer`'s `DeviceType` Property

DeviceType Value	*Media Device*
`dtAutoSelect`	The `TMediaPlayer` automatically should select the correct device type based on the filename to be played.
`dtAVIVideo`	AVI file. These files have the `.AVI` extension and contain both sound and full-motion video.
`dtCDAudio`	An audio CD played from your computer's CD-ROM drive.
`dtDAT`	A digital audio tape (DAT) player connected to your PC.
`dtDigitalVideo`	A digital video device, such as a digital video camera.
`dtMMMovie`	A multimedia movie format.
`dtOther`	An unspecified multimedia format.
`dtOverlay`	A video overlay device.
`dtScanner`	A scanner connected to your PC.
`dtSequencer`	A sequencer device capable of playing MIDI files. MIDI files typically end in a `.MID` or `.RMI` extension.
`dtVCR`	A video cassette recorder (VCR) connected to your PC.
`dtVideodisc`	A video disc player connected to your PC.
`dtWaveAudio`	A WAV audio file. These files end in the `.WAV` extension.

Although you can see that `TMediaPlayer` supports many formats, this chapter focuses primarily on the WAV, AVI, and CD Audio formats because those are the most common under Windows.

NOTE

The `TMediaPlayer` component is a `TWinControl` descendant, which means it can be easily encapsulated as an ActiveX control through the Delphi 5 wizards. One possible

benefit of doing this is the ability to embed a Media Player in a Web page to extend your pages with custom multimedia. Additionally, with a few lines of JavaScript or VBScript, you could provide a CD player for everyone on the Internet or your intranet running a Windows browser.

Creating an Audio CD Player

You'll learn about the finer points of the `TMediaPlayer` component by creating a full-featured audio CD player. Figure 18.4 shows the main form for this application, which is called `CDPlayer.dpr`. The main unit for this form is called `CDMain.pas`.

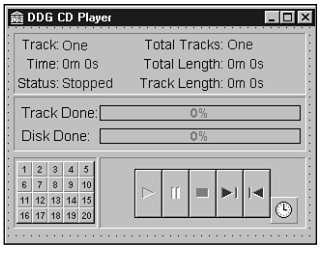

FIGURE 18.4
The audio CD player's main form.

Table 18.2 shows the important properties to be set for the components contained on the CD player's main form.

TABLE 18.2 Important Properties for the CD Player's Components

Component	*Property*	*Value*
`mpCDPlayer`	`DeviceType`	`dtAudioCD`
`sbTrack1-sbTrack20`	`Caption`	`'1' - '20'`
`sbTrack1-sbTrack20`	`Tag`	`1 - 20`

Displaying a Splash Screen

When the CD player is run, it takes a couple seconds for it to load, and it might take several more seconds for the `TMediaPlayer` component to initialize after calling its `Open()` method. This delay from the time the user clicks the icon in Explorer to the time he or she actually sees the program often gives the user an "is my program gonna start, or isn't it?" feeling. This delay is caused by the time Windows takes to load its multimedia subsystem, which occurs

when `TMediaPlayer` is opened. To avoid this problem, you can give the CD player program a splash screen that displays as the program starts. The splash screen tells users that, yes, the program will eventually start—it's just taking a moment to load, so enjoy this little screen in the meantime.

The first step in creating a splash screen is to create a form that you want to use as the splash screen. Generally, you want this form to contain a panel but not a border or title bar; this gives it a 3D, floating-panel appearance. On the panel, place one or more `TLabel` components and perhaps a `TImage` component that displays a bitmap or icon.

The splash screen form for the CD player is shown in Figure 18.5, and the unit, `Splash.pas`, is shown in Listing 18.3.

FIGURE 18.5
The CD player's splash screen form.

LISTING 18.3 The Source Code for SPLASH.PAS

```
unit Splash;
interface

uses Windows, Classes, Graphics, Forms, Controls, StdCtrls,
  ExtCtrls;

type
  TSplashScreen = class(TForm)
    StatusPanel: TPanel;
  end;

var
  SplashScreen: TSplashScreen;

implementation

{$R *.DFM}

begin
  { Since the splash screen is displayed before the main screen is created,
    it must be created before the main screen. }
  SplashScreen := TSplashScreen.Create(Application);
  SplashScreen.Show;
```

```
  SplashScreen.Update;
end.
```

Unlike a normal form, the splash screen is created and shown in the `initialization` section of its unit. Because the `initialization` section for all units is executed before the main program block in the DPR file, this form is displayed before the main portion of the program runs.

CAUTION

Do *not* use `Application.CreateForm()` to create your splash screen form instance. The first time `Application.CreateForm()` is called in an application, Delphi makes that form the main application form. It would be a "bad thing" to make your splash screen the main form.

Beginning the CD Player

Create an event handler for the form's `OnCreate` method. In this method, you open and initialize the CD player program. First, call `CDPlayer`'s `Open()` method. `Open()` checks to make sure that the system is capable of playing audio CDs and then initializes the device. If `Open()` fails, it raises an exception of type `EMCIDeviceError`. In the event of an exception opening the device, you should terminate the application. Here's the code:

```
try
  mpCDPlayer.Open; { Open the CD Player device. }
except
  { If an error occurred, the system may be incapable of playing CDs. }
  on EMCIDeviceError do
  begin
    MessageDlg('Error Initializing CD Player.  Program will now exit.',
               mtError, [mbOk], 0);
    Application.Terminate;   { bail out }
  end;
end;
```

NOTE

The preferred way to end a Delphi application is by calling the main form's `Close()` method or by calling `Application.Terminate`.

After opening `CDPlayer`, you should set its `EnabledButtons` property to ensure that the proper buttons are enabled for the device. Which buttons to enable, however, depends on the current state of the CD device. If a CD is already playing when you call `Open()`, for example, you obviously don't want to enable the Play button. To perform a check on the current status of the CD device, you can inspect `CDPlayer`'s `Mode` property. The `Mode` property, which has all its possible values laid out nicely in the online help, provides information on whether a CD device is currently playing, stopped, paused, seeking, and so on. In this case, your concern is only whether the device is stopped, paused, or playing. The following code enables the proper buttons:

```
case mpCDPlayer.Mode of
  mpPlaying: mpCDPlayer.EnabledButtons := [btPause, btStop, btNext, btPrev];
  mpStopped,        { show default buttons if stopped }
  mpPaused : mpCDPlayer.EnabledButtons := [btPlay, btNext, btPrev];
end;
```

The following is the completed source code for the `TMainForm.FormCreate()` method. Notice that you make calls to several methods after successfully opening `CDPlayer`. The purpose of these methods is to update various aspects of the CD player application, such as the number of tracks on the current CD and the current track position. (These methods are described in more detail later in this chapter.) Here's the code:

```
procedure TMainForm.FormCreate(Sender: TObject);
{ This method is called when the form is created. It opens and initializes the
  player }
begin
  try
    mpCDPlayer.Open;    // Open the CD Player device.
    { If a CD is already playing at startup, show playing status. }
    if mpCDPlayer.Mode = mpPlaying then
      LblStatus.Caption := 'Playing';
    GetCDTotals;      // Show total time and tracks on current CD
    ShowTrackNumber;  // Show current track
    ShowTrackTime;    // Show the minutes and seconds for the current track
    ShowCurrentTime;  // Show the current position of the CD
    ShowPlayerStatus; // Update the CD Player's status
  except
    { If an error occurred, the system may be incapable of playing CDs. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player.  Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
  { Check the current mode of the CD-ROM and enable the appropriate buttons. }
  case mpCDPlayer.Mode of
```

```
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release;  // Close and free the splash screen
end;
```

Notice that the last line of code in this method closes the splash screen form. The `OnCreate` event of the main form is generally the best place to do this.

Updating the CD Player Information

As the CD device plays, you can keep the information on `CDPlayerForm` up-to-date by using a `TTimer` component. Every time a timer event occurs, you can call the necessary updating methods, as shown in the form's `OnCreate` method, to ensure that the display stays current. Double-click `Timer1` to generate a method skeleton for its `OnTimer` event. Here's the source code you use for this event:

```
procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ This method is the heart of the CD Player.  It updates all information at
  every timer interval. }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
                            mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Show track number the CD player is currently on
    ShowTrackTime;   // Show total time for the current track
    ShowCurrentTime; // Show elapsed time for the current track
  end;
end;
```

Notice that, in addition to calling the various updating methods, this method also updates the `DiskDoneGauge` control for the amount of time elapsed on the current CD. To get the elapsed time, the method changes `CDPlayer`'s `TimeFormat` property to `tfMSF` and gets the minute and second value from the `Position` property by using the `mci_msf_Minute()` and `mci_msf_Second()` functions. This merits a bit more explanation.

TimeFormat

The `TimeFormat` property of a `TMediaPlayer` component determines how the values of the `StartPos`, `Length`, `Position`, `Start`, and `EndPos` properties should be interpreted. Table 18.3 lists the possible values for `TimeFormat`. These values represent information packed into a `Longint` type variable.

TABLE 18.3 Values for the `TMediaPlayer.TimeFormat` Property

Value	*Time Storage Format*
`tfBytes`	Number of bytes
`tfFrames`	Frames
`tfHMS`	Hours, minutes, and seconds
`tfMilliseconds`	Time in milliseconds
`tfMSF`	Minutes, seconds, and frames
`tfSamples`	Number of samples
`tfSMPTE24`	Hours, minutes, seconds, and frames based on 24 frames per second
`tfSMPTE25`	Hours, minutes, seconds, and frames based on 25 frames per second
`tfSMPTE30`	Hours, minutes, seconds, and frames based on 30 frames per second
`tfSMPTE30Drop`	Hours, minutes, seconds, and frames based on 30 drop frames per second
`tfTMSF`	Tracks, minutes, seconds, and frames

Time-Conversion Routines

The Windows API provides routines to retrieve the time information from the different packed formats shown in Table 18.4. *Packed format* means that multiple data values are packed (encoded) into one `Longint` value. These functions are located in `MMSystem.dll`, so be sure to have `MMSystem` in your `uses` clause when using them.

TABLE 18.4 Functions to Unpack Multimedia Time Formats

Function	*Works With*	*Returns*
`mci_HMS_Hour()`	`tfHMS`	Hours
`mci_HMS_Minute()`	`tfHMS`	Minutes
`mci_HMS_Second()`	`tfHMS`	Seconds
`mci_MSF_Frame()`	`tfMSF`	Frames
`mci_MSF_Minute()`	`tfMSF`	Minutes
`mci_MSF_Second()`	`tfMSF`	Seconds
`mci_TMSF_Frame()`	`tfTMSF`	Frames
`mci_TMSF_Minute()`	`tfTMSF`	Minutes
`mci_TMSF_Second()`	`tfTMSF`	Seconds
`mci_TMSF_Track()`	`tfTMSF`	Tracks

Methods for Updating the CD Player

As you learned earlier in this chapter, you use several methods to help keep the information displayed by the CD player up-to-date. The primary purpose of each of these methods is to update the labels in the top portion of the CD player form and to update the gauges in the middle portion of that form.

GetCDTotals()

The purpose of the `GetCDTotals()` method, shown in the following code, is to retrieve the length and total number of tracks on the current CD. This information is then used to update several labels and `DiskDoneGauge`. This code also calls the `AdjustSpeedButtons()` method, which enables the same number of speedbuttons as tracks. Notice that this method also makes use of the `TimeFormat` and time-conversion routines discussed earlier:

```
procedure TMainForm.GetCDTotals;
{ This method gets the total time and tracks of the CD and displays them. }
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF;                  // set time format
  TimeValue := mpCDPlayer.Length;                   // get CD length
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // get total tracks
  TotalLengthM := mci_msf_Minute(TimeValue);     // get total length in mins
  TotalLengthS := mci_msf_Second(TimeValue);     // get total length in secs
  { set caption of Total Tracks label }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { set caption of Total Time label }
  LblTotalLen.Caption := Format(MSFormatStr, [TotalLengthM, TotalLengthS]);
  { initialize gauge }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { enable the correct number of speed buttons }
  AdjustSpeedButtons;
end;
```

ShowCurrentTime()

The `ShowCurrentTime()` method is shown in the following code. This method is designed to obtain the elapsed minutes and seconds for the currently playing track as well as to update the necessary controls. Here, you also use the time-conversion routines provided by `MMSystem`:

```
procedure TMainForm.ShowCurrentTime;
{ This method displays the current time of the current track }
begin
  { Minutes for this track }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Seconds for this track }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
```

```
  { update track time label }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { update track gauge }
  ggTrackDone.Progress := (60 * m) + s;
end;
```

ShowTrackTime()

The `ShowTrackTime()` method, shown in the following code, obtains the total length of the current track in minutes and seconds, and it updates a label control. Again, you make use of the time-conversion routines. Also notice that you check to make sure that the track isn't the same as when this function was last called. This comparison ensures that you don't make unnecessary function calls or repaint components unnecessarily. Here's the code:

```
procedure TMainForm.ShowTrackTime;
{ This method changes the track time to display the total length of the
  currently selected track. }
var
  Min, Sec: Byte;
  Len: Longint;
begin
  { Don't update the information if player is still on the same track }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;
```

CD Player Source

You've now seen all aspects of the CD player as they relate to multimedia. Listings 18.4 and 18.5 show the complete source code for the `CDPlayer.dpr` and `CDMain.pas` modules. The `CDMain` unit also shows some of the techniques you use to manipulate the speedbuttons using their `Tag` properties as well as other techniques for updating the controls.

LISTING 18.4 The Source Code for CDPlayer.dpr

```
program CDPlayer;

uses
  Forms,
  Splash in 'Splash.pas' {SplashScreen},
```

```
  CDMain in 'CDMain.pas' {MainForm};

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

LISTING 18.5 The Source Code for CDMain.pas

```
unit CDMain;

interface

uses
  SysUtils, Windows, Classes, Graphics, Forms, Controls, MPlayer, StdCtrls,
  Menus, MMSystem, Messages, Buttons, Dialogs, ExtCtrls, Splash, Gauges;

type
  TMainForm = class(TForm)
    tmUpdateTimer: TTimer;
    MainScreenPanel: TPanel;
    LblStatus: TLabel;
    Label2: TLabel;
    LblCurTrk: TLabel;
    Label4: TLabel;
    LblTrackTime: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    LblTotTrk: TLabel;
    LblTotalLen: TLabel;
    Label12: TLabel;
    LblTrackLen: TLabel;
    Label15: TLabel;
    CDInfo: TPanel;
    SBPanel: TPanel;
    Panel1: TPanel;
    mpCDPlayer: TMediaPlayer;
    sbTrack1: TSpeedButton;
    sbTrack2: TSpeedButton;
    sbTrack3: TSpeedButton;
    sbTrack4: TSpeedButton;
    sbTrack5: TSpeedButton;
    sbTrack6: TSpeedButton;
    sbTrack7: TSpeedButton;
    sbTrack8: TSpeedButton;
    sbTrack9: TSpeedButton;
```

continues

LISTING 18.5 Continued

```
    sbTrack10: TSpeedButton;
    sbTrack11: TSpeedButton;
    sbTrack12: TSpeedButton;
    sbTrack13: TSpeedButton;
    sbTrack14: TSpeedButton;
    sbTrack15: TSpeedButton;
    sbTrack16: TSpeedButton;
    sbTrack17: TSpeedButton;
    sbTrack18: TSpeedButton;
    sbTrack19: TSpeedButton;
    sbTrack20: TSpeedButton;
    ggTrackDone: TGauge;
    ggDiskDone: TGauge;
    Label1: TLabel;
    Label3: TLabel;
    procedure tmUpdateTimerTimer(Sender: TObject);
    procedure mpCDPlayerPostClick(Sender: TObject; Button: TMPBtnType);
    procedure FormCreate(Sender: TObject);
    procedure sbTrack1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
    OldTrack, CurrentTrack: Byte;
    m, s: Byte;
    TotalTracks: Byte;
    TotalLengthM: Byte;
    TotalLengthS: Byte;
    procedure GetCDTotals;
    procedure ShowTrackNumber;
    procedure ShowTrackTime;
    procedure ShowCurrentTime;
    procedure ShowPlayerStatus;
    procedure AdjustSpeedButtons;
    procedure HighlightTrackButton;
    function TrackNumToString(InNum: Byte): String;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
```

```
  { Array of strings representing numbers from one to twenty: }
  NumStrings: array[1..20] of String[10] =
      ('One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine',
       'Ten', 'Eleven', 'Twelve', 'Thirteen', 'Fourteen', 'Fifteen', 'Sixteen',
       'Seventeen', 'Eighteen', 'Nineteen', 'Twenty');
  MSFormatStr = '%dm %ds';
  PlayButtons: TButtonSet = [btPause, btStop, btNext, btPrev];
  StopButtons: TButtonSet = [btPlay, btNext, btPrev];

function TMainForm.TrackNumToString(InNum: Byte): String;
{ This function returns a string corresponding to a integer between 1 and 20.
  If the number is greater than 20, then the integer is returned as a string. }
begin
  if (InNum > High(NumStrings)) or (InNum < Low(NumStrings)) then
    Result := IntToStr(InNum)    { if not in array, then just return number }
  else
    Result := NumStrings[InNum]; { return the string from NumStrings array }
end;

procedure TMainForm.AdjustSpeedButtons;
{ This method enables the proper number of speed buttons }
var
  i: integer;
begin
  { iterate through form's Components array... }
  for i := 0 to SBPanel.ControlCount - 1 do
    if SBPanel.Controls[i] is TSpeedButton then    // is it a speed button?
      { disable buttons higher than number of tracks on CD }
      with TSpeedButton(SBPanel.Controls[i]) do Enabled := Tag <= TotalTracks;
end;


procedure TMainForm.GetCDTotals;
{ This method gets the total time and tracks of the CD and displays them. }
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF;                  // set time format
  TimeValue := mpCDPlayer.Length;                   // get CD length
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // get total tracks
  TotalLengthM := mci_msf_Minute(TimeValue);     // get total length in mins
  TotalLengthS := mci_msf_Second(TimeValue);     // get total length in secs
  { set caption of Total Tracks label }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { set caption of Total Time label }
  LblTotalLen.Caption := Format(MSFormatStr, [TotalLengthM, TotalLengthS]);
```

continues

LISTING 18.5 Continued

```
  { intitialize gauge }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { enable the correct number of speed buttons }
  AdjustSpeedButtons;
end;

procedure TMainForm.ShowPlayerStatus;
{ This method displays the status of the CD Player and the CD that
  is currently being played. }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
    with LblStatus do
    begin
      case mpCDPlayer.Mode of
        mpNotReady: Caption := 'Not Ready';
        mpStopped:  Caption := 'Stopped';
        mpSeeking:  Caption := 'Seeking';
        mpPaused:   Caption := 'Paused';
        mpPlaying:  Caption := 'Playing';
      end;
    end
  { If these buttons are displayed the CD Player must be stopped... }
  else if mpCDPlayer.EnabledButtons = StopButtons then
    LblStatus.Caption := 'Stopped';
end;

procedure TMainForm.ShowCurrentTime;
{ This method displays the current time of the current track }
begin
  { Minutes for this track }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Seconds for this track }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
  { update track time label }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { update track gauge }
  ggTrackDone.Progress := (60 * m) + s;
end;

procedure TMainForm.ShowTrackTime;
{ This method changes the track time to display the total length of the
  currently selected track. }
var
  Min, Sec: Byte;
  Len: Longint;
```

```
begin
  { Don't update the information if player is still on the same track }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;

procedure TMainForm.HighlightTrackButton;
{ This procedure changes the color of the speedbutton font for the current
  track to red, while changing other speedbuttons to navy blue. }
var
  i: longint;
begin
  { iterate through form's components }
  for i := 0 to ComponentCount - 1 do
    { is it a speedbutton? }
    if Components[i] is TSpeedButton then
      if TSpeedButton(Components[i]).Tag = CurrentTrack then
        { turn red if current track }
        TSpeedButton(Components[i]).Font.Color := clRed
      else
        { turn blue if not current track }
        TSpeedButton(Components[i]).Font.Color := clNavy;
end;

procedure TMainForm.ShowTrackNumber;
{ This method displays the currently playing track number. }
var
  t: byte;
begin
  t := mci_Tmsf_Track(mpCDPlayer.Position);     // get current track
  CurrentTrack := t;                           // set instance variable
  LblCurTrk.Caption := TrackNumToString(t); // set Curr Track label caption
  HighlightTrackButton;                        // Highlight current speedbutton
end;

procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ This method is the heart of the CD Player.  It updates all information at
  every timer interval. }
begin
```

continues

LISTING 18.5 Continued

```
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
                            mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Show track number the CD player is currently on
    ShowTrackTime;   // Show total time for the current track
    ShowCurrentTime; // Show elapsed time for the current track
  end;
end;

procedure TMainForm.mpCDPlayerPostClick(Sender: TObject;
  Button: TMPBtnType);
{ This method displays the correct CD Player buttons when one of the buttons
  are clicked. }
begin
  Case Button of
    btPlay:
      begin
        mpCDPlayer.EnabledButtons := PlayButtons;
        LblStatus.Caption := 'Playing';
      end;
    btPause:
      begin
        mpCDPlayer.EnabledButtons := StopButtons;
        LblStatus.Caption := 'Paused';
      end;
    btStop:
      begin
        mpCDPlayer.Rewind;
        mpCDPlayer.EnabledButtons := StopButtons;
        LblCurTrk.Caption := 'One';
        LblTrackTime.Caption := '0m 0s';
        ggTrackDone.Progress := 0;
        ggDiskDone.Progress := 0;
        LblStatus.Caption := 'Stopped';
      end;
    btPrev, btNext:
      begin
        mpCDPlayer.Play;
        mpCDPlayer.EnabledButtons := PlayButtons;
        LblStatus.Caption := 'Playing';
      end;
  end;
```

```
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ This method is called when the form is created. It opens and initializes the
  player }
begin
  try
    mpCDPlayer.Open;    // Open the CD Player device.
    { If a CD is already playing at startup, show playing status. }
    if mpCDPlayer.Mode = mpPlaying then
      LblStatus.Caption := 'Playing';
    GetCDTotals;      // Show total time and tracks on current CD
    ShowTrackNumber;  // Show current track
    ShowTrackTime;    // Show the minutes and seconds for the current track
    ShowCurrentTime;  // Show the current position of the CD
    ShowPlayerStatus; // Update the CD Player's status
  except
    { If an error occurred, the system may be incapable of playing CDs. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player.  Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
  { Check the current mode of the CD-ROM and enable the appropriate buttons. }
  case mpCDPlayer.Mode of
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release;  // Close and free the splash screen
end;

procedure TMainForm.sbTrack1Click(Sender: TObject);
{ This method sets the current track when the user presses one of the track
  speed buttons.  This method works with all 20 speed buttons, so by looking at
  the 'Sender' it can tell which button was pressed by the button's tag. }
begin
  mpCDPlayer.Stop;
  { Set the start position on the CD to the start of the newly selected track }
  Track := (Sender as TSpeedButton).Tag;
  mpCDPlayer.StartPos := mpCDPlayer.TrackPosition[Track];
  { Start playing CD at new position }
  mpCDPlayer.Play;
  mpCDPlayer.EnabledButtons := PlayButtons;
  LblStatus.Caption := 'Playing';
```

continues

LISTING 18.5 Continued

```
end;

procedure TMainForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  mpCDPlayer.Close;
end;

end.
```

Summary

That about wraps up the basic concepts of Delphi's `TMediaPlayer` component. This chapter demonstrates the power and simplicity of this component through several examples. In particular, you learned about the common multimedia formats of WAV audio, AVI audio/video, and CD audio.

Testing and Debugging

CHAPTER

19

IN THIS CHAPTER

Some programmers in the industry believe that the knowledge and application of good programming practice make the need for debugging expertise unnecessary. In reality, however, the two complement each other, and whoever masters both will reap the greatest benefits. This is especially true when multiple programmers are working on different parts of the same program. It's simply impossible to completely remove the possibility of human error.

A surprising number of people say, "My code compiles all right, so I don't have any bugs, right?" Wrong. There's no correlation between whether a program compiles and whether it has bugs; there's a big difference between code that's syntactically correct and code that's logically correct and bug-free. Also, don't assume that because a particular piece of code worked yesterday or on another system that it's bug-free. When it comes to hunting software bugs, everything should be presumed guilty until proven innocent.

During the development of any application, you should allow the compiler to help you as much as possible. You can do this in Delphi by enabling all the runtime error-checking options in Project, Options, Compiler, as shown in Figure 19.1, or by enabling the necessary directives in your code. Additionally, you should have the Show Hints and Show Warnings options enabled in that same dialog box in order to receive more information on your code. It's common for a developer to spend needless hours trying to track down "that impossible bug," when he or she could have found the error immediately by simply employing these effective compiler-aided tools. (Of course, the authors would never be guilty of failing to remember to use these aids. You believe us, right?)

Table 19.1 describes the different runtime error options available through Delphi.

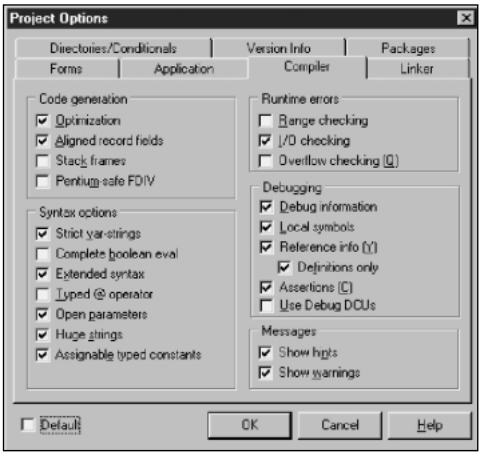

FIGURE 19.1
The Compiler page of the Project Options dialog box.

TABLE 19.1 Delphi Runtime Errors

Runtime Error	*Directive*	*Function*
Range Checking	`{$R+}`	Checks to ensure that you don't index an array or string beyond its bounds and that assignments don't assign a value to a scalar variable that's outside its range.
I/O Checking	`{$I+}`	Checks for an input/output error after every I/O call (`ReadLn()` and `WriteLn()`, for example). This almost always should be enabled.
Overflow Checking	`{$Q+}`	Checks to ensure that calculation results are not larger than the register size.

TIP

Keep in mind that each of these runtime errors exacts a performance penalty on your application. Therefore, once you're out of the debugging phase of development and are ready to ship a final product, you can improve performance by disabling some of the runtime error checks. It's common practice for developers to disable all of them except I/O Checking for the final product.

Common Program Bugs

This section shows some commonly made mistakes that cause programs to fail or crash. If you know what to look for when you're debugging code, you can lessen the time needed to find errors.

Using a Class Variable Before It's Created

One of the most common bugs that creeps up when you develop in Delphi occurs because you've used a class variable before it has been created. For example, take a look at the following code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList.Assign(ListBox1.Items);
end;
```

The `TStringList` class `MyStringList` has been declared; however, it's used before it's instantiated. This is a sure way to cause an access violation. You must be sure to instantiate any class

variables before you try to use them. The following code shows the correct way to instantiate and use a class variable. However, it also introduces another bug. Can you see it?

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
  MyStringList.Assign(ListBox1.Items);
end;
```

If your answer was, "You didn't free your `TStringList` class," you're correct. This won't cause your program to fail or crash, but it will eat up memory because, every time you call this method, another `TStringList` is created and thrown away, thereby leaking memory. Although the Win32 API will free all memory allocated by your process at the time it terminates, leaking memory while running an application can cause serious problems. For example, a leaky application will continue to eat more and more of the system's memory resources as it runs, causing the OS to have to perform more disk swapping, which ultimately slows down the entire system.

The corrected version of the preceding code listing is shown in the following code (minus a necessary enhancement discussed in the next topic):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;               // Create it!
  MyStringList.Assign(ListBox1.Items);              // Use it!
  { Do your stuff with your TStringList instance }
  MyStringList.Free;                                // Free it!
end;
```

Ensuring That Class Instances Are Freed

Suppose that in the previous code example, an exception occurs just after `TStringList` is created. The exception would cause the flow of execution to immediately exit the procedure, and none of the procedure's remaining code would be executed, which would cause a memory loss. Make sure your class instances are freed, even if an exception occurs, by using a `try..finally` construct, as shown here:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;                 // Create it!
  try
    MyStringList.Assign(ListBox1.Items);              // Use it!
```

```
    { Do your stuff with your TStringList instance }
  finally
    MyStringList.Free;                                  // Free it!
  end;
end;
```

After you read the section "Breakpoints" later in the chapter, try an experiment and place the following line right after the line where you assign the `ListBox1` items to the `TStringList`:

```
raise Exception.Create('Test Exception');
```

Then place a breakpoint at the beginning of the method's code and step through the code. You'll see that `TStringList` still gets freed, even after the exception is raised.

Taming the Wild Pointer

The *wild pointer bug* is a common error that clobbers some part of memory when you use the pointer to write to memory. The wild pointer genus has two common species: the uninitialized pointer and the stale pointer.

An *uninitialized pointer* is a pointer variable that's used before memory has been allocated for it. When such a pointer is used, you end up writing to whatever address happens to live at the location of the pointer variable. The following code example illustrates an uninitialized pointer:

```
var
  P: ^Integer;
begin
  P^ := 1971;  // Eeek! P is uninitialized!
```

A *stale pointer* is a pointer that references an area of memory that was once properly allocated but has been freed. The following code shows a stale pointer:

```
var
  P: ^Integer;
begin
  New(P);
  P^ := 1971;
  Dispose(P);
  P^ := 4;  // Eeek! P is stale!
```

If you're lucky, you'll receive an access violation when you attempt to write to a wild pointer. If you're not so lucky, you'll end up writing over data used by some other part of your application. This type of error is absolutely no fun to debug. On one machine, the pointer may appear to run just fine until you transfer it to another machine (and maybe make a few code changes in the process), where it begins to malfunction. This may lead you to believe that the recent changes you made are faulty or that the second machine has a hardware problem. Once you've fallen into this trap, all the good programming practice in the world won't save you. You may start adding instances of `ShowMessage()` to portions of your code in an attempt to find the

problem, but this serves only to modify the code's location in memory and might cause the bug to move around—or worse, disappear! Your best defense against wild pointer bugs is to avoid them in the first place. Whenever you need to work with pointers and manual memory allocation, make sure you check and double-check your algorithms to avoid the silly mistake that may introduce a bug.

Using Uninitialized PChar-Type Variables

You'll often see wild pointer errors when you use `PChar`-type variables. Because a `PChar` is just a pointer to a string, you have to remember to allocate memory for the `PChar` by using the `StrAlloc()`, `GetMem()`, `StrNew()`, `GlobalAlloc()`, or `VirtualAlloc()` function, as well as using the `FreeMem()`, `StrDispose()`, `GlobalFree()`, or `VirtualFree()` function to free it.

TIP

You can avoid potential bugs in your program by using `string`-type variables where possible, instead of `PChars`. You can typecast a `string` to a `PChar`, so the code involved is simple, and because `strings` are automatically allocated and freed, you don't have to concern yourself with memory allocation.

This holds true especially for Delphi 1.0 applications that you're porting to 32-bit Delphi. In Delphi 1.0, `PChars` are a necessary evil. In 32-bit Delphi, they're necessary only on rare occasions. Take the time to move to `strings` as you port your applications to 32-bit Delphi.

Dereferencing a nil Pointer

In addition to the wild pointer, another common mistake is dereferencing a `nil` (zero-value) pointer. Dereferencing a `nil` pointer always causes the operating system to issue an access violation error. Although this isn't an error that you want to have in your application, it's generally not fatal. Because it doesn't actually corrupt memory, it's safe to use exception handling to take care of the exception and move along. The sample procedure in the following code listing illustrates this point:

```
procedure I_AV;
var
  P: PByte;
begin
  P := Nil;
  try
    P^ := 1;
  except
```

```
    on EAccessViolation do
      MessageDlg('You can''t do that!!', mtError, [mbOk], 0);
  end;
end;
```

If you put this procedure in a program, you'll see that the message dialog box appears to inform you of the problem, but your program continues to run.

Using the Integrated Debugger

Delphi provides a feature-rich debugger built right into the IDE. Most of the facilities of the integrated debugger can be found on the Run menu. These facilities include all the features you would expect of a professional debugger, including the ability to specify command-line parameters for your application, set breakpoints, perform trace and step, add and view watches, evaluate and modify data, and view call stack information.

Using Command-Line Parameters

If your program is designed to use command-line parameters, you can specify them in the Run Parameters dialog box. In this dialog box, simply type the parameters as you would on the command line or in the Windows Start menu's Run dialog box.

Breakpoints

Breakpoints enable you to suspend the execution of your program whenever a certain condition is met. The most common type of breakpoint is a *source breakpoint*, which occurs when a particular line of code is about to be executed. You can set a source breakpoint by clicking to the far left of a line of code in the Code Editor, by using the local menu, or by selecting Run, Add Breakpoint. Whenever you want to see how your program is behaving inside a particular procedure or function, just set a breakpoint on the first line of code in that routine. Figure 19.2 shows a source breakpoint set on a line of program code.

Conditional Breakpoints

You can add additional information to a source breakpoint to suspend the execution of your program when some condition occurs in addition to when a line of code is reached. A typical example is when you want to examine the code inside a loop construct. You probably don't want to suspend and resume execution every time your code passes through the loop, especially if the loop occurs hundreds, or perhaps thousands, of times. Instead of continually pressing the F9 key to run, just set a breakpoint to occur whenever a variable reaches a certain value. For example, in a new project, place a `TButton` on the main form and add the following code to the button's event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  I: Integer;
begin
  for I := 1 to 100 do
  begin
    Caption := IntToStr(I);         // update form
    Button1.Caption := IntToStr(I); // update button
    Application.ProcessMessages;    // let updates happen
  end;
end;
```

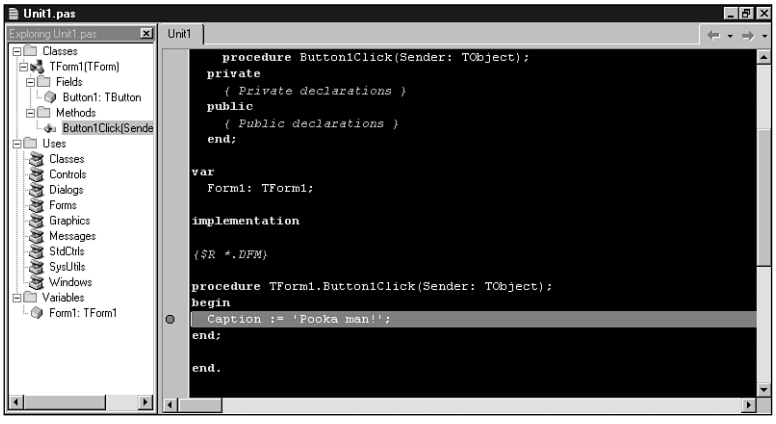

FIGURE 19.2
A source breakpoint set in the Code Editor.

Now set a breakpoint on the following line:

```
Caption := IntToStr(I);         // update form
```

After you've set a breakpoint, select View, Debug Windows, Breakpoints, which will bring up a Breakpoint List dialog box. Your breakpoint should show up in this list. Right-click your breakpoint and select Properties from the local menu. This will invoke the Edit Breakpoint dialog box, as shown in Figure 19.3. In the Condition input line, enter `I = 50` and select OK. This will cause the breakpoint that you previously set to suspend program execution only when the variable `I` contains the value `50`.

TIP

Figure 19.3 provides a glimpse into the breakpoint actions feature, which is new to Delphi 5. Breakpoint actions enable you to specify the exact behavior of the debugger when a breakpoint is encountered. These actions are controlled using the three

checkboxes shown in the figure. Break, as you might imagine, instructs the debugger to break when the breakpoint is encountered. Ignore Subsequent Exceptions causes the debugger to refrain from breaking when exceptions are encountered from the breakpoint forward. Handle Subsequent Exceptions causes the debugger to resume the default behavior of breaking when exceptions are encountered from the breakpoint forward.

The latter two options are designed to be used in tandem. If you have a particular bit of code that is causing you problems by raising exceptions in the debugger and you don't want to be notified about it, you can use these breakpoint options to instruct the debugger to ignore exceptions before entering the code block and begin handling exceptions once again after leaving the block.

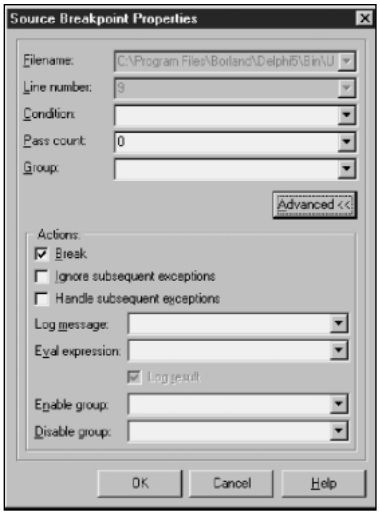

FIGURE 19.3
The Edit Breakpoint dialog box.

Data Breakpoints

Data breakpoints are breakpoints you can set to occur when memory at a particular address is modified. This is useful for low-level debugging, when you need to track down bugs that perhaps occur when a variable gets assigned. You can set data breakpoints by selecting Run, Add Breakpoint, Data Breakpoint from the main menu or by using the local menu on the Breakpoint List dialog box. This invokes the Add Data Breakpoint dialog box, as shown in Figure 19.4. In this dialog box, you can enter the start address of the area of memory you want to monitor and the length (number of bytes) to monitor after that address. By specifying the

number of bytes, you can watch anything from a `Char` (one byte) to an `Integer` (four bytes) to an array or record (any number of bytes). In a manner similar to source breakpoints, the Add Data Breakpoint dialog box also allows you to enter an expression that will be evaluated when the memory region is written to so that you can find those bugs that occur on the *n*th time a memory region is set. If you want the debugger to break when a specific variable is modified, just enter the name of the variable in the address field.

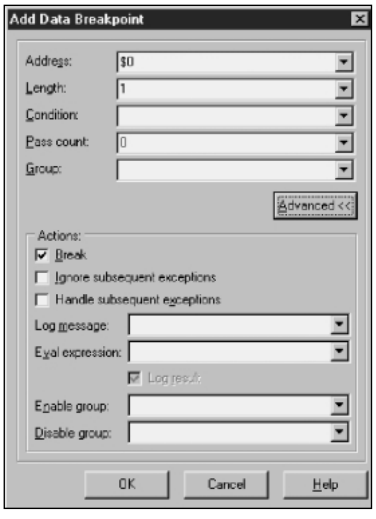

FIGURE 19.4
The Add Data Breakpoint dialog box.

Address Breakpoints

An *address breakpoint* is a breakpoint you can set to occur when code residing at a particular address is executed. These types of breakpoints are normally set from the local menu in the CPU view when you can't set a source breakpoint because you don't have the source code for a particular module. As with other types of breakpoints, you can also specify a condition for address breakpoints in order to fine-tune your breakpoints.

Module Load Breakpoints

As you can probably surmise from the name, *module load breakpoints* enable you to set breakpoints that occur when a specified module is loaded in the debugged application's process. This allows you to be notified immediately when a DLL or package is loaded by an application. The most common place to set module load breakpoints is the local menu in the Modules window, but they can also be set by using the Run, Add Breakpoint item on the main menu.

Breakpoint Groups

Breakpoint groups are one of the most powerful and time-saving features the integrated debugger offers. Using groups, any breakpoint can be set up to enable or disable any other breakpoint so that a very complex algorithm of breakpoints can be created to find very specific bugs. Suppose you suspect that a bug shows up in your `Paint()` method only after you choose a particular menu option. You could add a breakpoint to the `Paint()` method, run the program, and constantly tell the debugger to continue when you get barraged with hundreds of calls to your `Paint()` method. Alternatively, you could keep that breakpoint on your `Paint()` method, disable it so that it doesn't fire, and then add another breakpoint to your menu-select event handler to enable the `Paint()` method breakpoint. Now you can run full speed in the debugger and not break in your `Paint()` handler until after you select the menu choice.

Executing Code Line by Line

You can execute code line by line by using either the Step Over or Trace Into option (F8 and F7 keys, respectively, in the Default and IDE classic keymapping). Trace Into steps into your procedures and functions as they're called; Step Over executes the procedure or function immediately without stepping into it. Typically, you use these options after stopping somewhere in your code with a breakpoint. Get to know the F7 and F8 keys; they are your friends.

You can also tell Delphi to run your program up to the line that the cursor currently inhabits by using the Run To Cursor (F4) option. This is particularly useful when you want to bypass a loop that's iterated many times, in which case using F7 or F8 becomes tedious. Keep in mind that you can set breakpoints at any time in the Code Editor—even as your program executes; you don't have to set all the breakpoints up front.

TIP

If you accidentally step into a function that will be very difficult or time-consuming to step out of, choose Run, Run Until Return from the main menu to cause the debugger to break after the current procedure or function returns.

You can breakpoint your code dynamically by using the Program Pause option. This option often helps you determine whether your program is in an infinite loop. Keep in mind that VCL code is being run most of your program's life, so you often won't stop on a line of your program's code with this option.

TIP

When you debug your application, you've probably noticed the blue dots shown in the "gutter" on the left side of the Code Editor window. One of these blue dots is shown next to each line of code for which machine code is generated. You can't set a breakpoint on or step to a particular line of code if it doesn't have a blue dot next to it because no machine code is associated with the line.

Using the Watch Window

You can use the Watch window to track the values of your program's variables as your code executes. Keep in mind that you must be in a code view of your program (a breakpoint should be executed) for the contents of the Watch window to be accurate. You can enter an Object Pascal expression or register name into the Watch window. This is shown in Figure 19.5.

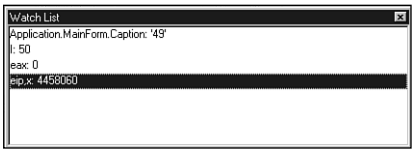

FIGURE 19.5
Using the Watch List window.

Debug Inspectors

A debug inspector is a kind of data inspector that's perhaps easier to use and more powerful in some ways than the Watch window. To use this feature, select Run, Inspect while debugging an application. This will invoke a simple dialog box into which you can enter an expression. Click OK, and you'll be presented with a Debug Inspector window for the expression you entered. For example, Figure 19.6 shows a Debug Inspector for the main form of a do-nothing Delphi application.

The Debug Inspector window provides a means for conveniently viewing data that consists of many individual elements, such as classes and records. Click the ellipses on the right of the value column in the Inspector to modify the value of a field. You can even drill down into record or class data members by double-clicking a field of this type in the list.

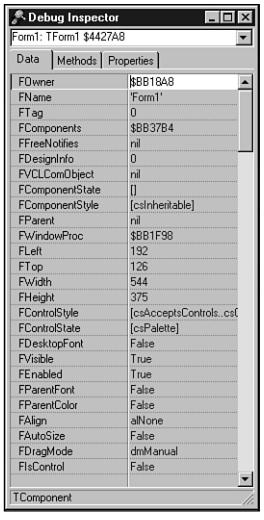

FIGURE 19.6
Inspecting a form using a Debug Inspector.

Using the Evaluate and Modify Options

The Evaluate and Modify options enable you to inspect and change the contents of variables, including arrays and records, on the fly as your application executes in the integrated debugger. Keep in mind that this feature doesn't enable you to access functions or variables that are out of scope.

CAUTION

Evaluating and modifying variables is perhaps one of the more powerful features of the integrated debugger, but with that power comes the responsibility of having direct access to memory. You must be careful when changing the values of variables, because changes can affect the behavior of your program later.

Accessing the Call Stack

You can access the call stack by choosing View, Debug Windows, Call Stack. This enables you to view function and procedure calls along with the parameters passed to them. The call stack is useful for seeing a road map of functions that were called up to the current point in your source code. Figure 19.7 shows a typical view of the Call Stack window.

FIGURE 19.7
The Call Stack window.

TIP

To view any procedure or function listed in the Call Stack window, simply right-click inside the window. This is a good trick for getting back to a function when you accidentally trace in too far.

Viewing Threads

If your application makes use of multiple threads, the integrated debugger allows you to obtain information on the various threads in your application through the Thread Status window. Select View, Debug Windows, Threads from the main menu to invoke this window. When your application is paused (has hit a breakpoint), you can use the local menu provided by this window to make another thread current or to view the source associated with a particular thread. Remember that whenever you modify the current thread, the next run or step command you issue is relative to that thread. Figure 19.8 shows the Thread Status window.

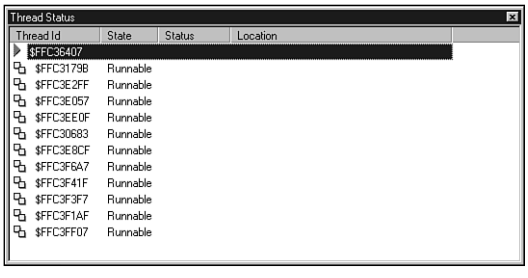

FIGURE 19.8
The Thread Status window.

Event Log

The Event Log provides a place into which the debugger will log a record for the occurrence of various events. The Event Log, shown in Figure 19.9, is accessible from the View, Debug menu. You can configure the Event Log by using its local menu or the Debugger page of the Tools, Environment Options dialog box.

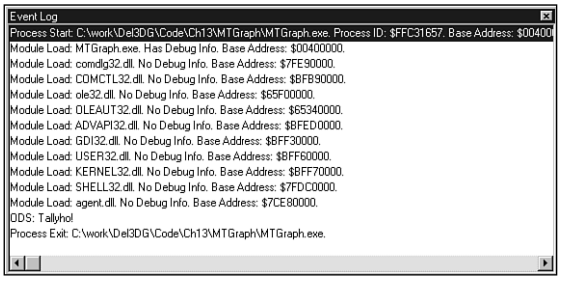

FIGURE 19.9
The Event Log.

The types of events you can log include process information such as process start, process stop, and module load debugger breakpoints, as well as Windows messages sent to the application and application output using `OutputDebugString()`.

TIP

The `OutputDebugString()` API function provides a handy means to help you debug applications. The single parameter to `OutputDebugString()` is a `PChar`. The string passed in this parameter will be passed on to the debugger, and in the case of Delphi, the string will be added to the Event Log. This allows you to keep track of variable values or similar debug information without having to use watches or displaying intrusive debug dialog boxes.

Modules View

The Modules view enables you to obtain information on all the modules (`EXE`, `DLL`, `BPL`, and so on) loaded into the debugged application's process. Shown in Figure 19.10, this window provides you with a list of who's who in your application's process, permits you to set module load breakpoints, and provides you with various types of information on each module.

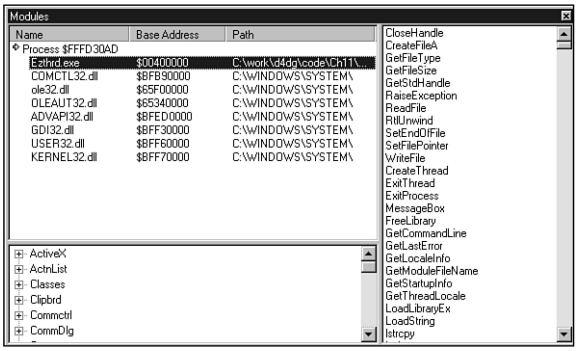

FIGURE 19.10
The Modules view.

DLL Debugging

The Delphi integrated debugger provides you with the ability to debug your DLL projects using any arbitrary application as the host. If fact, it's quite easy. Open your DLL project and select Run, Parameters from the main menu. Then specify a host application in the Run Parameters dialog box, as shown in Figure 19.11.

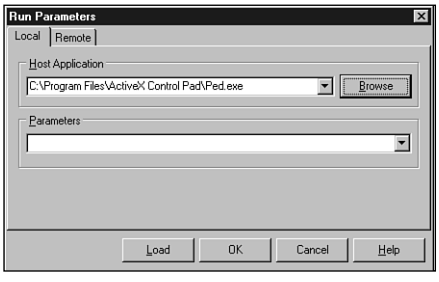

FIGURE 19.11
Specifying a host application.

The host application is an executable file that loads and uses the DLL you're currently debugging. After specifying a proper host application, you can use the integrated debugger much as you would for debugging a normal executable; you can set breakpoints, step, trace, and so on.

This feature is most useful for debugging ActiveX controls and in-process COM servers that are executed from within the context of another process. For example, you can use this feature to debug your ActiveX control from within Visual Basic.

The CPU View

The CPU view, found by selecting View, Debug Windows, CPU from the main menu, provides a developer's-eye view of what's going on inside the machine's CPU. The CPU view consists of five panes of information: the CPU pane, the Memory Dump pane, the Register pane, the Flags pane, and the Stack pane (see Figure 19.12). Each of these panes enables the user to view important aspects of the processor as an aid to debugging.

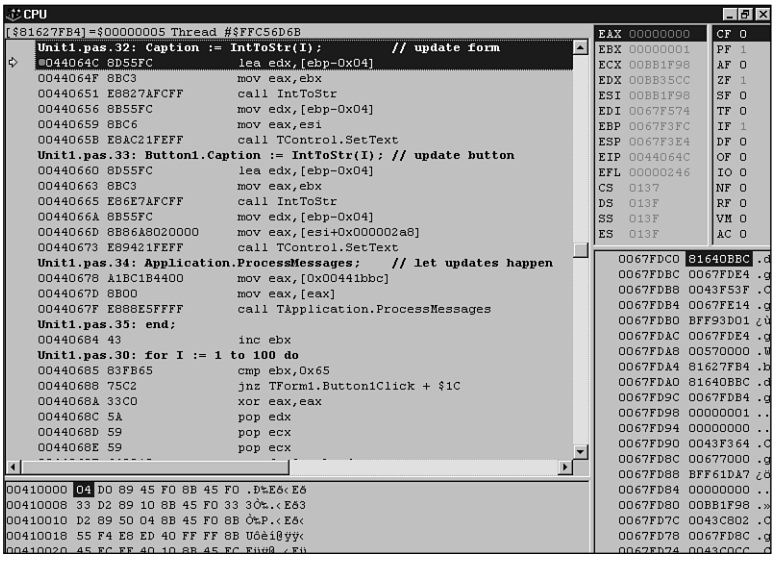

FIGURE 19.12
The CPU view.

The CPU pane shows the opcodes and mnemonics of the disassembled assembly code that's being executed. You can position the CPU pane at any address in your process to view instructions, or you can set the current instruction pointer to any new location, from which execution then continues. It's helpful to be able to understand the assembly code that the CPU pane displays, and experienced developers will attest to the fact that many bugs have been found and exterminated by examining the assembly code generated for a routine and realizing that it wasn't performing the desired operation. Someone who doesn't understand assembly language obviously wouldn't be able to find such a bug as quickly.

The local menu of the CPU view allows you to change the way items are displayed, look at a different address, go back to the current instruction pointer (EIP), search, go back to the source code, and so on. You can also pick the thread context in which to view the CPU information.

The Memory Dump pane enables you to view the contents of any range of memory. There are many ways in which it can be viewed—as `Byte`, `Word`, `DWORD`, `QWORD`, `Single`, `Double`, or `Extended`. You can search memory for a sequence of bytes as well as modify the data or follow it as code or data pointers.

The Register and Flags panes are pretty straightforward. All the CPU registers and flags are displayed here and can be modified.

The Stack pane gives you a stack-based view of the memory that's used for your program stack. In this pane, you can change values of data on the stack and follow addresses.

Summary

This chapter gives you some insight into the debugging process. It shows you the common problems you might run into while developing applications, and discusses the useful features of both the integrated and standalone debuggers. It's important to remember that debugging is as much a part of programming as is writing code. Your debugger can be one of your most powerful allies in writing clean code, so take the time to know it well.

In the next part of the book, you'll move into the realm of component-based development with COM and VCL components.

Extending Database VCL

CHAPTER

30

IN THIS CHAPTER

Out of the box, Visual Component Library's (VCL's) database architecture is equipped to communicate primarily by means of the Borland Database Engine (BDE)—feature-rich and reliable database middleware. What's more, VCL serves as a kind of insulator between you and your databases, allowing you to access different types of databases in much the same manner. Although all this adds up to reliability, scalability, and ease of use, there is a downside: database-specific features provided both within and outside the BDE are generally not provided for in the VCL database framework. This chapter provides you with the insight you'll need to extend VCL by communicating directly with the BDE and other data sources to obtain database functionality not otherwise available in Delphi.

Using the BDE

When you're writing applications that make direct calls to the BDE, there are a few rules of thumb to keep in mind. This section presents the general information you need to get into the BDE API from your Delphi applications.

The BDE Unit

All BDE functions, types, and constants are defined in the `BDE` unit. This unit will need to be in the `uses` clause of any unit from which you want to make BDE calls. Additionally, the interface portion of the `BDE` unit is available in the `BDE.INT` file, which you'll find in the `..\Delphi5\Doc` directory. You can use this file as a reference to the functions and records available to you.

TIP

For additional assistance on programming using the BDE API, take a look at the `BDE32.hlp` help file provided in your `BDE` directory (the default path for this directory is `\Program Files\Borland\Common Files\BDE`). This file contains detailed information on all BDE API functions and very good examples in both Object Pascal and C.

Check()

All BDE functions return a value of type `DBIRESULT`, which indicates the success or failure of the function call. Rather than going through the cumbersome process of checking the result of every BDE function call, Delphi defines a procedure called `Check()`, which accepts a `DBIRESULT` as a parameter. This procedure will raise an exception when the `DBIRESULT` indicates any value except success. The following code shows how to, and how not to, make a BDE function call:

```
// !!Don't do this:
var
```

```
  Rez: DBIRESULT;
  A: array[0..dbiMaxUserNameLen] of char;
begin
  Rez := dbiGetNetUserName(A);  // make BDE call
  if Rez <> DBIERR_NONE then    // handle error
    // handle error here
  else begin
    // continue with function
  end;
end;

// !!Do do this:
var
  A: array[0..dbiMaxUserNameLen] of char;
begin
  { Handle error and make BDE call at one time. }
  { Exception will be raised in case of error.  }
  Check(dbiGetNetUserName(A));
  // continue with function
end;
```

Cursors and Handles

Many BDE functions accept as parameters handles to cursors or databases. Roughly speaking, a *cursor handle* is a BDE object that represents a particular set of data positioned at some particular row in that data. The data type of a cursor handle is `hDBICur`. Delphi surfaces this concept as the current record in a particular table, query, or stored procedure. The `Handle` properties of `TTable`, `TQuery`, and `TStoredProc` hold this cursor handle. Remember to pass the `Handle` of one of these objects to any BDE function that requires an `hDBICur`.

Some BDE functions also require a handle to a database. A BDE database handle is of type `hDBIDb`, and it represents some particular open database—either a local or networked directory in the case of dBASE or Paradox, or a server database file in the case of a SQL server database. You can obtain this handle from a `TDatabase` through its `Handle` property. If you're not connecting to a database using a `TDatabase` object, the `DBHandle` properties of `TTable`, `TQuery`, and `TStoredProc` also contain this handle.

Synching Cursors

It's been established that an open Delphi dataset has the concept of a current record, whereas the underlying BDE maintains the concept of a cursor that points to some particular record in a dataset. Because of the way Delphi performs record caching to optimize performance, sometimes the Delphi current record is not in sync with the underlying BDE cursor. Normally, this is not a problem because this behavior is business as usual for VCL's database framework.

However, if you want to make a direct call to a BDE function that expects a cursor as a parameter, you need to ensure that Delphi's current cursor position is synchronized with the underlying BDE cursor. It might sound like a daunting task, but it's actually quite easy to do. Simply call the `UpdateCursorPos()` method of a `TDataSet` descendant to perform this synchronization.

In a similar vein, after making a BDE call that modifies the position of the underlying cursor, you need to inform VCL that it needs to resynchronize its own current record position with that of the BDE. To do this, you must call the `CursorPosChanged()` method of `TDataSet` descendants immediately after calling into the BDE. The following code demonstrates how to use these cursor-synchronization functions:

```
procedure DoSomethingWithTable(T: TTable);
begin
  T.UpdateCursorPos;
  // call BDE function(s) which modifies cursor position
  T.CursorPosChanged;
end;
```

dBASE Tables

dBASE tables have a number of useful capabilities that are not directly supported by Delphi. These features include, among other things, the maintenance of a unique physical record number for each record, the capability to "soft-delete" records (delete records without removing them from the table), the capability to undelete soft-deleted records, and the capability to pack a table to remove soft-deleted records. In this section, you'll learn about the BDE functions involved in performing these actions, and you'll create a `TTable` descendant called `TdBaseTable` that incorporates these features.

Physical Record Number

dBASE tables maintain a unique physical record number for each record in a table. This number represents a record's physical position relative to the beginning of the table (regardless of any index currently applied to the table). To obtain a physical record number, you must call the BDE's `DbiGetRecord()` function, which is defined as follows:

```
function DbiGetRecord(hCursor: hDBICur; eLock: DBILockType;
    pRecBuff: Pointer; precProps: pRECProps): DBIResult stdcall;
```

`hCursor` is the cursor handle. Usually, this is the `Handle` property of the `TDataSet` descendant.

`eLock` is an optional request for the type of lock to place on the record. This parameter is of type `DBILockType`, which is an enumerated type defined as follows:

```
type
  DBILockType = (
```

```
    dbiNOLOCK,             // No lock (Default)
    dbiWRITELOCK,          // Write lock
    dbiREADLOCK);          // Read lock
```

In this case you don't want to place a lock on the record because you're not intending to modify the record content; therefore, `dbiNOLOCK` is the appropriate choice.

`pRecBuff` is a pointer to a record buffer. Because you want to obtain only the record properties and not the data, you should pass `Nil` for this parameter.

`pRecProps` is a pointer to a `RECProps` record. This record is defined as follows:

```
type
  pRECProps = ^RECProps;
  RECProps = packed record       // Record properties
    iSeqNum         : Longint;   // When Seq# supported only
    iPhyRecNum      : Longint;   // When Phy Rec#s supported only
    iRecStatus      : Word;      // Delayed Updates Record Status
    bSeqNumChanged  : WordBool;  // Not used
    bDeleteFlag     : WordBool;  // When soft delete supported only
  end;
```

As you can see, you can obtain a variety of information from this record. In this case, you're concerned only with the `iPhyRecNum` field, which is valid only in the case of dBASE and FoxPro tables.

Putting this all together, the following code shows a method of `TdBaseTable` that returns the physical record number of the current record:

```
function TdBaseTable.GetRecNum: Longint;
{ Returns the physical record number of the current record. }
var
  RP: RECProps;
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get current record properties }
  Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // return value from properties
end;
```

Viewing Deleted Records

Viewing records that have been soft-deleted in a dBASE table is as easy as making one BDE API call. The function to call is `DbiSetProp()`, which is a very powerful function that enables you to modify the different properties of multiple types of BDE objects. For a complete description of this function and how it works, your best bet is to check out the "Properties—Getting and Setting" topic in the BDE help. This function is defined as follows:

```
function DbiSetProp(hObj: hDBIObj; iProp: Longint;
  iPropValue: Longint): DBIResult stdcall;
```

The `hObj` parameter holds a handle to some type of BDE object. In this case, it will be a cursor handle.

The `iProp` parameter will contain the identifier of the property to be set. You'll find a complete list of these in the aforementioned topic in the BDE help. For the purposes of enabling or disabling the view of deleted records, use the `curSOFTDELETEON` identifier.

`iPropValue` is the new value for the given property. In this case, it's a Boolean value (`0` meaning *off*; `1` meaning *on*).

The following code shows the `SetViewDeleted()` method of `TdBaseTable`:

```
procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Allows the user to toggle between viewing and not viewing }
{ deleted records. }
begin
  { Table must be active }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // avoid flicker
    try
      { Magic BDE call to toggle view of soft deleted records }
      Check(DbiSetProp(hDBIObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // update Delphi
      EnableControls;    // flicker avoidance complete
    end;
    FViewDeleted := Value
  end;
end;
```

This method first performs a test to ensure that the table is open and that the value to be set is different than the value the `FViewDeleted` field in the object already contains. It then calls `DisableControls()` to avoid flicker of any data-aware controls attached to the table. The `DbiSetProp()` function is called next (notice the necessary typecast of `hDBICur`'s `Handle` parameter to an `hDBIObj`). Think of `hDBIObj` as an untyped handle to some type of BDE object. After that, the dataset is refreshed and any attached controls are reenabled.

TIP

Whenever you use `DisableControls()` to suspend a dataset's connection to data-aware controls, you should always use a `try..finally` block to ensure that the subsequent call to `EnableControls()` takes place whether or not an error occurs.

Testing for a Deleted Record

When viewing a dataset that includes deleted records, you'll probably need to determine as you navigate through the dataset which records are deleted and which aren't. Actually, you've already learned how to perform this check. You can obtain this information using the `DbiGetRecord()` function that you used to obtain the physical record number. The following code shows this procedure. The only material difference between this procedure and `GetRecNum()` is the checking of the `bDeletedFlag` field rather than the `iPhyRecNo` field of the `RECProps` record. Here's the code:

```
function TdBaseTable.GetIsDeleted: Boolean;
{ Returns a boolean indicating whether or not the current record }
{ has been soft deleted. }
var
  RP: RECProps;
begin
  if not FViewDeleted then     // don't bother if they aren't viewing
    Result := False            // deleted records
  else begin
    UpdateCursorPos;           // update BDE from Delphi
    { Get current record properties }
    Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // return flag from properties
  end;
end;
```

Undeleting a Record

So far, you've learned how to view deleted records as well as determine whether a record has been deleted, and, of course, you already know how to delete a record. The only other thing you need to learn regarding record deletion is how to undelete a record. Fortunately, the BDE makes this an easy task thanks to the `DbiUndeleteRecord()` function, which is defined as follows:

```
function DbiUndeleteRecord(hCursor: hDBICur): DBIResult stdcall;
```

The lone parameter is a cursor handle for the current dataset. Using this function, you can create an `UndeleteRecord()` method for `TdBaseTable` as shown here:

```
procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(DbiUndeleteRecord(Handle));
  Refresh;
end;
```

Packing a Table

To remove soft-deleted records from a dBASE table, that table must go through a process called *packing*. For this, the BDE provides a function called `DbiPackTable()`, which is defined as follows:

```
function DbiPackTable(hDb: hDBIDb; hCursor: hDBICur;
    pszTableName: PChar; pszDriverType: PChar;
    bRegenIdxs: Bool): DBIResult stdcall;
```

`hDb` is a handle to a database. You should pass the `DBHandle` property of a `TDataSet` descendant or the `Handle` property of a `TDatabase` component in this parameter.

`hCursor` is a cursor handle. You should pass the `Handle` property of a `TDataSet` descendant in this parameter. You may also pass `Nil` if you want to instead use the `pszTableName` and `pszDriverType` parameters to identify the table.

`pszTableName` is a pointer to a string containing the name of the table.

`pszDriverType` is a pointer to a string representing the driver type of the table. If `hCursor` is `Nil`, this parameter must be set to `szDBASE`. As a side note, it's unusual that this parameter is required because this function is supported only for dBASE tables—we don't make the rules, we just play by them.

`bRegenIdxs` indicates whether or not you want to rebuild all out-of-date indexes associated with the table.

Here's the `Pack()` method for the `TdBaseTable` class:

```
procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Packs the table in order to removed soft deleted records }
{ from the file. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { Table must be active and opened exclusively }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Pack the table }
    Check(DbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { update Delphi from BDE }
    CursorPosChanged;
    Refresh;
  end;
end;
```

The complete listing of the `TdBaseTable` object is provided in Listing 30.1, later in this chapter.

Paradox Tables

Paradox tables don't have as many nifty features, such as soft deletion, but they do carry the concept of a record number and table pack. In this section, you'll learn how to extend a `TTable` to perform these Paradox-specific tasks and to create a new `TParadoxTable` class.

Sequence Number

Paradox tables do not have the concept of a physical record number in the dBASE sense. They do, however, maintain the concept of a sequence number for each record in a table. The sequence number differs from the physical record number in that the sequence number is dependent on whatever index is currently applied to the table. The sequence number of a record is the order in which the record appears based on the current index.

The BDE makes it pretty easy to obtain a sequence number using the `DbiGetSeqNo()` function, which is defined as follows:

```
function DbiGetSeqNo(hCursor: hDBICur; var iSeqNo: Longint): DBIResult;
  stdcall;
```

`hCursor` is a cursor handle for a Paradox table, and the `iSeqNo` parameter will be filled in with the sequence number of the current record. The following code shows the `GetRecNum()` function for `TParadoxTable`:

```
cfunction TParadoxTable.GetRecNum: Longint;
{ Returns the sequence number of the current record. }
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get sequence number of current record into Result }
  Check(DbiGetSeqNo(Handle, Result));
end;
```

Table Packing

Table packing in Paradox has a different meaning than in dBASE because Paradox does not support soft deletion of records. When a record is deleted in Paradox, the record is removed from the table, but a "hole" is left in the database file where the record used to be. To compress these holes left by deleted records and make the table smaller and more efficient, you must pack the table.

Unlike with dBASE tables, there's no obvious BDE function you can use to pack a Paradox table. Instead, you must use `DbiDoRestructure()` to restructure the table and specify that the table should be packed as it's restructured. `DbiDoRestructure()` is defined as follows:

```
function DbiDoRestructure(hDb: hDBIDb; iTblDescCount: Word;
    pTblDesc: pCRTblDesc; pszSaveAs, pszKeyviolName,
    pszProblemsName: PChar; bAnalyzeOnly: Bool): DBIResult stdcall;
```

`hDb` is the handle to a database. However, because this function will not work when Delphi has the table open, you won't be able to use the `DBHandle` property of a `TDataSet`. To overcome this, the sample code (shown a bit later) that uses this function demonstrates how to create a temporary database.

`iTblDescCount` is the number of table descriptors. This parameter must be set to `1` because the current version of the BDE supports only one table descriptor per call.

`pTblDesc` is a pointer to a `CRTblDesc` record. This is the record that identifies the table and specifies how the table is to be restructured. This record is defined as follows:

```
type
  pCRTblDesc = ^CRTblDesc;
  CRTblDesc = packed record     // Create/Restruct Table descr
    szTblName    : DBITBLNAME;  // TableName incl. optional path & ext
    szTblType    : DBINAME;     // Driver type (optional)
    szErrTblName : DBIPATH;     // Error Table name (optional)
    szUserName   : DBINAME;     // User name (if applicable)
    szPassword   : DBINAME;     // Password (optional)
    bProtected   : WordBool;    // Master password supplied in szPassword
    bPack        : WordBool;    // Pack table (restructure only)
    iFldCount    : Word;        // Number of field defs supplied
    pecrFldOp    : pCROpType;   // Array of field ops
    pfldDesc     : pFLDDesc;    // Array of field descriptors
    iIdxCount    : Word;        // Number of index defs supplied
    pecrIdxOp    : pCROpType;   // Array of index ops
    pidxDesc     : PIDXDesc;    // Array of index descriptors
    iSecRecCount : Word;        // Number of security defs supplied
    pecrSecOp    : pCROpType;   // Array of security ops
    psecDesc     : pSECDesc;    // Array of security descriptors
    iValChkCount : Word;        // Number of val checks
    pecrValChkOp : pCROpType;   // Array of val check ops
    pvchkDesc    : pVCHKDesc;   // Array of val check descs
    iRintCount   : Word;        // Number of ref int specs
    pecrRintOp   : pCROpType;   // Array of ref int ops
    printDesc    : pRINTDesc;   // Array of ref int specs
    iOptParams   : Word;        // Number of optional parameters
    pfldOptParams: pFLDDesc;    // Array of field descriptors
    pOptData     : Pointer;     // Optional parameters
  end;
```

For Paradox table packing, it's only necessary to specify values for the `szTblName`, `szTblType`, and `bPack` fields.

`pszSaveAs` is an optional string pointer that identifies the destination table if it is different than the source table.

`pszKeyviolName` is an optional string pointer that identifies the table to which records that cause key violations during the restructure will be sent.

`pszProblemsName` is an optional string pointer that identifies the table to which records that cause problems during the restructure will be sent.

`bAnalyzeOnly` is unused.

The following code shows the `Pack()` method of `TParadoxTable`. You can see from the code how the `CRTblDesc` record is initialized and how the temporary database is created using the `DbiOpenDatabase()` function. Also note the `finally` block, which ensures that the temporary database is cleaned up after use.

```
procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Initialize TblDesc record }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // fill with 0s
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // set table name
    StrCopy(szTblType, szPARADOX);      // set table type
    bPack := True;                       // set pack flag
  end;
  { Store table active state.  Must close table to pack. }
  WasActive := Active;
  if WasActive then Close;
  try
    { Create a temporary database.  Must be read-write/exclusive }
    Check(DbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Pack the table }
      Check(DbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
            False));
    finally
      { Close the temporary database }
      DbiCloseDatabase(TempDBHandle);
    end;
  finally
    { Reset table active state }
    Active := WasActive;
  end;
end;
```

Listing 30.1 shows the DDGTbls unit in which the TdBaseTable and TParadoxTable objects are defined.

LISTING 30.1 The DDGTbls.pas Unit

```
unit DDGTbls;

interface

uses DB, DBTables, BDE;

type
  TdBaseTable = class(TTable)
  private
    FViewDeleted: Boolean;
    function GetIsDeleted: Boolean;
    function GetRecNum: Longint;
    procedure SetViewDeleted(Value: Boolean);
  protected
    function CreateHandle: HDBICur; override;
  public
    procedure Pack(RegenIndexes: Boolean);
    procedure UndeleteRecord;
    property IsDeleted: Boolean read GetIsDeleted;
    property RecNum: Longint read GetRecNum;
    property ViewDeleted: Boolean read FViewDeleted write SetViewDeleted;
  end;

  TParadoxTable = class(TTable)
  private
  protected
    function CreateHandle: HDBICur; override;
    function GetRecNum: Longint;
  public
    procedure Pack;
    property RecNum: Longint read GetRecNum;
  end;

implementation

uses SysUtils;

{ TdBaseTable }

function TdBaseTable.GetIsDeleted: Boolean;
{ Returns a boolean indicating whether or not the current record }
{ has been soft deleted. }
```

```
var
  RP: RECProps;
begin
  if not FViewDeleted then     // don't bother if they aren't viewing
    Result := False            // deleted records
  else begin
    UpdateCursorPos;           // update BDE from Delphi
    { Get current record properties }
    Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // return flag from properties
  end;
end;

function TdBaseTable.GetRecNum: Longint;
{ Returns the physical record number of the current record. }
var
  RP: RECProps;
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get current record properties }
  Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;     // return value from properties
end;

function TdBaseTable.CreateHandle: HDBICur;
{ Overridden from ancestor in order to perform a check to }
{ ensure that this is a dBASE table. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;         // do inherited
  if Result <> Nil then begin
    { Get cursor properties, and raise exception if the }
    { table isn't using the dBASE driver. }
    Check(DbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szdBASE) then
      raise EDatabaseError.Create('Not a dBASE table');
  end;
end;

procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Packs the table in order to removed soft deleted records }
{ from the file. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
```

continues

LISTING 30.1 Continued

```
  { Table must be active and opened exclusively }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Pack the table }
    Check(DbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { update Delphi from BDE }
    CursorPosChanged;
    Refresh;
  end;
end;

procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Allows the user to toggle between viewing and not viewing }
{ deleted records. }
begin
  { Table must be active }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;     // avoid flicker
    try
      { Magic BDE call to toggle view of soft deleted records }
      Check(DbiSetProp(hdbiObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // update Delphi
      EnableControls;    // flicker avoidance complete
    end;
    FViewDeleted := Value
  end;
end;

procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(DbiUndeleteRecord(Handle));
  Refresh;
end;

function TParadoxTable.CreateHandle: HDBICur;
{ Overridden from ancestor in order to perform a check to }
{ ensure that this is a Paradox table. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;         // do inherited
```

```
  if Result <> Nil then begin
    { Get cursor properties, and raise exception if the }
    { table isn't using the Paradox driver. }
    Check(DbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szPARADOX) then
      raise EDatabaseError.Create('Not a Paradox table');
  end;
end;

function TParadoxTable.GetRecNum: Longint;
{ Returns the sequence number of the current record. }
begin
  UpdateCursorPos;             // update BDE from Delphi
  { Get sequence number of current record into Result }
  Check(DbiGetSeqNo(Handle, Result));
end;

procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Initialize TblDesc record }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // fill with 0s
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // set table name
    szTblType := szPARADOX;              // set table type
    bPack := True;                       // set pack flag
  end;
  { Store table active state.  Must close table to pack. }
  WasActive := Active;
  if WasActive then Close;
  try
    { Create a temporary database.  Must be read-write/exclusive }
    Check(DbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
          dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Pack the table }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil,
            False));
    finally
      { Close the temporary database }
      DbiCloseDatabase(TempDBHandle);
    end;
  finally
```

continues

LISTING 30.1 Continued

```
    { Reset table active state }
    Active := WasActive;
  end;
end;

end.
```

Limiting TQuery Result Sets

Here's a classic SQL programming *faux pas*: Your application issues a SQL statement to the server that returns a result set consisting of a gazillion rows, thereby making the application user wait forever for the query to return and tying up precious server and network bandwidth. Conventional SQL wisdom dictates that one shouldn't issue queries that are so general that they cause so many records to be fetched. However, this is sometimes unavoidable, and `TQuery` doesn't seem to help matters much, because it doesn't provide a means for restricting the number of records in a result set to be fetched from the server. Fortunately, the BDE does provide this capability, and it's not very difficult to surface in a `TQuery` descendant.

The BDE API call that performs this magic is the catchall `DbiSetProp()` function, which was explained earlier in this chapter. In this case, the first parameter to `DbiSetProp()` is the cursor handle for the query, the second parameter must be `curMAXROWS`, and the final parameter should be set to the maximum number of rows to which you want to restrict the result set.

The ideal place to make the call to this function is in the `PrepareCursor()` method of `TQuery`, which is called immediately after the query is opened. Listing 30.2 shows the `ResQuery` unit, in which the `TRestrictedQuery` component is defined.

LISTING 30.2 The `ResQuery.pas` Unit

```
unit ResQuery;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBTables, BDE;

type
  TRestrictedQuery = class(TQuery)
  private
    FMaxRowCount: Longint;
  protected
    procedure PrepareCursor; override;
```

```
  published
    property MaxRowCount: Longint read FMaxRowCount write FMaxRowCount;
  end;

procedure Register;

implementation

procedure TRestrictedQuery.PrepareCursor;
begin
  inherited PrepareCursor;
  if FMaxRowCount > 0 then
    Check(DbiSetProp(hDBIObj(Handle), curMAXROWS, FMaxRowCount));
end;

procedure Register;
begin
  RegisterComponents('DDG', [TRestrictedQuery]);
end;

end.
```

You can limit the result set of a query by simply setting the `MaxRowCount` property to a value greater than zero. To further illustrate the point, Figure 30.1 shows the result of a query restricted to three rows, as shown in SQL Monitor.

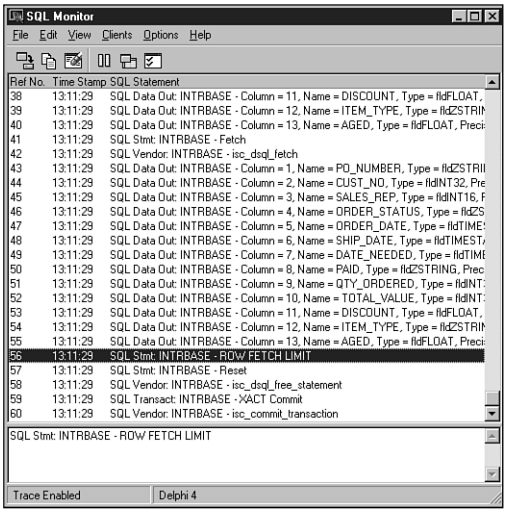

FIGURE 30.1
A restricted query viewed from SQL Monitor.

BDE Miscellany

Through our development of database applications, we've found a few common development tasks that could serve to be automated a bit. Some of these miscellaneous tasks include performing SQL aggregate functions on a table, copying tables, and obtaining a list of Paradox users for a particular session.

SQL Aggregate Functions

Generally speaking, SQL aggregate functions are functions built into the SQL language that perform some arithmetic operation on one or more columns from one or more rows. Some common examples of this are `sum()`, which adds columns from multiple rows, `avg()`, which calculates the average value of columns from multiple rows, `min()`, which finds the minimum value of columns in multiple rows, and `max()`, which (as you might guess) determines the maximum value of columns within multiple rows.

Aggregate functions such as these can sometimes be inconvenient to use in Delphi. For example, if you're working with `TTables` to access data, using these functions involves creating a `TQuery`, formulating the correct SQL statement for the table and column in question, executing the query, and obtaining the result from the query. Clearly this is a process crying out to be automated, and the code in Listing 30.3 does just that.

LISTING 30.3 Automating SQL Aggregate Functions

```
type
  TSQLAggFunc = (safSum, safAvg, safMin, safMax);

const
  // SQL aggregate functions
  SQLAggStrs: array[TSQLAggFunc] of string = (
    'select sum(%s) from %s',
    'select avg(%s) from %s',
    'select min(%s) from %s',
    'select max(%s) from %s');

function CreateQueryFromTable(T: TTable): TQuery;
// returns a query hooked to the same database and session as table T
begin
  Result := TQuery.Create(nil);
  try
    Result.DatabaseName := T.DatabaseName;
    Result.SessionName := T.SessionName;
  except
    Result.Free;
    Raise;
```

```
  end;
end;

function DoSQLAggFunc(T: TTable; FieldNames: string;
  Func: TSQLAggFunc): Extended;
begin
  with CreateQueryFromTable(T) do
  begin
    try
      SQL.Add(Format(SQLAggStrs[Func], [FieldNames, T.TableName]));
      Open;
      Result := Fields[0].AsFloat;
    finally
      Free;
    end;
  end;
end;

function SumField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safSum);
end;

function AvgField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safAvg);
end;

function MinField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMin);
end;

function MaxField(T: TTable; Field: string): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMax);
end;
```

As you can see from the listing, each of the individual aggregate function wrappers call into the `DoSQLAggFun()` function. In this function, the `CreateQueryFromTable()` function creates and returns a `TQuery` component that uses the same database and session as the `TTable` passed in the `T` parameter. The proper SQL string is then formatted from an array of strings, the query is executed, and the query's result is returned from the function.

Quick Table Copy

If you want to make a copy of a table, traditional wisdom might dictate a few different courses of action. The first might be to use the Win32 API's `CopyFile()` function to physically copy the table file(s) from one location to another. Another option is to use a `TBatchMove` component to copy one `TTable` to another. Yet another option is to use `TTable`'s `BatchMove()` method to perform the copy.

However, there are problems with each of these traditional alternatives: A brute-force file copy using the `CopyFile()` API function may not work if the table files are open by another process or user, and it certainly will not work if the table exists within some type of database file on a SQL server. A file copy might become a very complex task if you consider that you may also have to copy associated index, BLOB, or value files. The use of `TBatchMove` would solve these problems, but only if you submit to the disadvantage of the complexity involved in using this component. An additional drawback is the fact that the batch move process is much slower than a direct file copy. Using `TTable.BatchMove()` does help to alleviate the issue of complexity in performing the table copy, but it doesn't overcome the performance shortcomings inherent in the batch move process.

Fortunately, the BDE developers also recognized this issue and made a BDE API function available that provides the best of both worlds: speed and ease of use. The function in question is `DbiCopyTable()`, and it's declared as shown here:

```
function DbiCopyTable (            { Copy one table to another }
      hDb           : hDBIDb;      { Database handle }
      bOverWrite    : Bool;        { True, to overwrite existing file }
      pszSrcTableName : PChar;     { Source table name }
      pszSrcDriverType : PChar;    { Source driver type }
      pszDestTableName : PChar     { Destination table name }
   ): DBIResult stdcall;
```

Because the BDE API function can't deal directly with VCL `TTable` components, the following procedure wraps `DbiCopyTable()` into a nifty routine to which you can pass a `TTable` and a destination table name:

```
procedure QuickCopyTable(T: TTable; DestTblName: string;
  Overwrite: Boolean);
// Copies TTable T to an identical table with name DestTblName.
// Will overwrite existing table with name DestTblName if Overwrite is
// True.
var
  DBType: DBINAME;
  WasOpen: Boolean;
  NumCopied: Word;
begin
```

```
  WasOpen := T.Active;        // save table active state
  if not WasOpen then T.Open; // ensure table is open
  // Get driver type string
  Check(DbiGetProp(hDBIObj(T.Handle), drvDRIVERTYPE, @DBType,
    SizeOf(DBINAME), NumCopied));
  // Copy the table
  Check(DbiCopyTable(T.DBHandle, Overwrite, PChar(T.TableName), DBType,
    PChar(DestTblName)));
  T.Active := WasOpen;        // restore active state
end;
```

NOTE

For local databases (Paradox, dBASE, Access, and FoxPro), all files associated with the table—index and BLOB files, for example—are copied to the destination table. For tables residing in a SQL database, only the table will be copied, and it's up to you to ensure that the necessary indexes and other elements are applied to the destination table.

Paradox Session Users

If your application uses Paradox tables, you may come across a situation where you need to determine which users are currently using a particular Paradox table. You can accomplish this with the `DbiOpenUserList()` BDE API function. This function provides a BDE cursor for a list of users for the current session. The following procedure demonstrates how to use this function effectively:

```
procedure GetPDoxUsersForSession(Sess: TSession; UserList: TStrings);
// Clears UserList and adds each user using the same netfile as session
// Sess to the list.  If Sess = nil, then procedure works for default
// net file.
var
  WasActive: Boolean;
  SessHand: hDBISes;
  ListCur: hDBICur;
  User: UserDesc;
begin
  if UserList = nil then Exit;
  UserList.Clear;
  if Assigned(Sess) then
  begin
    WasActive := Sess.Active;
    if not WasActive then Sess.Open;
    Check(DbiStartSession(nil, SessHand, PChar(Sess.NetFileDir)));
```

```
  end
  else
    Check(DbiStartSession(nil, SessHand, nil));
  try
    Check(DbiOpenUserList(ListCur));
    try
      while DbiGetNextRecord(ListCur, dbiNOLOCK, @User, nil) =
        DBIERR_NONE do
        UserList.Add(User.szUserName);
    finally
      DbiCloseCursor(ListCur);  // close "user list table" cursor
    end;
  finally
    DbiCloseSession(SessHand);
    if Assigned(Sess) then Sess.Active := WasActive;
  end;
end;
```

The interesting thing about the `DbiOpenUserList()` function is that it creates a cursor for a table that's manipulated in the same manner as any other BDE table cursor. In this case, `DbiGetNextRecord()` is called repeatedly until the end of the table is reached. The record buffer for this table follows the format of the `UserDesc` record, which is defined in the BDE unit as follows:

```
type
  pUSERDesc = ^USERDesc;
  USERDesc = packed record           { User description }
    szUserName   : DBIUSERNAME;      { User Name }
    iNetSession  : Word;             { Net level session number }
    iProductClass: Word;             { Product class of user }
    szSerialNum  : packed array [0..21] of Char; { Serial number }
  end;
```

Each call to `DbiGetNextRecord()` fills a `UserDesc` record called `User`, and the `szUserName` field of that record is added to the `UserList string list`.

TIP

Note the use of the `try..finally` resource protection blocks in the `GetPDoxUsersForSession()` procedure. These ensure that both the BDE resources associated with the session and cursor are properly released.

Writing Data-Aware VCL Controls

Chapter 21, "Writing Delphi Custom Components," and Chapter 22, "Advanced Component Techniques," provided you with thorough coverage of component-building techniques and methodologies. One large topic that wasn't covered, however, is data-aware controls. Actually, there isn't much more to creating a data-aware control than there is to creating a regular VCL control, but a typical data-aware component is different in four key respects:

- Data-aware controls maintain an internal data link object. A descendant of `TDataLink`, this object provides the means by which the control communicates with a `TDataSource`. For data-aware controls that connect to a single field of a dataset, this is usually a `TFieldDataLink`. The control should handle the `OnDataChange` event of the data link in order to receive notifications when the field or record data has changed.
- Data-aware controls must handle the `CM_GETDATALINK` message. The typical response to this message is to return the data link object in the message's `Result` field.
- Data-aware controls should surface a property of type `TDataSource` so the control can be connected to a data source by which it will communicate with a dataset. By convention, this property is called `DataSource`. Controls that connect to a single field should also surface a string property to hold the name of the field to which it is connected. By convention, this property is called `DataField`.
- Data-aware controls should override the `Notification()` method of `TComponent`. By overriding this method, the data-aware control can be notified if the data source component connected to the control has been deleted from the form.

To demonstrate the creation of a simple data-aware control, Listing 30.4 shows the `DBSound` unit. This unit contains the `TDBWavPlayer` component, a component that plays WAV sounds from a BLOB field in a dataset.

LISTING 30.4 The `DBSound.pas` Unit

```
unit DBSound;

interface

uses Windows, Messages, Classes, SysUtils, Controls, Buttons, DB,
  DBTables, DbCtrls;

type
  EDBWavError = class(Exception);

  TDBWavPlayer = class(TSpeedButton)
  private
```

continues

LISTING 30.4 Continued

```
    FAutoPlay: Boolean;
    FDataLink: TFieldDataLink;
    FDataStream: TMemoryStream;
    FExceptOnError: Boolean;
    procedure DataChange(Sender: TObject);
    function GetDataField: string;
    function GetDataSource: TDataSource;
    function GetField: TField;
    procedure SetDataField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    procedure CMGetDataLink(var Message: TMessage); message
      CM_GETDATALINK;
    procedure CreateDataStream;
    procedure PlaySound;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Click; override;
    property Field: TField read GetField;
  published
    property AutoPlay: Boolean read FAutoPlay write FAutoPlay
      default False;
    property ExceptOnError: Boolean read FExceptOnError
      write FExceptOnError;
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
  end;

implementation

uses MMSystem;

const
  // Error strings
  SNotBlobField = 'Field "%s" is not a blob field';
  SPlaySoundErr = 'Error attempting to play sound';

constructor TDBWavPlayer.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);             // call inherited
  FDataLink := TFieldDataLink.Create;   // create field data link
```

```
  FDataLink.OnDataChange := DataChange; // get data link notifications
  FDataStream := TMemoryStream.Create;  // create worker memory stream
end;

destructor TDBWavPlayer.Destroy;
begin
  FDataStream.Free;
  FDataLink.Free;
  FDataLink := Nil;
  inherited Destroy;
end;

procedure TDBWavPlayer.Click;
begin
  inherited Click; // do default behavior
  PlaySound;       // play the sound
end;

procedure TDBWavPlayer.CreateDataStream;
// creates memory stream from wave file in blob field
var
  BS: TBlobStream;
begin
  // make sure it's a blob field
  if not (Field is TBlobField) then
    raise EDBWavError.CreateFmt(SNotBlobField, [DataField]);
  // create a blob stream
  BS := TBlobStream.Create(TBlobField(Field), bmRead);
  try
    // copy from blob stream to memory stream
    FDataStream.SetSize(BS.Size);
    FDataStream.CopyFrom(BS, BS.Size);
  finally
    BS.Free;  // free blob stream
  end;
end;

procedure TDBWavPlayer.PlaySound;
// plays wave sound loaded in memory stream
begin
  // make sure we are hooked to a dataset and field
  if (DataSource <> nil) and (DataField <> '') then
  begin
    // make sure data stream is created
    if FDataStream.Size = 0 then CreateDataStream;
    // Play the sound in the memory stream, raise exception on error
```

continues

LISTING 30.4 Continued

```
    if (not MMSystem.PlaySound(FDataStream.Memory, 0, SND_ASYNC or
      SND_MEMORY)) and FExceptOnError then
      raise EDBWavError.Create(SPlaySoundErr);
  end;
end;

procedure TDBWavPlayer.DataChange(Sender: TObject);
// OnChange handler FFieldDataLink.DataChange
begin
  // deallocate memory occupied by previous wave file
  with FDataStream do if Size <> 0 then SetSize(0);
  // if AutoPlay is on, the play the sound
  if FAutoPlay then PlaySound;
end;

procedure TDBWavPlayer.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  // do some required housekeeping
  if (Operation = opRemove) and (FDataLink <> nil) and
    (AComponent = DataSource) then DataSource := nil;
end;

function TDBWavPlayer.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBWavPlayer.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBWavPlayer.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TDBWavPlayer.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;
```

```
function TDBWavPlayer.GetField: TField;
begin
  Result := FDataLink.Field;
end;

procedure TDBWavPlayer.CMGetDataLink(var Message: TMessage);
begin
  Message.Result := Integer(FDataLink);
end;

end.
```

This component is a `TSpeedButton` descendant that, when pressed, can play a WAV sound residing in a database BLOB field. The `AutoPlay` property can also be set to `True`, which will cause the sound to play every time the user navigates to a new record in the table. When this property is set, it might also make sense to set the `Visible` property of the component to `False` so that a button doesn't appear visually on the form.

In the `FDataLink.OnChange` handler, `DataChange()`, the component works by extracting the BLOB field using a `TBlobStream` and copying the BLOB stream to a memory stream, `FDataStream`. When the sound is in a memory stream, you can play it using the `PlaySound()` Win32 API function.

Extending TDataSet

One of the marquee features of the database VCL is the abstract `TDataSet`, which provides the capability to manipulate non-BDE data sources within the database VCL framework.

In the Olden Days…

In previous versions of Delphi, the VCL database architecture was closed, making it was very difficult to manipulate non-BDE data sources using VCL components. Figure 30.2 illustrates the BDE-centric data set architecture found in Delphi 1 and 2.

As Figure 30.2 shows, `TDataSet` is essentially hard-coded for the BDE, and there's no room in this architecture for non-BDE data sources. Developers wanting to use non-BDE data sources within VCL had two choices:

- Creating a DLL that looks to VCL like the BDE but talks to a different type of data
- Throwing `TDataSet` out the window and writing their own dataset class and data-aware controls

Clearly, either of these options involves a very significant amount of work, and neither is a particularly elegant solution. Something had to be done.

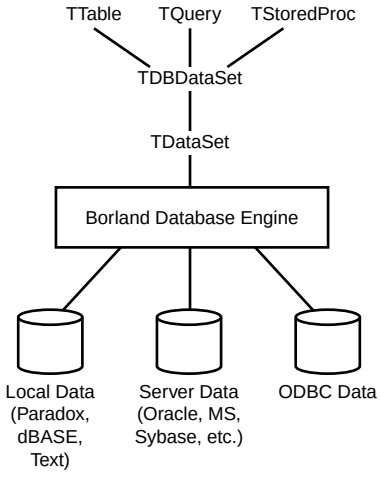

FIGURE 30.2
Delphi 1 and 2 VCL dataset architecture.

Modern Times

Recognizing these issues and the strong customer demand for easier access to non-BDE data sources, the Delphi development team made it a priority to extend VCL's data set architecture in Delphi 3. The idea behind the new architecture was to make the `TDataSet` class an abstraction of a VCL dataset and to move the BDE-specific data set code into the new `TBDEDataSet` class. Figure 30.3 provides an illustration of this new architecture.

When you understand how `TDataSet` was uncoupled from the BDE, the challenge becomes how to employ this concept to create a `TDataSet` descendant that manipulates some type of non-BDE data. And we're not using the term *challenge* loosely; creating a functional `TDataSet` descendant is not a task for the faint of heart. This is a fairly demanding task that requires familiarity with VCL database architecture and component writing.

TIP

Delphi provides two examples of creating a `TDataSet` descendant—one very simple and one very complicated. The simple example is the `TTextDataSet` class found in the `TextData` unit in the `\Delphi5\Demos\Db\TextData` directory. This example encapsulates `TStringList` as a one-field dataset. The complex example is the `TBDEDataSet` class found in the `DbTables` unit in the VCL source. As mentioned earlier, this class maps VCL's dataset architecture to the BDE.

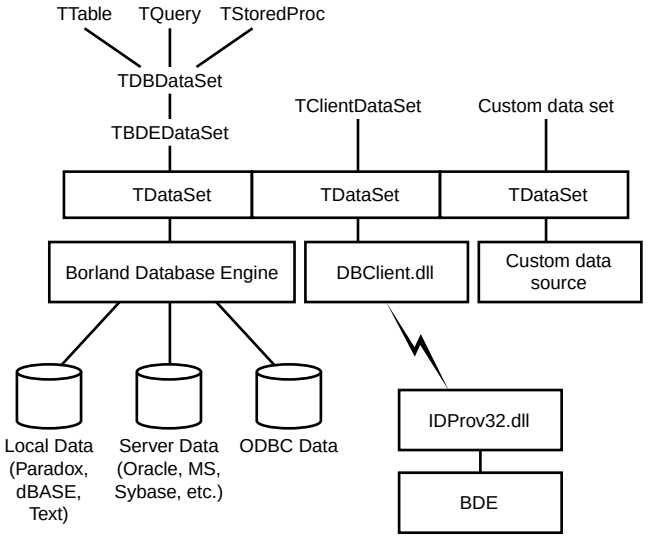

FIGURE 30.3
Delphi 3 and higher VCL dataset architecture.

Creating a TDataSet Descendant

Most dataset implementations will fall in between `TTextDataSet` and `TBDEDataSet` in terms of complexity. To provide an example of this, we'll demonstrate how to create a `TDataSet` descendant that manipulates an Object Pascal `file of record` (for a description of `file of record`, see Chapter 12, "Working With Files"). The following record and file type will be used for this example:

```
type
  // arbitrary-length array of char used for name field
  TNameStr = array[0..31] of char;

  // this record info represents the "table" structure:
  PDDGData = ^TDDGData;
  TDDGData = record
    Name: TNameStr;
    Height: Double;
    ShoeSize: Integer;
  end;

  // Pascal file of record which holds "table" data:
  TDDGDataFile = file of TDDGData;
```

An Object Pascal `file of record` can provide a convenient and efficient way to store information, but the format is inherently limited by its inability to insert records into or delete records from the middle of the file. For this reason, we'll use a two-file scheme to track the "table" information: the first, *data file*, being the `file of record`; the second, *index file*, maintaining a list of integers that represent seek values into the first file. This means that a record's position in the data file doesn't necessarily coincide with its position in the dataset. A record's position in the dataset is controlled by the order of the index file; the first integer in the index file contains the seek value of the first record into the data file, the second integer in the index file contains the next seek value into the data file, and so on.

In this section we'll discuss what's necessary to create a `TDataSet` descendant called `TDDGDataSet`, which communicates to this `file of record`.

TDataSet Abstract Methods

`TDataSet`, being an abstract class, is useless until you override the methods necessary for manipulation of some particular type of dataset. In particular, you must at least override each of `TDataSet`'s 23 abstract methods and perhaps some optional methods. For the sake of discussion, we've divided them into six logical groupings: record buffer methods, navigational methods, bookmark methods, editing methods, miscellaneous methods, and optional methods.

The following code shows an edited version of `TDataSet` as it is defined in `Db.pas`. For clarity, only the methods mentioned thus far are shown, and the methods are categorized based on the logical groupings we discussed. Here's the code:

```
type
  TDataSet = class(TComponent)
  { ... }
  protected
  { Record buffer methods }
    function AllocRecordBuffer: PChar; virtual; abstract;
    procedure FreeRecordBuffer(var Buffer: PChar); virtual; abstract;
    procedure InternalInitRecord(Buffer: PChar); virtual; abstract;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean):
      TGetResult; virtual; abstract;
    function GetRecordSize: Word; virtual; abstract;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
      override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); virtual;
      abstract;
  { Bookmark methods }
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
      override;
```

```
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
  { Navigational methods }
    procedure InternalFirst; virtual; abstract;
    procedure InternalLast; virtual; abstract;
  { Editing methods }
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
      virtual; abstract;
    procedure InternalDelete; virtual; abstract;
    procedure InternalPost; virtual; abstract;
  { Miscellaneous methods }
    procedure InternalClose; virtual; abstract;
    procedure InternalHandleException; virtual; abstract;
    procedure InternalInitFieldDefs; virtual; abstract;
    procedure InternalOpen; virtual; abstract;
    function IsCursorOpen: Boolean; virtual; abstract;
  { optional methods }
    function GetRecordCount: Integer; virtual;
    function GetRecNo: Integer; virtual;
    procedure SetRecNo(Value: Integer); virtual;
  { ... }
  end;
```

Record Buffer Methods

You must override a number of methods that deal with record buffers. Actually, VCL does a pretty good job of hiding the gory details of its record buffer implementation; `TDataSet` will create and manage groups of buffers, so your job is primarily to decide what goes in the buffers and to move data between different buffers. Because it's a requirement for all `TDataSet` descendants to implement bookmarks, we'll store bookmark information after the record data in the record buffer. The record we'll use to describe bookmark information is as follows:

```
type
  // Bookmark information record to support TDataset bookmarks:
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;
```

The `BookmarkData` field will represent a simple seek value into the data file. The `BookmarkFlag` field is used to determine whether the buffer contains a valid bookmark, and it will contain special values when the dataset is positioned on the BOF and EOF cracks.

NOTE

Keep in mind that this implementation of bookmarks and record buffers is specific to this solution. If you were creating a TDataSet descendant to manipulate some other type of data, you might choose to implement your record buffer or bookmarks differently. For example, the data source you're trying to encapsulate may natively support bookmarks.

Before examining the record buffer–specific methods, first take a look at the constructor for the `TDDGDataSet` class:

```
constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;
```

This constructor does three important things: First, it creates the `TIndexList` object. This list object is used as the index file described earlier to maintain order in the dataset. Next, the `FRecordSize` and `FBufferSize` fields are initialized. `FRecordSize` holds the size of the data record, and `FBufferSize` represents the total size of the record buffer (the data record size plus the size of the bookmark information record). Finally, this method calls the inherited constructor to perform the default `TDataSet` setup.

The following are `TDataSet` methods that deal with record buffers that must be overridden in a descendant. Except for `GetFieldData()`, all are declared as abstract in the base class:

```
function AllocRecordBuffer: PChar; override;
procedure FreeRecordBuffer(var Buffer: PChar); override;
procedure InternalInitRecord(Buffer: PChar); override;
function GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult; override;
function GetRecordSize: Word; override;
function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
procedure SetFieldData(Field: TField; Buffer: Pointer); override;
```

AllocRecordBuffer()

The `AllocRecordBuffer()` method is called to allocate memory for a single record buffer. In this implementation of the method, the `AllocMem()` function is used to allocate enough memory to hold both the record data and the bookmark data:

```
function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;
```

FreeRecordBuffer()

As you might expect, `FreeRecordBuffer()` must free the memory allocated by the `AllocRecordBuffer()` method. It's implemented using the `FreeMem()` procedure, as shown here:

```
procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;
```

InternalInitRecord()

The `InternalInitRecord()` method is called to initialize a record buffer. In this method, you can do things such as setting default field values and performing some type of initialization of custom record buffer data. In this case, we simply zero-initialize the record buffer:

```
procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;
```

GetRecord()

The primary function of the `GetRecord()` method is to retrieve the record data for either the previous, current, or next record in the dataset. The return value of this function is of type `TGetResult`, which is defined in the `Db` unit as follows:

```
TGetResult = (grOK, grBOF, grEOF, grError);
```

The meaning of each of the enumerations is pretty much self-explanatory: `grOk` means success, `grBOF` means the dataset is at the beginning, `grEOF` means the dataset is at the end, and `grError` means an error has occurred.

The implementation of this method is as follows:

```
function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
```

```
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;
```

The `FRecordPos` field tracks the current record position in the dataset. You'll notice that `FRecordPos` is incremented or decremented, as appropriate, when `GetRecord()` is called to obtain the next or previous record. If `FRecordPos` contains a valid record number, `FRecordPos` is used as an index into `FIndexList`. The number at that index is a seek value into the data file, and the record data is read from that position in the data file into the buffer specified by the `Buffer` parameter.

`GetRecord()` also has one additional job: When the `DoCheck` parameter is `True` and `grError` is the potential return value, an exception should be raised.

GetRecordSize()

The `GetRecordSize()` method should return the size, in bytes, of the record data portion of the record buffer. Be careful not to return the size of the entire record buffer; just return the size of the data portion. In this implementation, we return the value of the `FRecordSize` field:

```
function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;
```

GetFieldData()

The `GetFieldData()` method is responsible for copying data from the active record buffer (as provided by the `ActiveBuffer` property) into a field buffer. This is often accomplished most expediently using the `Move()` procedure. You can differentiate which field to copy using `Field`'s `Index` or `Name` property. Also, be sure to copy from the correct offset into `ActiveBuffer` because `ActiveBuffer` contains a complete record's data and `Buffer` only holds one field's data. This implementation copies the fields from the internal buffer structure to its respective TField:

```
function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer):
  Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;
```

Both this method and `SetFieldData()` can become much more complex if you want to support more advanced features such as calculated fields and filters.

SetFieldData()

The purpose of `SetFieldData()` is inverse to that of `GetFieldData()`; `SetFieldData()` copies data from a field buffer into the active record buffer. As you can see from the following code, the implementations of these two methods are very similar:

```
procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
```

```
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;
```

After the data is copied, the `DataEvent()` method is called to signal that a field has changed and fire the `OnChange` event of the field.

Bookmark Methods

We mentioned earlier that bookmark support is required for `TDataSet` descendants. The following abstract methods of `TDataSet` are overridden to provide this support:

```
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure InternalGotoBookmark(Bookmark: Pointer); override;
procedure InternalSetToRecord(Buffer: PChar); override;
```

For `TDDGDataSet`, you'll see that the implementations of these methods revolve mostly around manipulating the bookmark information tacked onto the end of the record buffer.

GetBookmarkFlag() and SetBookmarkFlag()

Bookmark flags are used internally by `TDataSet` to determine whether a particular record is the first or last in the dataset. For this purpose, you must override the `GetBookmarkFlag()` and `SetBookmarkFlag()` methods. The `TDDGDataSet` implementation of these methods reads from and writes to the record buffer to keep track of this information, as shown here:

```
function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;
```

GetBookmarkData() and SetBookmarkData()

The `GetBookmarkData()` and `SetBookmarkData()` methods provide a means for `TDataSet` to manipulate a record's bookmark data without repositioning the current record. As you can see, these methods are implemented in a manner similar to the methods described in the preceding example:

```
procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ :=PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData :=PInteger(Data)^;
end;
```

InternalGotoBookmark()

The `InternalGotoBookmark()` method is called to reposition the current record to that represented by the `Bookmark` parameter. Because a bookmark value is the same as the record number for `TDDGDataSet`, the implementation of this method is straightforward:

```
procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;
```

InternalSetToRecord()

`InternalSetToRecord()` is similar to `InternalGotoBookmark()` except that it receives as a parameter a record buffer instead of a bookmark value. The job of this method is to position the dataset to the record provided in the `Buffer` parameter. This implementation of a record buffer contains the bookmark information because the bookmark value is the same as the record position; therefore, the implementation of this method is a one-liner:

```
procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // bookmark value is the same as an offset into the file
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;
```

Navigational Methods

You must override several abstract navigational methods in `TDataSet` in order to position the dataset on the first or last record:

```
procedure InternalFirst; override;
procedure InternalLast; override;
```

The implementations of these methods are quite simple; `InternalFirst()` sets the `FRecordPos` value to `-1` (the BOF crack value), and `InternalLast()` sets the record position to the record count. Because the record index is zero based, the count is 1 greater than the last index (the EOF crack). Here's an example:

```
procedure TDDGDataSet.InternalFirst;
begin
```

```
  FRecordPos := -1;
end;

procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;
```

Editing Methods

Three abstract `TDataSet` methods must be overridden in order to allow for the editing, appending, inserting, and deleting of records:

```
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
procedure InternalDelete; override;
procedure InternalPost; override;
```

InternalAddRecord()

`InternalAddRecord()` is called when a record is inserted or appended to the dataset. The `Buffer` parameter points to the record buffer to be added to the dataset, and the `Append` parameter is `True` when a record is being appended and `False` when a record is being inserted. The `TDDGDataSet` implementation of this method seeks to the end of the data file, writes the record data to the file, and then adds or inserts the data file seek value into the appropriate position in the index list:

```
procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer;
  Append: Boolean);
var
RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

InternalDelete()

The `InternalDelete()` method deletes the current record from the dataset. Because it's not practical to remove a record from the middle of the data file, the current record is deleted from

the index list. This, in effect, orphans the deleted record in the data file by removing the index entry for a data record. Here's an example:

```
procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;
```

NOTE

This method of deletion means that the data file will not shrink in size even as records are deleted (similar to dBASE files). If you intend to use this type of dataset for commercial work, a good addition would be a file-pack method, which removes orphaned records from the data file.

InternalPost()

The `InternalPost()` method is called by `TDataSet.Post()`. In this method, you should write the data from the active record buffer to the data file. You'll note that the implementation of this method is quite similar to that of `InternalAddRecord()`, as shown here:

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

Miscellaneous Methods

Several other abstract methods must be overridden in order to create a working `TDataSet` descendant. These are general housekeeping methods, and because these methods can't be

pigeonholed into a particular category, we'll call them miscellaneous methods. These methods are as follows:

```
procedure InternalClose; override;
procedure InternalHandleException; override;
procedure InternalInitFieldDefs; override;
procedure InternalOpen; override;
function IsCursorOpen: Boolean; override;
```

InternalClose()

`InternalClose()` is called by `TDataSet.Close()`. In this method, you should deallocate all resources associated with the dataset that were allocated by `InternalOpen()` or that were allocated throughout the course of using the dataset. In this implementation, the data file is closed, and we ensure that the index list has been persisted to disk. Additionally, the `FRecordPos` is set to the BOF crack, and the data file record is zeroed out:

```
procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;
```

InternalHandleException()

`InternalHandleException()` is called if an exception is raised while this component is being read from or written to a stream. Unless you have a specific need to handle these exceptions, you should implement this method as follows:

```
procedure TDDGDataSet.InternalHandleException;
begin
  // standard implementation for this method:
  Application.HandleException(Self);
end;
```

InternalInitFieldDefs()

In the `InternalInitFieldDefs()` method is where you should define the fields contained in the dataset. This is done by instantiating `TFieldDef` objects, passing the `TDataSet`'s `FieldDefs` property as the `Owner`. In this case, three `TFieldDef` objects are created, representing the three fields in this dataset:

```
procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // create FieldDefs which map to each field in the data record
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False,
   1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

InternalOpen()

The `InternalOpen()` method is called by `TDataSet.Open()`. In this method, you should open the underlying data source, initialize any internal fields or properties, create the field defs if necessary, and bind the field defs to the data. The following implementation of this method opens the data file, loads the index list from a file, initializes the `FRecordPos` field and the `BookmarkSize` property, and creates and binds the field defs. You'll see from the following code that this method also gives the user a chance to create the database files if they aren't found on disk:

```
procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // make sure table and index files exist
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // open data file
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
```

```
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); //initialize index TList from file
    FRecordPos := -1;                  //initial record pos before BOF
    BookmarkSize := SizeOf(Integer);   //initialize bookmark size for VCL
    InternalInitFieldDefs;             //initialize FieldDef objects
    // Create TField components when no persistent fields have been
    // created
    if DefaultFields then CreateFields;
    BindFields(True);                  //bind FieldDefs to actual data
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```

NOTE

Any resource allocations made in `InternalOpen()` should be freed in `InternalClose()`.

IsCursorOpen()

The `IsCursorOpen()` method is called internal to `TDataSet` while the dataset is being opened in order to determine whether data is available even though the dataset is inactive. The `TDDGData` implementation of this method returns `True` only if the data file has been opened, as shown here:

```
function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Cursor" is open if data file is open.  File is open if FDataFile's
  // Mode includes the FileMode in which the file was open.
  Result := TFileRec(FDataFile).Mode <> 0;
end;
```

TIP

The preceding method illustrates an interesting feature of Object Pascal: a *file of record* or untyped file can be typecast to a `TFileRec` in order to obtain low-level information about the file. `TFileRec` is described in Chapter 12, "Working with Files."

Optional Record Number Methods

If you want to take advantage of `TDBGrid`'s ability to scroll relative to the cursor position in the dataset, you must override three methods:

```
function GetRecordCount: Integer; override;
function GetRecNo: Integer; override;
procedure SetRecNo(Value: Integer); override;
```

Although this feature makes sense for this implementation, in many cases this capability isn't practical or even possible. For example, if you're working with a huge amount of data, it might not be practical to obtain a record count, or if you're communicating with a SQL server, this information might not even be available.

This `TDataSet` implementation is fairly simple, and these methods are appropriately straightforward to implement:

```
function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;
```

TDDGDataSet

Listing 30.5 shows the `DDG_DS` unit, which contains the complete implementation of the `TDDGDataSet` unit.

LISTING 30.5 The DDG_DS.pas Unit

```
unit DDG_DS;

interface

uses Windows, Db, Classes, DDG_Rec;

type

  // Bookmark information record to support TDataset bookmarks:
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;

  // List used to maintain access to file of record:
  TIndexList = class(TList)
  public
    procedure LoadFromFile(const FileName: string); virtual;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToFile(const FileName: string); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
  end;

  // Specialized DDG TDataset descendant for our "table" data:
  TDDGDataSet = class(TDataSet)
  private
    function GetDataFileSize: Integer;
  public
    FDataFile: TDDGDataFile;
    FIdxName: string;
    FIndexList: TIndexList;
    FTableName: string;
    FRecordPos: Integer;
    FRecordSize: Integer;
    FBufferSize: Integer;
    procedure SetTableName(const Value: string);
  protected
    { Mandatory overrides }
    // Record buffer methods:
    function AllocRecordBuffer: PChar; override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    procedure InternalInitRecord(Buffer: PChar); override;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean): TGetResult; override;
```

```
  function GetRecordSize: Word; override;
  procedure SetFieldData(Field: TField; Buffer: Pointer); override;
  // Bookmark methods:
  procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
  function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
  procedure InternalGotoBookmark(Bookmark: Pointer); override;
  procedure InternalSetToRecord(Buffer: PChar); override;
  procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
    override;
  procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
  // Navigational methods:
  procedure InternalFirst; override;
  procedure InternalLast; override;
  // Editing methods:
  procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
    override;
  procedure InternalDelete; override;
  procedure InternalPost; override;
  // Misc methods:
  procedure InternalClose; override;
  procedure InternalHandleException; override;
  procedure InternalInitFieldDefs; override;
  procedure InternalOpen; override;
  function IsCursorOpen: Boolean; override;
  { Optional overrides }
  function GetRecordCount: Integer; override;
  function GetRecNo: Integer; override;
  procedure SetRecNo(Value: Integer); override;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
    override;

  // Additional procedures
  procedure EmptyTable;
published
  property Active;
  property TableName: string read FTableName write SetTableName;
  property BeforeOpen;
  property AfterOpen;
  property BeforeClose;
  property AfterClose;
  property BeforeInsert;
  property AfterInsert;
  property BeforeEdit;
```

continues

LISTING 30.5 Continued

```
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnDeleteError;
    property OnEditError;

    // Additional Properties
    property DataFileSize: Integer read GetDataFileSize;
  end;

procedure Register;

implementation

uses BDE, DBTables, SysUtils, DBConsts, Forms, Controls, Dialogs;

const
  feDDGTable = '.ddg';
  feDDGIndex = '.ddx';
  // note that file is not being locked!

{ TIndexList }

procedure TIndexList.LoadFromFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  try
    LoadFromStream;
  finally
    F.Free;
  end;
end;

procedure TIndexList.LoadFromStream(Stream: TStream);
var
  Value: Integer;
begin
```

```
  while Stream.Position < Stream.Size do
  begin
    Stream.Read(Value, SizeOf(Value));
    Add(Pointer(Value));
  end;
  ShowMessage(IntToStr(Count));
end;

procedure TIndexList.SaveToFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  try
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TIndexList.SaveToStream(Stream: TStream);
var
  i: Integer;
  Value: Integer;
begin
  for i := 0 to Count - 1 do
  begin
    Value := Integer(Items[i]);
    Stream.Write(Value, SizeOf(Value));
  end;
end;

{ TDDGDataSet }

constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;

destructor TDDGDataSet.Destroy;
begin
  inherited Destroy;
  FIndexList.Free;
```

continues

LISTING 30.5 Continued

```
end;

function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;

procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;

procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;

function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
 if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
           Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
```

```
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;

function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;

function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer):
  Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;

procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;

procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
```

continues

LISTING 30.5 Continued

```
begin
  PInteger(Data)^ :=PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;

procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // bookmark value is the same as an offset into the file
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData :=PInteger(Data)^;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;

procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
end;

procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // create FieldDefs which map to each field in the data record
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False,
    1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

```
procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;

procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;

procedure TDDGDataSet.InternalHandleException;
begin
  // standard implementation for this method:
  Application.HandleException(Self);
end;

procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;

procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer;
  Append: Boolean);
var
  RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
```

continues

LISTING 30.5 Continued

```
  end;
  FIndexList.SaveToFile(FIdxName);
end;

procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // make sure table and index files exist
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found.  Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // open data file
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); //initialize index TList from file
    FRecordPos := -1;                  //initial record pos before BOF
    BookmarkSize := SizeOf(Integer);   //initialize bookmark size for VCL
    InternalInitFieldDefs;             //initialize FieldDef objects
    // Create TField components when no persistent fields have been
    // created
    if DefaultFields then CreateFields;
    BindFields(True);                  //bind FieldDefs to actual data
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;

function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Cursor" is open if data file is open.  File is open if FDataFile's
  // Mode includes the FileMode in which the file was open.
  Result := TFileRec(FDataFile).Mode <> 0;
end;

function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
```

continues

LISTING 30.5 Continued

```
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;

procedure TDDGDataSet.SetTableName(const Value: string);
begin
  CheckInactive;
  FTableName := Value;
  if ExtractFileExt(FTableName) = '' then
    FTableName := FTableName + feDDGTable;

  FIdxName := ChangeFileExt(FTableName, feDDGIndex);

end;

procedure Register;
begin
  RegisterComponents('DDG', [TDDGDataSet]);
end;


function TDDGDataSet.GetDataFileSize: Integer;
begin
  Result := FileSize(FDataFile);
end;

procedure TDDGDataSet.EmptyTable;
var
  HFile: THandle;
begin
  Close;

  DeleteFile(FTableName);
  HFile := FileCreate(FTableName);
  FileClose(HFile);

  DeleteFile(FIdxName);
  HFile := FileCreate(FIdxName);
  FileClose(HFile);

  Open;
end;

end.
```

Summary

This chapter demonstrated how to extend your Delphi database applications to incorporate features that aren't encapsulated by VCL. Additionally, you learned some of the rules and processes for making direct calls into the BDE from Delphi applications. You also learned the specifics for extending the behavior of `TTable` with regard to dBASE and Paradox tables. Finally, you went step by step through the challenging process of creating a working `TDataSet` descendant. In the next chapter, "Internet-Enabling your Applications with WebBroker," you'll learn how to create server-side applications for the Web and deliver data to Web clients in real time.

Error Messages and Exceptions

APPENDIX

A

IN THIS APPENDIX

One difference between good software and great software is that whereas good software runs well, great software runs well and *fails* well. In Delphi programs, errors that are detected at runtime usually are reported and handled as exceptions. This allows your code the opportunity to respond to problems and recover (by backing up and trying another approach) or at least to "degrade gracefully" (free allocated resources, close files, and display an error message), instead of just crashing and making a mess of your system. Most exceptions in Delphi programs are raised and handled completely within the program; very few runtime errors actually will bring a Delphi program to a screeching halt.

This appendix lists the most common error messages that a Delphi application can report and provides field notes to help you find the cause of the error condition. Because each component you add to your Delphi environment often has its own set of error messages, this list can never be complete, so we'll focus on the most common or most insidious error messages you're likely to face while developing and debugging your Delphi applications.

Layers of Handlers, Layers of Severity

Every Delphi program has two default exception handlers, one below the other. VCL provides the default exception handler you'll see most of the time. VCL wraps an exception handler around the window procedure entry points of every VCL object. If an exception occurs while your program is responding to a Windows message (which is what your program spends 99 percent of its lifetime doing) and the exception is not handled by your code or a VCL component, the exception eventually will wind up stopping at the VCL default exception handler in the window procedure. That exception handler calls `Application.HandleException`, which will show the exception instance's text message to the user in a pop-up message box. After that, your program continues running and processing additional window messages.

The lowest-level exception handler lives at the heart of the Delphi RTL, several subbasements below the default VCL exception handler. If an exception occurs outside the context of message processing—such as during program startup or shutdown or during the execution of the VCL default exception handler—and the exception goes unhandled, it eventually will wind up stopping at the RTL default exception handler. At this level, there's no recourse for recovery—no message loop to keep things going. When activated, the RTL default exception handler displays a detailed error message to the user and then terminates the application.

In addition to the exception message text, the RTL default exception handler also reports the address of the code that raised the exception, in the form of a hexadecimal address. Use the Search, Find Error option in the Delphi IDE and enter this address in the dialog box. Delphi will move the cursor to the place in your source code that corresponds to this address, if it can locate the address and the source code.

If Delphi responds with "Address Not Found," this could mean that the error occurred in another module (for example, a "wild" pointer overwrote memory in use by some other application). More often, however, "Address Not Found" indicates that you have disabled line-number information in the unit that the address corresponds to (`{$D-}`) or that you don't have source code for that unit. Double-check that you're compiling your project with compiler debug info enabled, in the Project, Options dialog box, Compiler page, Debugging section. While you have the Project, Options dialog box open, check to see that the search path on the Directories/Conditionals page contains all the source code directories you want to use during debugging. If the Delphi IDE can't find a source file, it can't show you the source code line that corresponds to the exception error address. Use Project, Build All to recompile all your units with the new compiler settings.

Runtime Errors

This section will give you some pointers on what you should do when you experience errors in the form of exceptions or Win32 API function failures. These types of errors are rarely fatal, but you should know how to tackle them when the need arises.

Exceptions

Here we describe additional exceptions that Delphi's VCL components can raise. Keep in mind that custom components and your own code can (and often should) define additional exception classes specific to the task at hand.

Several of the exception classes listed here describe related error conditions: families of errors. The relationship of the exception classes to each other is captured by creating a general-purpose exception class to represent the entire family and specific exception classes that inherit from the general-purpose class. When you want to handle all errors in that family the same way, use the general-purpose exception class in the `on` clause of your `except` block. When you want to handle only certain specific errors from that family, use the specific exception classes in `on` clauses in your `except` block.

In the following list, we use indentation to group related exception classes together beneath their common generic ancestor class:

- `Exception`. This is the ancestor of all exception classes. There is nothing wrong with using this class to raise exceptions in quick-and-dirty code, but in production code, you'll want to be able to distinguish between the multitude of families of errors that your application can encounter. The best way to distinguish a family of related error conditions from the rest of the pack is to use a custom exception class to report those related errors.

- `EAbort`. Referred to as Delphi's "silent" exception, this exception is trapped by the VCL default exception handler, but VCL does not inform the user that the exception occurred. Use `EAbort` when you want to take advantage of the exception's capability to abort and unwind out of a complicated process, but you don't want the user to see an error message. Remember, the terms *exception* and *error* are *not* equivalent: exceptions are a means of changing program flow to facilitate error handling (among other things).
- `EAccessViolation`. An access violation has occurred in the operating system. Usually caused by a `Nil` or "wild" pointer.
- `EAssertionFailed`. The statement passed to the `Assert()` procedure evaluated to `False`.
- `EBitsError`. Raised when the `Bits` or `Size` property of a `TBits` object is out of bounds.
- `EComponentError`. This exception is raised in two situations. The first situation is when you use `RegisterClasses()` to attempt to register a component outside the `Register()` procedure. The second is when the name of your component is invalid or not unique.
- `EControlC`. The user has interrupted with the Ctrl+C key combination. This exception only occurs within console-mode applications.
- `EDbEditError`. The user entered text into a `TMaskEdit` or `TDbEdit` component that's incompatible with the current edit mask.
- `EDdeError`. An error occurred during a DDE operation with any of the `TDdeClientConv`, `TDdeClientItem`, `TDdeServerConv`, and `TDdeServerItem` components.
- `EExternalException`. This exception occurs when an unrecognized exception is raised by the operating system.
- `EInOutError`. This exception is raised when any I/O error occurs in your program. This exception will only occur when I/O checking is enabled using `{$I+}` in code or by enabling I/O Checking on the Compiler page of the Project Options dialog in IDE.
- `EIntError`. This is the ancestor of all integer math exceptions. Here are the descendents of this class:
  - `EDivByZero`. This exception is raised when you divide an integral number by zero. This exception is raised as a result of runtime error 200. This code example will cause an `EDivByZero` exception:

    ```
    var
      I: integer;
    begin
      I := 0;
      I := 10 div I;   { exception raised here }
    end;
    ```

  - `EIntOverflow`. This exception is raised when you attempt to perform an operation that overflows an integral variable beyond that variable type's capacity. This excep-

tion is raised as a result of runtime error 215. This exception will only be raised if overflow checking is enabled using `{$Q+}` in code or by enabling Overflow Checking on the Compiler page of the Project Options dialog in the IDE. The following code will cause this exception to be raised:

```
var
  l: longint;
begin
  l := MaxLongint;
  l := l * l;   { exception raised here }
end;
```

- `ERangeError`. This exception is raised when you attempt to index an array beyond its declared bounds or when you attempt to store a value that's too large in an integral type variable. This exception is raised as a result of runtime error 201. Range checking must be enabled with `{$R+}` in code or by enabling Range Checking on the Compiler page of the Project Options dialog in the IDE for this error to occur. The following example will cause Delphi to raise this exception:

```
var
  a: array[1..16] of integer;
  i: integer;
begin
  i := 17;
  a[i] := 1;  { exception raised here }
end;
```

- `EIntfCastError`. An attempt was made to cast an object or interface to an unsupported interface.
- `EInvalidCast`. This exception is raised when you attempt to use the `as` operator to typecast a class to an incompatible class. This exception is raised as a result of runtime error 219. The following code will cause this exception to be raised:

```
var
  B: TObject;
begin
  B := TButton.Create(nil);
  { exception raised here - TMemo is not an ancestor of TButton }
  with B as TMemo do
  ...
end;
```

- `EInvalidGraphic`. This exception is raised when you attempt to use `LoadFromFile()` on a file that's not a compatible graphics format in a class expecting a graphics file.
- `EInvalidGraphicOperation`. This exception is raised when you attempt to perform an illegal operation on a graphic object. For example, resizing a `TIcon` is illegal.

- `EInvalidOperation`. This exception occurs when you try to display or perform any other operation that requires a window handle on a control without a parent. Here's an example:

```
var
  b: TBitBtn;
begin
  b := TBitBtn.Create(Self);
  b.SetFocus;   { exception raised here }
end;
```

- `EInvalidPointer`. This exception is raised usually when you attempt to free an invalid or already-freed portion of memory in a call to `Dispose()`, `FreeMem()`, or a class destructor. This example causes an `EInvalidPointer` exception to be raised:

```
var
  p: pointer;
begin
  GetMem(p, 8);
  FreeMem(p, 8);
  FreeMem(p, 8);   { exception raised here }
end;
```

- `EListError`. This exception will be raised if you try to index past the end of a `TList` descendant. Here's an example:

```
var
  S: TStringList;
  Strng: String;
begin
  S := TStringList.Create;
  S.Add('One String');
  Strng := S.Strings[2]; { exception raised here }
end;
```

- `EMathError`. This is the ancestor object from which the following floating-point exceptions are derived:
  - `EInvalidOp`. This exception is raised when an invalid instruction is sent to the numeric coprocessor. This exception is uncommon unless you control the coprocessor directly with BASM code.
  - `EOverflow`. This exception is raised as a result of floating-point overflow (that is, when a value becomes too large to hold in a floating point variable). This exception corresponds to runtime error 205.
  - `EUnderflow`. Raised as a result of floating-point underflow (that is, when a value becomes too small to hold in a floating point variable). This exception corresponds to runtime error 206.
  - `EZeroDivide`. Raised when a floating point number is divided by zero.

- `EMCIDeviceError`. This exception indicates that an error occurred in the `TMediaPlayer` component. Most commonly, this exception is raised when the user attempts to play some media whose type is unsupported by the hardware.
- `EMenuError`. This is a generic exception that occurs in almost any error condition involving a `TMenu`, `TMenuItem`, or `TPopupMenu` component.
- `EOleCtrlError`. This exception is reserved for ActiveX control wrapper errors, but it's currently not being used in VCL.
- `EOleError`. This exception is raised when an OLE Automation error occurs.
    - `EOleSysError`. This exception is raised by the `OleCheck()` and `OleError()` routines when an error occurs while calling an OLE API function.
        - `EOleException`. Raised when an error occurs inside of a `safecall` function or procedure.
- `EOutlineError`. This is a generic exception that's raised when an error occurs while working with a `TOutline` component.
- `EOutOfMemory`. This exception is raised when you call `New()`, `GetMem()`, or a class constructor and not enough memory is available on the heap for the allocation. This exception corresponds to runtime error 203.
    - `EOutOfResources`. This exception occurs when Windows cannot fill an allocation request for a Windows resource, such as a window handle. This exception often reflects bugs in your video driver, especially if you're running in a high-color (32KB or 64KB colors) mode. If this error goes away when you switch to using the standard Windows VGA driver or to a lesser mode of your normal video driver, it's very likely that you've found a bug in your video driver. Contact your video card manufacturer for a driver update.
- `EPackageError`. Raised when an error occurs loading, initializing, or finalizing a package.
- `EParserError`. Raised when Delphi is unable to parse your text form file back to the binary DFM format. Generally, this is the result of a syntax error while editing the form in the IDE.
- `EPrinter`. This is a generic exception that will be raised when an error occurs while you're trying to use the `TPrinter` object.
- `EPrivilege`. This exception indicates that an attempt was made to execute a privileged instruction.
- `EPropertyError`. This exception is raised when an error occurs inside of a component property editor.
- `ERegistryException`. The `TRegistry` and `TRegIniFile` objects raise this exception when an error occurs while reading from or writing to the system Registry.

- `EStackOverflow`. This exception represents a serious operating system–level error in management of the stack. This error should be rare since an application's stack is dynamically expanded as needed by the operating system, but can occur in low-memory conditions.
- `EReportError`. This is a generic exception for an error that occurs while working with a report component.
- `EResNotFound`. This exception is raised when there are problems loading a form from a DFM file. This exception usually indicates that you've edited the DFM file to make it invalid, the DFM or EXE file has become corrupted, or the DFM file was not linked into the EXE. Make sure you haven't deleted or altered the `{$R *.DFM}` directive in your form unit.
- `EStreamError`. This exception is the base class of all stream exceptions. This exception usually indicates a problem loading a `TStrings` from a stream or setting the capacity of a memory stream. The following descendent exception classes signal other specific error conditions:
  - `EFCreateError`. Raised when an error occurs while creating a stream file. This exception often indicates that a file can't be created because the filename is invalid or in use by another process.
  - `EFilerError`. This exception is raised when you attempt to register the same class twice using the `RegisterClasses()` procedure. This class also serves as the base for other filer-related exceptions:
    - `EClassNotFound`. This exception is raised when Delphi reads a component class name from a stream but cannot find a declaration for the component in its corresponding unit. Remember that code and declarations that are not used by a program will not be copied into the EXE file by Delphi's smart linker.
    - `EInvalidImage`. This exception is raised when you attempt to read components from an invalid resource file.
    - `EMethodNotFound`. This exception is raised when a method specified in the DFM file or resource does not exist in the corresponding unit. This can happen if you've deleted code from the unit, recompiled the EXE, ignored the many warnings about the DFM file containing references to deleted code, and run the EXE anyway.
    - `EReadError`. This exception occurs when your application doesn't read the number of bytes from a stream that it's supposed to (for example, unexpected end of file) or when Delphi cannot read a property.

  - `EFOPenError`. This exception is raised when the specified stream file cannot be opened; it usually occurs when the file does not exist.
- `EStringListError`. This is a generic exception that's raised when an error condition results while working with a `TStringList` object.
- `EThread`. This is a `TThread`-related exception. Currently, this exception is only raised when a user attempts to call `Synchronize()` on a waiting thread.
- `ETreeViewError`. This exception is raised when you pass an invalid item index to a `TTreeView` method or property.
- `EWin32Error`. This exception is raised when an error occurs calling a Win32 API function. The message associated with this exception has error code and error string information.

Win32 System Errors

When you encounter an error in calling a Win32 API function or procedure, the error code is typically obtained by calling the `GetLastError()` function. Because the value returned from `GetLastError()` is a `DWORD` number, it's sometimes difficult to match that with an actual explanation of what might be the problem. To help you better decipher error codes, Table A.1 contains a list of the constant identifiers and values for the errors and a short description for each.

TABLE A.1 Win32 Error Codes

Constant	*Value*	*Description*
`ERROR_SUCCESS`	`0`	The operation completed successfully.
`ERROR_INVALID_FUNCTION`	`1`	The function is incorrect.
`ERROR_FILE_NOT_FOUND`	`2`	The system cannot find the file specified.
`ERROR_PATH_NOT_FOUND`	`3`	The system cannot find the path specified.
`ERROR_TOO_MANY_OPEN_FILES`	`4`	The system cannot open the file.
`ERROR_ACCESS_DENIED`	`5`	Access is denied.
`ERROR_INVALID_HANDLE`	`6`	The handle is invalid.
`ERROR_ARENA_TRASHED`	`7`	The storage control blocks were destroyed.
`ERROR_NOT_ENOUGH_MEMORY`	`8`	Not enough storage is available to process this command.
`ERROR_INVALID_BLOCK`	`9`	The storage control block address is invalid.
`ERROR_BAD_ENVIRONMENT`	`10`	The environment is incorrect.
`ERROR_BAD_FORMAT`	`11`	An attempt was made to load a program with an incorrect format.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_ACCESS	12	The access code is invalid.
ERROR_INVALID_DATA	13	The data is invalid.
ERROR_OUTOFMEMORY	14	Not enough storage is available to complete this operation.
ERROR_INVALID_DRIVE	15	The system cannot find the drive specified.
ERROR_CURRENT_DIRECTORY	$10	The directory cannot be removed.
ERROR_NOT_SAME_DEVICE	17	The system cannot move the file to a different disk drive.
ERROR_NO_MORE_FILES	18	There are no more files.
ERROR_WRITE_PROTECT	19	The media is write-protected.
ERROR_BAD_UNIT	20	The system cannot find the device specified.
ERROR_NOT_READY	21	The device is not ready.
ERROR_BAD_COMMAND	22	The device does not recognize the command.
ERROR_CRC	23	A data error (cyclic redundancy check) has occurred.
ERROR_BAD_LENGTH	24	The program issued a command, but the command length is incorrect.
ERROR_SEEK	25	The drive cannot locate a specific area or track on the disk.
ERROR_NOT_DOS_DISK	26	The specified disk or diskette cannot be accessed.
ERROR_SECTOR_NOT_FOUND	27	The drive cannot find the sector requested.
ERROR_OUT_OF_PAPER	28	The printer is out of paper.
ERROR_WRITE_FAULT	29	The system cannot write to the specified device.
ERROR_READ_FAULT	30	The system cannot read from the specified device.
ERROR_GEN_FAILURE	31	A device attached to the system is not functioning.
ERROR_SHARING_VIOLATION	$20	The process cannot access the file because another process is using it.
ERROR_LOCK_VIOLATION	33	The process cannot access the file because another process has locked a portion of the file.

Constant	*Value*	*Description*
ERROR_WRONG_DISK	34	The wrong diskette is in the drive. Insert %2 (Volume Serial Number: %3) into drive %1.
ERROR_SHARING_BUFFER_EXCEEDED	36	Too many files have been opened for sharing.
ERROR_HANDLE_EOF	38	The end of the file has been reached.
ERROR_HANDLE_DISK_FULL	39	The disk is full.
ERROR_NOT_SUPPORTED	50	The network request is not supported.
ERROR_REM_NOT_LIST	51	The remote computer is not available.
ERROR_DUP_NAME	52	A duplicate name exists on the network.
ERROR_BAD_NETPATH	53	The network path was not found.
ERROR_NETWORK_BUSY	54	The network is busy.
ERROR_DEV_NOT_EXIST	55	The specified network resource or device is no longer available.
ERROR_TOO_MANY_CMDS	56	The network BIOS command limit has been reached.
ERROR_ADAP_HDW_ERR	57	A network adapter hardware error occurred.
ERROR_BAD_NET_RESP	58	The specified server cannot perform the requested operation.
ERROR_UNEXP_NET_ERR	59	An unexpected network error has occurred.
ERROR_BAD_REM_ADAP	60	The remote adapter is not compatible.
ERROR_PRINTQ_FULL	61	The printer queue is full.
ERROR_NO_SPOOL_SPACE	62	Space to store the file waiting to be printed is not available on the server.
ERROR_PRINT_CANCELLED	63	Your file waiting to be printed was deleted.
ERROR_NETNAME_DELETED	$40	The specified network name is no longer available.
ERROR_NETWORK_ACCESS_DENIED	65	Network access is denied.
ERROR_BAD_DEV_TYPE	66	The network resource type is not correct.
ERROR_BAD_NET_NAME	67	The network name cannot be found.
ERROR_TOO_MANY_NAMES	68	The name limit for the local computer network adapter card was exceeded.
ERROR_TOO_MANY_SESS	69	The network BIOS session limit was exceeded.
ERROR_SHARING_PAUSED	70	The remote server has been paused or is in the process of being started.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_REQ_NOT_ACCEP	71	No more connections can be made to this remote computer at this time because there are already as many connections as the computer can accept.
ERROR_REDIR_PAUSED	72	The specified printer or disk device has been paused.
ERROR_FILE_EXISTS	80	The file exists.
ERROR_CANNOT_MAKE	82	The directory or file cannot be created.
ERROR_FAIL_I24	83	A failure has occurred on INT 24.
ERROR_OUT_OF_STRUCTURES	84	Storage to process this request is not available.
ERROR_OUT_OF_STRUCTURES	85	The local device name is already in use.
ERROR_INVALID_PASSWORD	86	The specified network password is not correct.
ERROR_INVALID_PARAMETER	87	The parameter is incorrect.
ERROR_NET_WRITE_FAULT	88	A write fault occurred on the network.
ERROR_NO_PROC_SLOTS	89	The system cannot start another process at this time.
ERROR_TOO_MANY_SEMAPHORES	100	Another system semaphore cannot be created.
ERROR_EXCL_SEM_ALREADY_OWNED	101	The exclusive semaphore is owned by another process.
ERROR_SEM_IS_SET	102	The semaphore is set and cannot be closed.
ERROR_TOO_MANY_SEM_REQUESTS	103	The semaphore cannot be set again.
ERROR_INVALID_AT_INTERRUPT_TIME	104	Exclusive semaphores cannot be requested at interrupt time.
ERROR_SEM_OWNER_DIED	105	The previous ownership of this semaphore has ended.
ERROR_SEM_USER_LIMIT	106	Insert the diskette for drive %1.
ERROR_DISK_CHANGE	107	The program has stopped because an alternate diskette was not inserted.
ERROR_DRIVE_LOCKED	108	The disk is in use or locked by another process.
ERROR_BROKEN_PIPE	109	The pipe has been ended.
ERROR_OPEN_FAILED	110	The system cannot open the device or file specified.

Constant	*Value*	*Description*
ERROR_BUFFER_OVERFLOW	111	The filename is too long.
ERROR_DISK_FULL	112	The disk does not contain enough space.
ERROR_NO_MORE_SEARCH_HANDLES	113	No more internal file identifiers are available.
ERROR_INVALID_TARGET_HANDLE	114	The target internal file identifier is incorrect.
ERROR_INVALID_CATEGORY	117	The IOCTL call made by the application program is not correct.
ERROR_INVALID_VERIFY_SWITCH	118	The verify-on-write switch parameter value is not correct.
ERROR_BAD_DRIVER_LEVEL	119	The system does not support the command requested.
ERROR_CALL_NOT_IMPLEMENTED	120	This function is only valid in Windows NT mode.
ERROR_SEM_TIMEOUT	121	The semaphore timeout period has expired.
ERROR_INSUFFICIENT_BUFFER	122	The data area passed to a system call is too small.
ERROR_INVALID_NAME	123	The filename, directory name, or volume label syntax is incorrect.
ERROR_INVALID_LEVEL	124	The system call level is not correct.
ERROR_NO_VOLUME_LABEL	125	The disk has no volume label.
ERROR_MOD_NOT_FOUND	126	The specified module could not be found.
ERROR_PROC_NOT_FOUND	127	The specified procedure could not be found.
ERROR_WAIT_NO_CHILDREN	$80	There are no child processes to wait for.
ERROR_CHILD_NOT_COMPLETE	129	The %1 application cannot run in Windows NT mode.
ERROR_DIRECT_ACCESS_HANDLE	130	An attempt was made to use a file handle to an open disk partition for an operation other than raw disk I/O.
ERROR_NEGATIVE_SEEK	131	An attempt was made to move the file pointer before the beginning of the file.
ERROR_SEEK_ON_DEVICE	132	The file pointer cannot be set on the specified device or file.
ERROR_IS_JOIN_TARGET	133	A JOIN or SUBST command cannot be used for a drive that contains previously joined drives.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_IS_JOINED	134	An attempt was made to use a JOIN or SUBST command on a drive that has already been joined.
ERROR_IS_SUBSTED	135	An attempt was made to use a JOIN or SUBST command on a drive that has already been substituted.
ERROR_NOT_JOINED	136	The system tried to delete the join of a drive that's not joined.
ERROR_NOT_SUBSTED	137	The system tried to delete the substitution of a drive that's not substituted.
ERROR_JOIN_TO_JOIN	138	The system tried to join a drive to a directory on a joined drive.
ERROR_SUBST_TO_SUBST	139	The system tried to substitute a drive to a directory on a substituted drive.
ERROR_JOIN_TO_SUBST	140	The system tried to join a drive to a directory on a substituted drive.
ERROR_SUBST_TO_JOIN	141	The system tried to perform a SUBST command on a drive to a directory on a joined drive.
ERROR_BUSY_DRIVE	142	The system cannot perform a JOIN or SUBST operation at this time.
ERROR_SAME_DRIVE	143	The system cannot join or substitute a drive to or for a directory on the same drive.
ERROR_DIR_NOT_ROOT	144	The directory is not a subdirectory of the root directory.
ERROR_DIR_NOT_EMPTY	145	The directory is not empty.
ERROR_IS_SUBST_PATH	146	The path specified is being used in a substitute.
ERROR_IS_JOIN_PATH	147	Not enough resources are available to process this command.
ERROR_PATH_BUSY	148	The path specified cannot be used at this time.
ERROR_IS_SUBST_TARGET	149	An attempt was made to join or substitute a drive for which a directory on the drive is the target of a previous substitute.

Constant	*Value*	*Description*
ERROR_SYSTEM_TRACE	150	System trace information was not specified in your CONFIG.SYS file or tracing is disallowed.
ERROR_INVALID_EVENT_COUNT	151	The number of specified semaphore events for DosMuxSemWait is not correct.
ERROR_TOO_MANY_MUXWAITERS	152	DosMuxSemWait did not execute; too many semaphores are already set.
ERROR_INVALID_LIST_FORMAT	153	The DosMuxSemWait list is not correct.
ERROR_LABEL_TOO_LONG	154	The volume label you entered exceeds the 11-character limit. The first 11 characters were written to disk. Any characters that exceeded the 11-character limit were automatically deleted.
ERROR_TOO_MANY_TCBS	155	Cannot create another thread.
ERROR_SIGNAL_REFUSED	156	The recipient process has refused the signal.
ERROR_DISCARDED	157	The segment is already discarded and cannot be locked.
ERROR_NOT_LOCKED	158	The segment is already unlocked.
ERROR_BAD_THREADID_ADDR	159	The address for the thread ID is not correct.
ERROR_BAD_ARGUMENTS	160	The argument string passed to DosExecPgm is not correct.
ERROR_BAD_PATHNAME	161	The specified path is invalid.
ERROR_SIGNAL_PENDING	162	A signal is already pending.
ERROR_MAX_THRDS_REACHED	164	No more threads can be created in the system.
ERROR_LOCK_FAILED	167	A region of a file cannot be locked.
ERROR_BUSY	170	The requested resource is in use.
ERROR_CANCEL_VIOLATION	173	A lock request was not outstanding for the supplied cancel region.
ERROR_ATOMIC_LOCKS_NOT_SUPPORTED	174	The file system does not support atomic changes to the lock type.
ERROR_INVALID_SEGMENT_NUMBER	180	The system detected a segment number that was not correct.
ERROR_INVALID_ORDINAL	182	The operating system cannot run %1.
ERROR_ALREADY_EXISTS	183	Cannot create a file when that file already exists.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_FLAG_NUMBER	186	The flag passed is not correct.
ERROR_SEM_NOT_FOUND	187	The specified system semaphore name was not found.
ERROR_INVALID_STARTING_CODESEG	188	The operating system cannot run %1.
ERROR_INVALID_STACKSEG	189	The operating system cannot run %1.
ERROR_INVALID_MODULETYPE	190	The operating system cannot run %1.
ERROR_INVALID_EXE_SIGNATURE	191	Windows NT mode cannot run %1.
ERROR_EXE_MARKED_INVALID	192	The operating system cannot run %1.
ERROR_BAD_EXE_FORMAT	193	%1 is not a valid Windows NT application.
ERROR_ITERATED_DATA_EXCEEDS_64k	194	The operating system cannot run %1.
ERROR_INVALID_MINALLOCSIZE	195	The operating system cannot run %1.
ERROR_DYNLINK_FROM_INVALID_RING	196	The operating system cannot run this application program.
ERROR_IOPL_NOT_ENABLED	197	The operating system is not presently configured to run this application.
ERROR_INVALID_SEGDPL	198	The operating system cannot run %1.
ERROR_AUTODATASEG_EXCEEDS_64k	199	The operating system cannot run this application program.
ERROR_RING2SEG_MUST_BE_MOVABLE	200	The code segment cannot be greater than or equal to 64KB.
ERROR_RELOC_CHAIN_XEEDS_SEGLIM	201	The operating system cannot run %1.
ERROR_INFLOOP_IN_RELOC_CHAIN	202	The operating system cannot run %1.
ERROR_ENVVAR_NOT_FOUND	203	The system could not find the environment option that was entered.
ERROR_NO_SIGNAL_SENT	205	No process in the command subtree has a signal handler.
ERROR_FILENAME_EXCED_RANGE	206	The filename or extension is too long.
ERROR_RING2_STACK_IN_USE	207	The ring 2 stack is in use.
ERROR_META_EXPANSION_TOO_LONG	208	The global filename characters (such as * and ?) are entered incorrectly or too many global filename characters are specified.
ERROR_INVALID_SIGNAL_NUMBER	209	The signal being posted is not correct.
ERROR_THREAD_1_INACTIVE	210	The signal handler cannot be set.

Constant	*Value*	*Description*
ERROR_LOCKED	212	The segment is locked and cannot be reallocated.
ERROR_TOO_MANY_MODULES	214	Too many dynamic link modules are attached to this program or dynamic link module.
ERROR_NESTING_NOT_ALLOWED	215	Calls can't be nested to `LoadModule`.
ERROR_BAD_PIPE	230	The pipe state is invalid.
ERROR_PIPE_BUSY	231	All pipe instances are busy.
ERROR_NO_DATA	232	The pipe is being closed.
ERROR_PIPE_NOT_CONNECTED	233	No process is on the other end of the pipe.
ERROR_MORE_DATA	234	More data is available.
ERROR_VC_DISCONNECTED	240	The session was cancelled.
ERROR_INVALID_EA_NAME	254	The specified extended attribute name was invalid.
ERROR_EA_LIST_INCONSISTENT	255	The extended attributes are inconsistent.
ERROR_NO_MORE_ITEMS	259	No more data is available.
ERROR_CANNOT_COPY	266	The Copy API cannot be used.
ERROR_DIRECTORY	267	The directory name is invalid.
ERROR_EAS_DIDNT_FIT	275	The extended attributes did not fit in the buffer.
ERROR_EA_FILE_CORRUPT	276	The extended attribute file on the mounted file system is corrupt.
ERROR_EA_TABLE_FULL	277	The extended attribute table file is full.
ERROR_INVALID_EA_HANDLE	278	The specified extended attribute handle is invalid.
ERROR_EAS_NOT_SUPPORTED	282	The mounted file system does not support extended attributes.
ERROR_NOT_OWNER	288	An attempt has been made to release a mutex not owned by the caller.
ERROR_TOO_MANY_POSTS	298	Too many posts were made to a semaphore.
ERROR_PARTIAL_COPY	299	Only part of a `Read/Write ProcessMemory` request was completed.
ERROR_MR_MID_NOT_FOUND	317	The system cannot find a message for message number $%1 in message file for %2.
ERROR_INVALID_ADDRESS	487	Attempt to access invalid address.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_ARITHMETIC_OVERFLOW	534	The arithmetic result exceeded 32 bits.
ERROR_PIPE_CONNECTED	535	A process is on the other end of the pipe.
ERROR_PIPE_LISTENING	536	A pipe is waiting for a process to open the other end of the pipe.
ERROR_EA_ACCESS_DENIED	994	Access to the extended attribute was denied.
ERROR_OPERATION_ABORTED	995	The I/O operation has been aborted because of a thread exit or an application request.
ERROR_IO_INCOMPLETE	996	The overlapped I/O event is not in a signaled state.
ERROR_IO_PENDING	997	The overlapped I/O operation is in progress.
ERROR_NOACCESS	998	Access to the memory location is invalid.
ERROR_SWAPERROR	999	An error has occurred in performing in page operation.
ERROR_STACK_OVERFLOW	1001	Recursion is too deep and the stack overflowed.
ERROR_INVALID_MESSAGE	1002	The window cannot act on the sent message.
ERROR_CAN_NOT_COMPLETE	1003	This function cannot be completed.
ERROR_INVALID_FLAGS	1004	Flags are invalid.
ERROR_UNRECOGNIZED_VOLUME	1005	The volume does not contain a recognized file system. Make sure that all required file system drivers are loaded and that the volume is not corrupt.
ERROR_FILE_INVALID	1006	The volume for a file has been externally altered such that the opened file is no longer valid.
ERROR_FULLSCREEN_MODE	1007	The requested operation cannot be performed in full-screen mode.
ERROR_NO_TOKEN	1008	An attempt was made to reference a token that does not exist.
ERROR_BADDB	1009	The configuration Registry database is corrupt.
ERROR_BADKEY	1010	The configuration Registry key is invalid.
ERROR_CANTOPEN	1011	The configuration Registry key could not be opened.

Constant	*Value*	*Description*
ERROR_CANTREAD	1012	The configuration Registry key could not be read.
ERROR_CANTWRITE	1013	The configuration Registry key could not be written.
ERROR_REGISTRY_RECOVERED	1014	One of the files in the Registry database had to be recovered by use of a log or alternate copy. The recovery was successful.
ERROR_REGISTRY_CORRUPT	1015	The Registry is corrupt. The structure of one of the files that contains Registry data is corrupt, the system's image of the file in memory is corrupt, or the file could not be recovered because the alternate copy or log was absent or corrupt.
ERROR_REGISTRY_IO_FAILED	1016	An I/O operation initiated by the Registry failed and was unable to be recovered. The Registry could not read in, write out, or flush one of the files that contains the system's image of the Registry.
ERROR_NOT_REGISTRY_FILE	1017	The system has attempted to load or restore a file into the Registry, but the specified file is not in a Registry file format.
ERROR_KEY_DELETED	1018	An illegal operation was attempted on a Registry key that has been marked for deletion.
ERROR_NO_LOG_SPACE	1019	The system could not allocate the required space in a Registry log.
ERROR_KEY_HAS_CHILDREN	1020	A symbolic link could not be created in a Registry key that already has subkeys or values.
ERROR_CHILD_MUST_BE_VOLATILE	1021	A stable subkey under a volatile parent key could not be created.
ERROR_NOTIFY_ENUM_DIR	1022	A "notify change" request is being completed, and the information is not being returned in the caller's buffer. The caller now needs to enumerate the files to find the changes.
ERROR_DEPENDENT_SERVICES_RUNNING	1051	A stop control has been sent to a service upon which other running services depend.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_SERVICE_CONTROL	1052	The requested control is not valid for this service.
ERROR_SERVICE_REQUEST_TIMEOUT	1053	The service did not respond to the start or control request in a timely fashion.
ERROR_SERVICE_NO_THREAD	1054	A thread could not be created for the service.
ERROR_SERVICE_DATABASE_LOCKED	1055	The service database is locked.
ERROR_SERVICE_ALREADY_RUNNING	1056	An instance of the service is already running.
ERROR_INVALID_SERVICE_ACCOUNT	1057	The account name is invalid or does not exist.
ERROR_SERVICE_DISABLED	1058	The specified service is disabled and cannot be started.
ERROR_CIRCULAR_DEPENDENCY	1059	A circular service dependency was specified.
ERROR_SERVICE_DOES_NOT_EXIST	1060	The specified service does not exist as an installed service.
ERROR_SERVICE_CANNOT_ACCEPT_CTRL	1061	The service cannot accept control messages at this time.
ERROR_SERVICE_NOT_ACTIVE	1062	The service has not been started.
ERROR_FAILED_SERVICE_CONTROLLER	1063	The service process could not connect to the service controller.
ERROR_EXCEPTION_IN_SERVICE	1064	An exception occurred in the service when handling the control request.
ERROR_DATABASE_DOES_NOT_EXIST	1065	The database specified does not exist.
ERROR_SERVICE_SPECIFIC_ERROR	1066	The service has returned a service-specific error code.
ERROR_PROCESS_ABORTED	1067	The process terminated unexpectedly.
ERROR_SERVICE_DEPENDENCY_FAIL	1068	The dependency service or group failed to start.
ERROR_SERVICE_LOGON_FAILED	1069	The service did not start due to a logon failure.
ERROR_SERVICE_START_HANG	1070	After starting, the service hung up in a start-pending state.
ERROR_INVALID_SERVICE_LOCK	1071	The specified service database lock is invalid.
ERROR_SERVICE_MARKED_FOR_DELETE	1072	The specified service has been marked for deletion.
ERROR_SERVICE_EXISTS	1073	The specified service already exists.

Constant	*Value*	*Description*
ERROR_ALREADY_RUNNING_LKG	1074	The system is currently running with the last-known-good configuration.
ERROR_SERVICE_DEPENDENCY_DELETED	1075	The dependency service does not exist or has been marked for deletion.
ERROR_BOOT_ALREADY_ACCEPTED	1076	The current boot has already been accepted for use as the last-known-good control set.
ERROR_SERVICE_NEVER_STARTED	1077	No attempts to start the service have been made since the last boot.
ERROR_DUPLICATE_SERVICE_NAME	1078	The name is already in use as either a service name or a service display name.
ERROR_END_OF_MEDIA	1100	The physical end of the tape has been reached.
ERROR_FILEMARK_DETECTED	1101	A tape access reached a filemark.
ERROR_BEGINNING_OF_MEDIA	1102	The beginning of the tape or partition was encountered.
ERROR_SETMARK_DETECTED	1103	A tape access reached the end of a set of files.
ERROR_NO_DATA_DETECTED	1104	No more data is on the tape.
ERROR_PARTITION_FAILURE	1105	The tape could not be partitioned.
ERROR_INVALID_BLOCK_LENGTH	1106	During the access of a new tape of a multivolume partition, the current block size is incorrect.
ERROR_DEVICE_NOT_PARTITIONED	1107	The tape partition information could not be found when a tape was loaded.
ERROR_UNABLE_TO_LOCK_MEDIA	1108	The media-eject mechanism could not be locked.
ERROR_UNABLE_TO_UNLOAD_MEDIA	1109	The media could not be loaded.
ERROR_MEDIA_CHANGED	1110	The media in the drive may have changed.
ERROR_BUS_RESET	1111	The I/O bus was reset.
ERROR_NO_MEDIA_IN_DRIVE	1112	No media was in the drive.
ERROR_NO_UNICODE_TRANSLATION	1113	No mapping for the Unicode character exists in the target multibyte code page.
ERROR_DLL_INIT_FAILED	1114	A dynamic link library (DLL) initialization routine failed.
ERROR_SHUTDOWN_IN_PROGRESS	1115	A system shutdown is in progress.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_NO_SHUTDOWN_IN_PROGRESS	1116	The system shutdown could not be aborted because no shutdown was in progress.
ERROR_IO_DEVICE	1117	The request could not be performed because of an I/O device error.
ERROR_SERIAL_NO_DEVICE	1118	No serial device was successfully initialized. The serial driver will unload.
ERROR_IRQ_BUSY	1119	A device could not be opened that was sharing an interrupt request (IRQ) with other devices. At least one other device that uses that IRQ was already opened.
ERROR_MORE_WRITES	1120	A serial I/O operation was completed by another write to the serial port. (IOCTL_SERIAL_XOFF_COUNTER reached zero.)
ERROR_COUNTER_TIMEOUT	1121	A serial I/O operation completed because the timeout period expired. (IOCTL_SERIAL_XOFF_COUNTER did not reach zero.)
ERROR_FLOPPY_ID_MARK_NOT_FOUND	1122	No ID address mark was found on the floppy disk.
ERROR_FLOPPY_WRONG_CYLINDER	1123	A mismatch occurred between the floppy disk sector ID field and the floppy disk controller track address.
ERROR_FLOPPY_UNKNOWN_ERROR	1124	The floppy disk controller reported an error that the floppy disk driver does not recognize.
ERROR_FLOPPY_BAD_REGISTERS	1125	The floppy disk controller returned inconsistent results in its registers.
ERROR_DISK_RECALIBRATE_FAILED	1126	During a hard disk access, a recalibrate operation failed, even after retries.
ERROR_DISK_OPERATION_FAILED	1127	During a hard disk access, a disk operation failed, even after retries.
ERROR_DISK_RESET_FAILED	1128	During a hard disk access, a disk controller reset was needed, but even that failed.
ERROR_EOM_OVERFLOW	1129	The physical end of the tape was encountered.
ERROR_NOT_ENOUGH_SERVER_MEMORY	1130	Not enough server storage is available to process this command.

Constant	*Value*	*Description*
ERROR_POSSIBLE_DEADLOCK	1131	A potential deadlock condition has been detected.
ERROR_MAPPED_ALIGNMENT	1132	The base address or the file offset specified does not have the proper alignment.
ERROR_SET_POWER_STATE_VETOED	1140	An attempt to change the system power state was vetoed by another application or driver.
ERROR_SET_POWER_STATE_FAILED	1141	The system BIOS failed an attempt to change the system power state.
ERROR_OLD_WIN_VERSION	1150	The specified program requires a newer version of Windows.
ERROR_APP_WRONG_OS	1151	The specified program is not a Windows or MS-DOS program.
ERROR_SINGLE_INSTANCE_APP	1152	You cannot start more than one instance of the specified program.
ERROR_RMODE_APP	1153	You cannot start more than one instance of the specified program.
ERROR_INVALID_DLL	1154	One of the library files needed to run this application is damaged.
ERROR_NO_ASSOCIATION	1155	No application is associated with the specified file for this operation.
ERROR_DDE_FAIL	1156	An error occurred in sending the command to the application.
ERROR_DLL_NOT_FOUND	1157	One of the library files needed to run this application cannot be found.
ERROR_BAD_USERNAME	2202	The specified username is invalid.
ERROR_NOT_CONNECTED	2250	This network connection does not exist.
ERROR_OPEN_FILES	2401	This network connection has files open or requests pending.
ERROR_ACTIVE_CONNECTIONS	2402	Active connections still exist.
ERROR_DEVICE_IN_USE	2404	The device is in use by an active process and cannot be disconnected.
ERROR_BAD_DEVICE	1200	The specified device name is invalid.
ERROR_CONNECTION_UNAVAIL	1201	The device is not currently connected, but it is a remembered connection.
ERROR_DEVICE_ALREADY_REMEMBERED	1202	An attempt was made to remember a device that had previously been remembered.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_NO_NET_OR_BAD_PATH	1203	No network provider accepted the given network path.
ERROR_BAD_PROVIDER	1204	The specified network provider name is invalid.
ERROR_CANNOT_OPEN_PROFILE	1205	The network connection profile could not be opened.
ERROR_BAD_PROFILE	1206	The network connection profile is corrupt.
ERROR_NOT_CONTAINER	1207	A noncontainer cannot be enumerated.
ERROR_EXTENDED_ERROR	1208	An extended error has occurred.
ERROR_INVALID_GROUPNAME	1209	The format of the specified group name is invalid.
ERROR_INVALID_COMPUTERNAME	1210	The format of the specified computer name is invalid.
ERROR_INVALID_EVENTNAME	1211	The format of the specified event name is invalid.
ERROR_INVALID_DOMAINNAME	1212	The format of the specified domain name is invalid.
ERROR_INVALID_SERVICENAME	1213	The format of the specified service name is invalid.
ERROR_INVALID_NETNAME	1214	The format of the specified network name is invalid.
ERROR_INVALID_SHARENAME	1215	The format of the specified share name is invalid.
ERROR_INVALID_PASSWORDNAME	1216	The format of the specified password is invalid.
ERROR_INVALID_MESSAGENAME	1217	The format of the specified message name is invalid.
ERROR_INVALID_MESSAGEDEST	1218	The format of the specified message destination is invalid.
ERROR_SESSION_CREDENTIAL_CONFLIC	1219	The credentials supplied conflict with an existing set of credentials.
ERROR_REMOTE_SESSION_LIMIT_EXCEE	1220	An attempt was made to establish a session to a network server, but too many sessions are already established to that server.
ERROR_DUP_DOMAINNAME	1221	Another computer on the network is already using the workgroup or domain name.

Constant	*Value*	*Description*
ERROR_NO_NETWORK	1222	The network is not present or not started.
ERROR_CANCELLED	1223	The user cancelled the operation.
ERROR_USER_MAPPED_FILE	1224	The requested operation cannot be performed on a file with a user-mapped section open.
ERROR_CONNECTION_REFUSED	1225	The remote system refused the network connection.
ERROR_GRACEFUL_DISCONNECT	1226	The network connection was gracefully closed.
ERROR_ADDRESS_ALREADY_ASSOCIATED	1227	The network transport endpoint already has an address associated with it.
ERROR_ADDRESS_NOT_ASSOCIATED	1228	An address has not yet been associated with the network endpoint.
ERROR_CONNECTION_INVALID	1229	An operation was attempted on a nonexistent network connection.
ERROR_CONNECTION_ACTIVE	1230	An invalid operation was attempted on an active network connection.
ERROR_NETWORK_UNREACHABLE	1231	The transport cannot reach the remote network.
ERROR_HOST_UNREACHABLE	1232	The transport cannot reach the remote system.
ERROR_PROTOCOL_UNREACHABLE	1233	The transport protocol cannot reach the remote system.
ERROR_PORT_UNREACHABLE	1234	No service is operating at the destination network endpoint on the remote system.
ERROR_REQUEST_ABORTED	1235	The request was aborted.
ERROR_CONNECTION_ABORTED	1236	The local system aborted the network connection.
ERROR_RETRY	1237	The operation could not be completed. A retry should be performed.
ERROR_CONNECTION_COUNT_LIMIT	1238	A connection to the server could not be made because the limit on the number of concurrent connections for this account has been reached.
ERROR_LOGIN_TIME_RESTRICTION	1239	An attempt was made to log in during an unauthorized time of day for this account.
ERROR_LOGIN_WKSTA_RESTRICTION	1240	The account is not authorized to log in from this station.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INCORRECT_ADDRESS	1241	The network address could not be used for the operation requested.
ERROR_ALREADY_REGISTERED	1242	The service is already registered.
ERROR_SERVICE_NOT_FOUND	1243	The specified service does not exist.
ERROR_NOT_AUTHENTICATED	1244	The operation being requested was not performed because the user has not been authenticated.
ERROR_NOT_LOGGED_ON	1245	The operation being requested was not performed because the user has not logged onto the network. The specified service does not exist.
ERROR_CONTINUE	1246	This is a return that wants the caller to continue with work in progress.
ERROR_ALREADY_INITIALIZED	1247	An attempt was made to perform an initialization operation when initialization has already been completed.
ERROR_NO_MORE_DEVICES	1248	No more local devices exist.
ERROR_NOT_ALL_ASSIGNED	1300	Not all privileges referenced are assigned to the caller.
ERROR_SOME_NOT_MAPPED	1301	Some mapping between account names and security IDs was not done.
ERROR_NO_QUOTAS_FOR_ACCOUNT	1302	No system quota limits are specifically set for this account.
ERROR_LOCAL_USER_SESSION_KEY	1303	No encryption key is available. A well-known encryption key was returned.
ERROR_NULL_LM_PASSWORD	1304	The NT password is too complex to be converted to a LAN Manager password. The LAN Manager password returned is a null string.
ERROR_UNKNOWN_REVISION	1305	The revision level is unknown.
ERROR_REVISION_MISMATCH	1306	The two revision levels are incompatible.
ERROR_INVALID_OWNER	1307	This security ID may not be assigned as the owner of this object.
ERROR_INVALID_PRIMARY_GROUP	1308	This security ID may not be assigned as the primary group of an object.

Constant	*Value*	*Description*
ERROR_NO_IMPERSONATION_TOKEN	1309	A thread that's not currently impersonating a client has made an attempt to operate on an impersonation token.
ERROR_CANT_DISABLE_MANDATORY	1310	The group may not be disabled.
ERROR_NO_LOGON_SERVERS	1311	No logon servers are currently available to service the logon request.
ERROR_NO_SUCH_LOGON_SESSION	1312	A specified logon session does not exist. It may already have been terminated.
ERROR_NO_SUCH_PRIVILEGE	1313	A specified privilege does not exist.
ERROR_PRIVILEGE_NOT_HELD	1314	A required privilege is not held by the client.
ERROR_INVALID_ACCOUNT_NAME	1315	The name provided is not a properly formed account name.
ERROR_USER_EXISTS	1316	The specified user already exists.
ERROR_NO_SUCH_USER	1317	The specified user does not exist.
ERROR_GROUP_EXISTS	1318	The specified group already exists.
ERROR_NO_SUCH_GROUP	1319	The specified group does not exist.
ERROR_MEMBER_IN_GROUP	1320	Either the specified user account is already a member of the specified group or the specified group cannot be deleted because it contains a member.
ERROR_MEMBER_NOT_IN_GROUP	1321	The specified user account is not a member of the specified group account.
ERROR_LAST_ADMIN	1322	The last remaining administration account cannot be disabled or deleted.
ERROR_WRONG_PASSWORD	1323	The password cannot be updated. The value provided as the current password is incorrect.
ERROR_ILL_FORMED_PASSWORD	1324	The password cannot be updated. The value provided for the new password contains values that are not allowed in passwords.
ERROR_PASSWORD_RESTRICTION	1325	The password cannot be updated because a password update rule has been violated.
ERROR_LOGON_FAILURE	1326	A logon failure has occurred: unknown user name or bad password.
ERROR_ACCOUNT_RESTRICTION	1327	A logon failure has occurred: user account restriction.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_LOGON_HOURS	1328	A logon failure has occurred: Account logon time restriction violation.
ERROR_INVALID_WORKSTATION	1329	A logon failure has occurred: The user is not allowed to logon to this computer.
ERROR_PASSWORD_EXPIRED	1330	A logon failure has occurred: The specified account password has expired.
ERROR_ACCOUNT_DISABLED	1331	A logon failure has occurred: The account has been currently disabled.
ERROR_NONE_MAPPED	1332	No mapping between account names and security IDs was done.
ERROR_TOO_MANY_LUIDS_REQUESTED	1333	Too many local user identifiers (LUIDs) were requested at one time.
ERROR_LUIDS_EXHAUSTED	1334	No more local user identifiers (LUIDs) are available.
ERROR_INVALID_SUB_AUTHORITY	1335	The subauthority part of a security ID is invalid for this particular use.
ERROR_INVALID_ACL	1336	The Access Control List (ACL) structure is invalid.
ERROR_INVALID_SID	1337	The security ID structure is invalid.
ERROR_INVALID_SECURITY_DESCR	1338	The security descriptor structure is invalid.
ERROR_BAD_INHERITANCE_ACL	1340	The inherited Access Control List (ACL) or Access Control Entry (ACE) could not be built.
ERROR_SERVER_DISABLED	1341	The server is currently disabled.
ERROR_SERVER_NOT_DISABLED	1342	The server is currently enabled.
ERROR_INVALID_ID_AUTHORITY	1343	The value provided was an invalid value for an identifier authority.
ERROR_ALLOTTED_SPACE_EXCEEDED	1344	No more memory is available for security information updates.
ERROR_INVALID_GROUP_ATTRIBUTES	1345	The specified attributes are invalid or incompatible with the attributes for the group as a whole.
ERROR_BAD_IMPERSONATION_LEVEL	1346	Either a required impersonation level was not provided or the provided impersonation level is invalid.

Constant	*Value*	*Description*
ERROR_CANT_OPEN_ANONYMOUS	1347	An anonymous-level security token cannot be opened.
ERROR_BAD_VALIDATION_CLASS	1348	The validation information class requested was invalid.
ERROR_BAD_TOKEN_TYPE	1349	The type of the token is inappropriate for its attempted use.
ERROR_NO_SECURITY_ON_OBJECT	1350	A security operation could not be performed on an object that has no associated security.
ERROR_CANT_ACCESS_DOMAIN_INFO	1351	A Windows NT server could not be contacted or objects within the domain are protected such that necessary information could not be retrieved.
ERROR_INVALID_SERVER_STATE	1352	The Security Account Manager (SAM) or Local Security Authority (LSA) server was in the wrong state to perform the security operation.
ERROR_INVALID_DOMAIN_STATE	1353	The domain was in the wrong state to perform the security operation.
ERROR_INVALID_DOMAIN_ROLE	1354	This operation is only allowed for the Primary Domain Controller of the domain.
ERROR_NO_SUCH_DOMAIN	1355	The specified domain did not exist.
ERROR_DOMAIN_EXISTS	1356	The specified domain already exists.
ERROR_DOMAIN_LIMIT_EXCEEDED	1357	An attempt was made to exceed the limit on the number of domains per server.
ERROR_INTERNAL_DB_CORRUPTION	1358	The requested operation could not be completed because of a catastrophic media failure or a data structure corruption on the disk.
ERROR_INTERNAL_ERROR	1359	The security account database contains an internal inconsistency.
ERROR_GENERIC_NOT_MAPPED	1360	Generic access types were contained in an access mask that should already be mapped to nongeneric types.
ERROR_BAD_DESCRIPTOR_FORMAT	1361	A security descriptor is not in the correct format (absolute or self-relative).
ERROR_NOT_LOGON_PROCESS	1362	The requested action is restricted for use by logon processes only. The calling process has not been registered as a logon process.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_LOGON_SESSION_EXISTS	1363	You cannot start a new logon session with an ID that's already in use.
ERROR_NO_SUCH_PACKAGE	1364	A specified authentication package is unknown.
ERROR_BAD_LOGON_SESSION_STATE	1365	The logon session is not in a state that's consistent with the requested operation.
ERROR_LOGON_SESSION_COLLISION	1366	The logon session ID is already in use.
ERROR_INVALID_LOGON_TYPE	1367	A logon request contained an invalid logon type value.
ERROR_CANNOT_IMPERSONATE	1368	You cannot impersonate via a named pipe until data has been read from that pipe.
ERROR_RXACT_INVALID_STATE	1369	The transaction state of a Registry subtree is incompatible with the requested operation.
ERROR_RXACT_COMMIT_FAILURE	1370	An internal security database corruption has been encountered.
ERROR_SPECIAL_ACCOUNT	1371	You cannot perform this operation on built-in accounts.
ERROR_SPECIAL_GROUP	1372	You cannot perform this operation on this built-in special group.
ERROR_SPECIAL_USER	1373	You cannot perform this operation on this built-in special user.
ERROR_MEMBERS_PRIMARY_GROUP	1374	The user cannot be removed from a group because the group is currently the user's primary group.
ERROR_TOKEN_ALREADY_IN_USE	1375	The token is already in use as a primary token.
ERROR_NO_SUCH_ALIAS	1376	The specified local group does not exist.
ERROR_MEMBER_NOT_IN_ALIAS	1377	The specified account name is not a member of the local group.
ERROR_MEMBER_IN_ALIAS	1378	The specified account name is already a member of the local group.
ERROR_ALIAS_EXISTS	1379	The specified local group already exists.
ERROR_LOGON_NOT_GRANTED	1380	A logon failure has occurred: The user has not been granted the requested logon type at this computer.

Constant	*Value*	*Description*
ERROR_TOO_MANY_SECRETS	1381	The maximum number of secrets that may be stored in a single system has been exceeded.
ERROR_SECRET_TOO_LONG	1382	The length of a secret exceeds the maximum length allowed.
ERROR_INTERNAL_DB_ERROR	1383	The Local Security Authority database contains an internal inconsistency.
ERROR_TOO_MANY_CONTEXT_IDS	1384	During a logon attempt, the user's security context accumulated too many security IDs.
ERROR_LOGON_TYPE_NOT_GRANTED	1385	A logon failure has occurred: The user has not been granted the requested logon type at this computer.
ERROR_NT_CROSS_ENCRYPTION_REQUIR	1386	A cross-encrypted password is necessary to change a user password.
ERROR_NO_SUCH_MEMBER	1387	A new member could not be added to a local group because the member does not exist.
ERROR_INVALID_MEMBER	1388	A new member could not be added to a local group because the member has the wrong account type.
ERROR_TOO_MANY_SIDS	1389	Too many security IDs have been specified.
ERROR_LM_CROSS_ENCRYPTION_REQUIR	1390	A cross-encrypted password is necessary to change this user password.
ERROR_NO_INHERITANCE	1391	Indicates a TACL contains no inheritable components.
ERROR_FILE_CORRUPT	1392	The file or directory is corrupt and unable to be read.
ERROR_DISK_CORRUPT	1393	The disk structure is corrupt and unable to be read.
ERROR_NO_USER_SESSION_KEY	1394	No user session key exists for the specified logon session.
ERROR_LICENSE_QUOTA_EXCEEDED	1395	The service being accessed is licensed for a particular number of connections. No more connections can be made to the service at this time because as many connections as the service can accept already exist.
ERROR_INVALID_WINDOW_HANDLE	1400	The window handle is invalid.
ERROR_INVALID_MENU_HANDLE	1401	The menu handle is invalid.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_CURSOR_HANDLE	1402	The cursor handle is invalid.
ERROR_INVALID_ACCEL_HANDLE	1403	The accelerator table handle is invalid.
ERROR_INVALID_HOOK_HANDLE	1404	The hook handle is invalid.
ERROR_INVALID_DWP_HANDLE	1405	The handle to a multiple-window position structure is invalid.
ERROR_TLW_WITH_WSCHILD	1406	A top-level child window cannot be created.
ERROR_CANNOT_FIND_WND_CLASS	1407	A window class cannot be found.
ERROR_WINDOW_OF_OTHER_THREAD	1408	The window is invalid; it belongs to the other thread.
ERROR_HOTKEY_ALREADY_REGISTERED	1409	The hotkey is already registered.
ERROR_CLASS_ALREADY_EXISTS	1410	The class already exists.
ERROR_CLASS_DOES_NOT_EXIST	1411	The class does not exist.
ERROR_CLASS_HAS_WINDOWS	1412	The class still has open windows.
ERROR_INVALID_INDEX	1413	The index is invalid.
ERROR_INVALID_ICON_HANDLE	1414	The icon handle is invalid.
ERROR_PRIVATE_DIALOG_INDEX	1415	You're using private dialog window words.
ERROR_LISTBOX_ID_NOT_FOUND	1416	The list box identifier was not found.
ERROR_NO_WILDCARD_CHARACTERS	1417	No wildcards were found.
ERROR_CLIPBOARD_NOT_OPEN	1418	The thread does not have a Clipboard open.
ERROR_HOTKEY_NOT_REGISTERED	1419	The hotkey is not registered.
ERROR_WINDOW_NOT_DIALOG	1420	The window is not a valid dialog window.
ERROR_CONTROL_ID_NOT_FOUND	1421	The control ID was not found.
ERROR_INVALID_COMBOBOX_MESSAGE	1422	The message for a combo box was invalid because it does not have an edit control.
ERROR_WINDOW_NOT_COMBOBOX	1423	The window is not a combo box.
ERROR_INVALID_EDIT_HEIGHT	1424	The height must be less than 256 pixels.
ERROR_DC_NOT_FOUND	1425	The device context (DC) handle is invalid.
ERROR_INVALID_HOOK_FILTER	1426	The hook procedure type is invalid.
ERROR_INVALID_FILTER_PROC	1427	The hook procedure is invalid.
ERROR_HOOK_NEEDS_HMOD	1428	You cannot set a nonlocal hook without a module handle.
ERROR_GLOBAL_ONLY_HOOK	1429	This hook procedure can only be set globally.

Constant	*Value*	*Description*
ERROR_JOURNAL_HOOK_SET	1430	The journal hook procedure is already installed.
ERROR_HOOK_NOT_INSTALLED	1431	The hook procedure is not installed.
ERROR_INVALID_LB_MESSAGE	1432	The message for a single-selection list box is invalid.
ERROR_SETCOUNT_ON_BAD_LB	1433	LB_SETCOUNT was sent to a nonlazy list box.
ERROR_LB_WITHOUT_TABSTOPS	1434	This list box does not support tab stops.
ERROR_DESTROY_OBJECT_OF_OTHER_TH	1435	You cannot destroy an object created by another thread.
ERROR_CHILD_WINDOW_MENU	1436	Child windows cannot have menus.
ERROR_NO_SYSTEM_MENU	1437	The window does not have a system menu.
ERROR_INVALID_MSGBOX_STYLE	1438	The message box style is invalid.
ERROR_INVALID_SPI_VALUE	1439	The systemwide (SPI_*) parameter is invalid.
ERROR_SCREEN_ALREADY_LOCKED	1440	The screen is already locked.
ERROR_HWNDS_HAVE_DIFF_PARENT	1441	All handles to windows in a multiple-window position structure must have the same parent.
ERROR_NOT_CHILD_WINDOW	1442	The window is not a child window.
ERROR_INVALID_GW_COMMAND	1443	The GW_* command is invalid.
ERROR_INVALID_THREAD_ID	1444	The thread identifier is invalid.
ERROR_NON_MDICHILD_WINDOW	1445	A message cannot be processed from a window that's not a multiple-document interface (MDI) window.
ERROR_POPUP_ALREADY_ACTIVE	1446	The pop-up menu is already active.
ERROR_NO_SCROLLBARS	1447	The window does not have scrollbars.
ERROR_INVALID_SCROLLBAR_RANGE	1448	The scrollbar range cannot be greater than $7FFF.
ERROR_INVALID_SHOWWIN_COMMAND	1449	You cannot show or remove the window in the way specified.
ERROR_EVENTLOG_FILE_CORRUPT	1500	The event log file is corrupt.
ERROR_EVENTLOG_CANT_START	1501	No event log file could be opened, so the event-logging service did not start.
ERROR_LOG_FILE_FULL	1502	The event log file is full.
ERROR_EVENTLOG_FILE_CHANGED	1503	The event log file has changed between reads.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_INVALID_USER_BUFFER	1784	The supplied user buffer is not valid for the requested operation.
ERROR_UNRECOGNIZED_MEDIA	1785	The disk media is not recognized. It might not be formatted.
ERROR_NO_TRUST_LSA_SECRET	1786	The workstation does not have a trust secret.
ERROR_NO_TRUST_SAM_ACCOUNT	1787	The SAM database on the Windows NT server does not have a computer account for this workstation trust relationship.
ERROR_TRUSTED_DOMAIN_FAILURE	1788	The trust relationship between the primary domain and the trusted domain failed.
ERROR_TRUSTED_RELATIONSHIP_FAILU	1789	The trust relationship between this workstation and the primary domain failed.
ERROR_TRUST_FAILURE	1790	The network logon failed.
ERROR_NETLOGON_NOT_STARTED	1792	A remote procedure call is already in progress for this thread. An attempt was made to log on, but the network logon service was not started.
ERROR_ACCOUNT_EXPIRED	1793	The user's account has expired.
ERROR_REDIRECTOR_HAS_OPEN_HANDLE	1794	The redirector is in use and cannot be unloaded.
ERROR_PRINTER_DRIVER_ALREADY_INS	1795	The specified printer driver is already installed.
ERROR_UNKNOWN_PORT	1796	The specified port is unknown.
ERROR_UNKNOWN_PRINTER_DRIVER	1797	The printer driver is unknown.
ERROR_UNKNOWN_PRINTPROCESSOR	1798	The print processor is unknown.
ERROR_INVALID_SEPARATOR_FILE	1799	The specified separator file is invalid.
ERROR_INVALID_PRIORITY	1800	The specified priority is invalid.
ERROR_INVALID_PRINTER_NAME	1801	The printer name is invalid.
ERROR_PRINTER_ALREADY_EXISTS	1802	The printer already exists.
ERROR_INVALID_PRINTER_COMMAND	1803	The printer command is invalid.
ERROR_INVALID_DATATYPE	1804	The specified data type is invalid.
ERROR_INVALID_ENVIRONMENT	1805	The environment specified is invalid.
ERROR_NOLOGON_INTERDOMAIN_TRUST	1807	No more bindings exist. The account used is an interdomain trust account. Use your global user account or local user account to access this server.

Constant	*Value*	*Description*
ERROR_NOLOGON_WORKSTATION_TRUST	1808	The account used is a computer account. Use your global user account or local user account to access this server.
ERROR_NOLOGON_SERVER_TRUST_ACCOU	1809	The account used is an server trust account. Use your global user account or local user account to access this server.
ERROR_DOMAIN_TRUST_INCONSISTENT	1810	The name or security ID (SID) of the domain specified is inconsistent with the trust information for that domain.
ERROR_SERVER_HAS_OPEN_HANDLES	1811	The server is in use and cannot be unloaded.
ERROR_RESOURCE_DATA_NOT_FOUND	1812	The specified image file did not contain a resource section.
ERROR_RESOURCE_TYPE_NOT_FOUND	1813	The specified resource type cannot be found in the image file.
ERROR_RESOURCE_NAME_NOT_FOUND	1814	The specified resource name cannot be found in the image file.
ERROR_RESOURCE_LANG_NOT_FOUND	1815	The specified resource language ID cannot be found in the image file.
ERROR_NOT_ENOUGH_QUOTA	1816	Not enough quota is available to process this command.
ERROR_INVALID_TIME	1901	The specified time is invalid.
ERROR_INVALID_FORM_NAME	1902	The specified form name is invalid.
ERROR_INVALID_FORM_SIZE	1903	The specified form size is invalid.
ERROR_ALREADY_WAITING	1904	The specified printer handle is already being waited on.
ERROR_PRINTER_DELETED	1905	The specified printer has been deleted.
ERROR_INVALID_PRINTER_STATE	1906	The state of the printer is invalid.
ERROR_PASSWORD_MUST_CHANGE	1907	The user must change his or her password before logging on for the first time.
ERROR_DOMAIN_CONTROLLER_NOT_FOUN	1908	The domain controller for this domain could not be found.
ERROR_ACCOUNT_LOCKED_OUT	1909	The referenced account is currently locked out and may not be logged on to.
ERROR_NO_BROWSER_SERVERS_FOUND	6118	The list of servers for this workgroup is not currently available.
ERROR_INVALID_PIXEL_FORMAT	2000	The pixel format is invalid.

continues

TABLE A.1 Continued

Constant	*Value*	*Description*
ERROR_BAD_DRIVER	2001	The specified driver is invalid.
ERROR_INVALID_WINDOW_STYLE	2002	The window style or class attribute is invalid for this operation.
ERROR_METAFILE_NOT_SUPPORTED	2003	The requested metafile operation is not supported.
ERROR_TRANSFORM_NOT_SUPPORTED	2004	The requested transformation operation is not supported.
ERROR_CLIPPING_NOT_SUPPORTED	2005	The requested clipping operation is not supported.
ERROR_UNKNOWN_PRINT_MONITOR	3000	The specified print monitor is unknown.
ERROR_PRINTER_DRIVER_IN_USE	3001	The specified printer driver is currently in use.
ERROR_SPOOL_FILE_NOT_FOUND	3002	The spool file was not found.
ERROR_SPL_NO_STARTDOC	3003	A `StartDocPrinter` call was not issued.
ERROR_SPL_NO_ADDJOB	3004	An `AddJob` call was not issued.
ERROR_PRINT_PROCESSOR_ALREADY_IN	3005	The specified print processor has already been installed.
ERROR_PRINT_MONITOR_ALREADY_INST	3006	The specified print monitor has already been installed.
ERROR_WINS_INTERNAL	4000	WINS encountered an error while processing the command.
ERROR_CAN_NOT_DEL_LOCAL_WINS	4001	The local WINS cannot be deleted.
ERROR_STATIC_INIT	4002	The importation from the file failed.
ERROR_INC_BACKUP	4003	The backup failed. Was a full backup done before?
ERROR_FULL_BACKUP	4004	The backup failed. Check the directory to which you are backing up the database.
ERROR_REC_NON_EXISTENT	4005	The name does not exist in the WINS database.
ERROR_RPL_NOT_ALLOWED	4006	Replication with an unconfigured partner is not allowed.

BDE Error Codes

APPENDIX

B

When working with the Borland Database Engine, occasionally you'll receive an error dialog box indicating that some error has occurred in the engine. Most commonly, this happens when customers or clients install your software on their machines and they have some configuration problems with their machines that you're trying to track down for them. Typically, the aforementioned error dialog box provides you with a hexadecimal error code as the description of the error. The question is how to turn that number into a meaningful error message. In order to help you with this task, we've provided the following table. Table B.1 lists all the possible BDE error codes as well as the BDE error strings associated with these error codes.

TABLE B.1 BDE Error Codes

Error Code		
Decimal	*Hex*	*Error String*
0	0000	Successful completion.
33	0021	System error.
34	0022	Object of interest not found.
35	0023	Physical data corruption.
36	0024	I/O-related error.
37	0025	Resource or limit error.
38	0026	Data integrity violation.
39	0027	Invalid request.
40	0028	Lock violation.
41	0029	Access/security violation.
42	002A	Invalid context.
43	002B	OS error.
44	002C	Network error.
45	002D	Optional parameter.
46	002E	Query processor.
47	002F	Version mismatch.
48	0030	Capability not supported.
49	0031	System configuration error.
50	0032	Warning.
51	0033	Miscellaneous.
52	0034	Compatibility error.
62	003E	Driver-specific error.
63	003F	Internal symbol.
256	0100	`KEYVIOL`.

Error Code		
Decimal	*Hex*	*Error String*
257	0101	PROBLEMS.
258	0102	CHANGED.
512	0200	Production index file missing, corrupt, or cannot interpret index key.
513	0201	Open read-only.
514	0202	Open the table in read-only mode.
515	0203	Open and detach.
516	0204	Open the table and detach the production index file.
517	0205	Fail open.
518	0206	Do not open the table.
519	0207	Convert non-dBASE index.
520	0208	Convert production index to dBASE format.
521	0209	BLOB file not found.
522	020A	Open without BLOB file.
523	020B	Open the table without the BLOB file.
524	020C	Empty all BLOB fields.
525	020D	Reinitialize BLOB file and lose all BLOBs.
526	020E	Fail open.
527	020F	Do not open the table.
528	0210	Import non-dBASE BLOB file.
529	0211	Import BLOB file to dBASE format.
530	0212	Open as non-dBASE table.
531	0213	Open table and BLOB file in its native format.
532	0214	Production index language driver mismatch.
533	0215	Production index damaged.
534	0216	Rebuild production index.
535	0217	Rebuild all the production indexes.
1024	0400	Lookup table not found or corrupt.
1025	0401	BLOB file not found or corrupt.
1026	0402	Open read-only.
1027	0403	Open the table in read-only mode.
1028	0404	Fail open.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
1029	0405	Do not open the table.
1030	0406	Remove lookup.
1031	0407	Remove link to lookup table.
1280	0500	Dictionary object exists.
1281	0501	Skip this object.
1282	0502	Skip importing this object and its associated relationships.
1283	0503	Use existing object.
1284	0504	Use existing dictionary object for relationships.
1285	0505	Abort.
1286	0506	Abort the operation.
1287	0507	Dictionary object import failed.
4608	1200	SQL Unknown.
4609	1201	SQL Prepare.
4610	1202	SQL Execute.
4611	1203	SQL Error.
4612	1204	SQL STMT.
4613	1205	SQL Connect.
4614	1206	SQL Transact.
4615	1207	SQL BLOB I/O.
4616	1208	SQL Misc.
4617	1209	SQL Vendor.
4618	120A	ORACLE - `orlon`.
4619	120B	ORACLE - `olon`.
4620	120C	ORACLE - `ologof`.
4621	120D	ORACLE - `ocon`.
4622	120E	ORACLE - `ocof`.
4623	120F	ORACLE - `oopen`.
4624	1210	ORACLE - `osql3`.
4625	1211	ORACLE - `odsc`.
4626	1212	ORACLE - `odefin`.
4627	1213	ORACLE - `obndrv`.

Error Code		
Decimal	*Hex*	*Error String*
4628	1214	ORACLE - obndrvn.
4629	1215	ORACLE - oexec.
4630	1216	ORACLE - ofetch.
4631	1217	ORACLE - ofen.
4632	1218	ORACLE - ocan.
4633	1219	ORACLE - oclose.
4634	121A	ORACLE - oerhms.
4635	121B	ORACLE - oparse.
4636	121C	ORACLE - oflng.
4637	121D	ORACLE - odessp.
4638	121E	ORACLE - odescr.
4639	121F	ORACLE - oexn.
4648	1228	INTRBASE - isc_attach_database.
4649	1229	INTRBASE - isc_blob_default_desc.
4650	122A	INTRBASE - isc_blob_gen_bpb.
4651	122B	INTRBASE - isc_blob_info.
4652	122C	INTRBASE - isc_blob_lookup_desc.
4653	122D	INTRBASE - isc_close_blob.
4654	122E	INTRBASE - isc_commit_retaining.
4655	122F	INTRBASE - isc_commit_transaction.
4656	1230	INTRBASE - isc_create_blob.
4657	1231	INTRBASE - isc_create_blob2.
4658	1232	INTRBASE - isc_decode_date.
4659	1233	INTRBASE - isc_detach_database.
4660	1234	INTRBASE - isc_dsql_allocate_statement.
4661	1235	INTRBASE - isc_dsql_execute.
4662	1236	INTRBASE - isc_dsql_execute2.
4663	1237	INTRBASE - isc_dsql_fetch.
4664	1238	INTRBASE - isc_dsql_free_statement.
4665	1239	INTRBASE - isc_dsql_prepare.
4666	123A	INTRBASE - isc_dsql_set_cursor_name.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
4667	123B	INTRBASE - `isc_dsql_sql_info`.
4668	123C	INTRBASE - `isc_encode_date`.
4669	123D	INTRBASE - `isc_get_segment`.
4670	123E	INTRBASE - `isc_interprete`.
4671	123F	INTRBASE - `isc_open_blob`.
4672	1240	INTRBASE - `isc_open_blob2`.
4673	1241	INTRBASE - `isc_put_segment`.
4674	1242	INTRBASE - `isc_rollback_transaction`.
4675	1243	INTRBASE - `isc_sqlcode`.
4676	1244	INTRBASE - `isc_start_transaction`.
4677	1245	INTRBASE - `isc_vax_integer`.
4688	1250	MSSQL - `dbbind`.
4689	1251	MSSQL - `dbcmd`.
4690	1252	MSSQL - `dbcancel`.
4691	1253	MSSQL - `dbclose`.
4692	1254	MSSQL - `dbcollen`.
4693	1255	MSSQL - `dbcolname`.
4694	1256	MSSQL - `dbcoltype`.
4695	1257	MSSQL - `dbconvert`.
4696	1258	MSSQL - `dbdataready`.
4697	1259	MSSQL - `dbdatlen`.
4698	125A	MSSQL - `dberrhandle`.
4699	125B	MSSQL - `dbfreebuf`.
4700	125C	MSSQL - `dbfreelogin`.
4701	125D	MSSQL - `dbhasretstat`.
4702	125E	MSSQL - `dbinit`.
4703	125F	MSSQL - `dblogin`.
4704	1260	MSSQL - `dbmoretext`.
4705	1261	MSSQL - `dbmsghandle`.
4706	1262	MSSQL - `dbnextrow`.
4707	1263	MSSQL - `dbnumcols`.

Error Code		
Decimal	*Hex*	*Error String*
4708	1264	MSSQL - dbnumrets.
4709	1265	MSSQL - dbopen.
4710	1266	MSSQL - dbresults.
4711	1267	MSSQL - dbretdata.
4712	1268	MSSQL - dbretlen.
4713	1269	MSSQL - dbretstatus.
4714	126A	MSSQL - dbrpcinit.
4715	126B	MSSQL - dbrpcparam.
4716	126C	MSSQL - dbrpcsend.
4717	126D	MSSQL - dbsetlogintime.
4718	126E	MSSQL - dbsetmaxprocs.
4719	126F	MSSQL - dbsetopt.
4720	1270	MSSQL - dbsettime.
4721	1271	MSSQL - dbsqlexec.
4722	1272	MSSQL - dbsqlok.
4723	1273	MSSQL - dbsqlsend.
4724	1274	MSSQL - dbtxptr.
4725	1275	MSSQL - dbtxtimestamp.
4726	1276	MSSQL - dbtxtsnewval.
4727	1277	MSSQL - dbuse.
4728	1278	MSSQL - dbwinexit.
4729	1279	MSSQL - dbwritetext.
4738	1282	ODBC - SQLAllocConnect.
4739	1283	ODBC - SQLAllocEnv.
4740	1284	ODBC - SQLAllocStmt.
4741	1285	ODBC - SQLBindCol.
4742	1286	ODBC - SQLBindParameter.
4743	1287	ODBC - SQLCancel.
4744	1288	ODBC - SQLColumns.
4745	1289	ODBC - SQLConnect.
4746	128A	ODBC - SQLDataSources.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
4747	128B	ODBC - `SQLDescribeCol`.
4748	128C	ODBC - `SQLDisconnect`.
4750	128E	ODBC - `SQLError`.
4751	128F	ODBC - `SQLExecDirect`.
4752	1290	ODBC - `SQLExtendedFetch`.
4753	1291	ODBC - `SQLFetch`.
4754	1292	ODBC - `SQLFreeConnect`.
4755	1293	ODBC - `SQLFreeEnv`.
4756	1294	ODBC - `SQLFreeStmt`.
4757	1295	ODBC - `SQLGetConnectOption`.
4758	1296	ODBC - `SQLGetCursorName`.
4760	1298	ODBC - `SQLGetFunctions`.
4761	1299	ODBC - `SQLGetInfo`.
4762	129A	ODBC - `SQLGetTypeInfo`.
4763	129B	ODBC - `SQLNumResultCols`.
4764	129C	ODBC - `SQLProcedures`.
4765	129D	ODBC - `SQLProcedureColumns`.
4766	129E	ODBC - `SQLRowCount`.
4767	129F	ODBC - `SQLSetConnectOption`.
4768	12A0	ODBC - `SQLSetCursorName`.
4769	12A1	ODBC - `SQLSetParam`.
4770	12A2	ODBC - `SQLSetStmtOption`.
4771	12A3	ODBC - `SQLStatistics`.
4772	12A4	ODBC - `SQLTables`.
4773	12A5	ODBC - `SQLTransact`.
4788	12B4	SYBASE - `dbbind`.
4789	12B5	SYBASE - `dbcmd`.
4790	12B6	SYBASE - `dbcancel`.
4791	12B7	SYBASE - `dbclose`.
4792	12B8	SYBASE - `dbcollen`.
4793	12B9	SYBASE - `dbcolname`.

Error Code		
Decimal	*Hex*	*Error String*
4794	12BA	SYBASE - dbcoltype.
4795	12BB	SYBASE - dbconvert.
4796	12BC	SYBASE - dbpoll.
4797	12BD	SYBASE - dbdatlen.
4798	12BE	SYBASE - dberrhandle.
4799	12BF	SYBASE - dbfreebuf.
4800	12C0	SYBASE - dbloginfree.
4801	12C1	SYBASE - dbhasretstat.
4802	12C2	SYBASE - dbinit.
4803	12C3	SYBASE - dblogin.
4804	12C4	SYBASE - dbmoretext.
4805	12C5	SYBASE - dbmsghandle.
4806	12C6	SYBASE - dbnextrow.
4807	12C7	SYBASE - dbnumcols.
4808	12C8	SYBASE - dbnumrets.
4809	12C9	SYBASE - dbopen.
4810	12CA	SYBASE - dbresults.
4811	12CB	SYBASE - dbretdata.
4812	12CC	SYBASE - dbretlen.
4813	12CD	SYBASE - dbretstatus.
4814	12CE	SYBASE - dbrpcinit.
4815	12CF	SYBASE - dbrpcparam.
4816	12D0	SYBASE - dbrpcsend.
4817	12D1	SYBASE - dbsetlogintime.
4818	12D2	SYBASE - dbsetmaxprocs.
4819	12D3	SYBASE - dbsetopt.
4820	12D4	SYBASE - dbsettime.
4821	12D5	SYBASE - dbsqlexec.
4822	12D6	SYBASE - dbsqlok.
4823	12D7	SYBASE - dbsqlsend.
4824	12D8	SYBASE - dbtxptr.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
4825	12D9	SYBASE - `dbtxtimestamp`.
4826	12DA	SYBASE - `dbtxtsnewval`.
4827	12DB	SYBASE - `dbuse`.
4828	12DC	SYBASE - `dbwinexit`.
4829	12DD	SYBASE - `dbwritetext`.
4830	12DE	SYBASE - `dbcount`.
4831	12DF	SYBASE - `dbdead`.
4942	134E	Unmapped SQL error code.
8449	2101	Cannot open a system file.
8450	2102	I/O error on a system file.
8451	2103	Data structure corruption.
8452	2104	Cannot find engine configuration file.
8453	2105	Cannot write to engine configuration file.
8454	2106	Cannot initialize with different configuration file.
8455	2107	System has been illegally reentered.
8456	2108	Cannot locate `IDAPI32.DLL`.
8457	2109	Cannot load `IDAPI32.DLL`.
8458	210A	Cannot load an IDAPI service library.
8459	210B	Cannot create or open temporary file.
8705	2201	At beginning of table.
8706	2202	At end of table.
8707	2203	Record moved because key value changed.
8708	2204	Record/key deleted.
8709	2205	No current record.
8710	2206	Could not find record.
8711	2207	End of BLOB.
8712	2208	Could not find object.
8713	2209	Could not find family member.
8714	220A	BLOB file is missing.
8715	220B	Could not find language driver.
8961	2301	Corrupt table/index header.

Error Code		
Decimal	*Hex*	*Error String*
8962	2302	Corrupt file - other than header.
8963	2303	Corrupt memo/BLOB file.
8965	2305	Corrupt index.
8966	2306	Corrupt lock file.
8967	2307	Corrupt family file.
8968	2308	Corrupt or missing VAL file.
8969	2309	Foreign index file format.
9217	2401	Read failure.
9218	2402	Write failure.
9219	2403	Cannot access directory.
9220	2404	File Delete operation failed.
9221	2405	Cannot access file.
9222	2406	Access to table disabled because of previous error.
9473	2501	Insufficient memory for this operation.
9474	2502	Not enough file handles.
9475	2503	Insufficient disk space.
9476	2504	Temporary table resource limit.
9477	2505	Record size is too big for table.
9478	2506	Too many open cursors.
9479	2507	Table is full.
9480	2508	Too many sessions from this workstation.
9481	2509	Serial number limit (Paradox).
9482	250A	Some internal limit (see context).
9483	250B	Too many open tables.
9484	250C	Too many cursors per table.
9485	250D	Too many record locks on table.
9486	250E	Too many clients.
9487	250F	Too many indexes on table.
9488	2510	Too many sessions.
9489	2511	Too many open databases.
9490	2512	Too many passwords.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
9491	2513	Too many active drivers.
9492	2514	Too many fields in Table Create.
9493	2515	Too many table locks.
9494	2516	Too many open BLOBs.
9495	2517	Lock file has grown too large.
9496	2518	Too many open queries.
9498	251A	Too many BLOBs.
9499	251B	Filename is too long for a Paradox version 5.0 table.
9500	251C	Row fetch limit exceeded.
9501	251D	Long name not allowed for this table level.
9729	2601	Key violation.
9730	2602	Minimum validity check failed.
9731	2603	Maximum validity check failed.
9732	2604	Field value required.
9733	2605	Master record missing.
9734	2606	Master has detail records. Cannot delete or modify.
9735	2607	Master table level is incorrect.
9736	2608	Field value out of lookup table range.
9737	2609	Lookup Table Open operation failed.
9738	260A	Detail Table Open operation failed.
9739	260B	Master Table Open operation failed.
9740	260C	Field is blank.
9741	260D	Link to master table already defined.
9742	260E	Master table is open.
9743	260F	Detail tables exist.
9744	2610	Master has detail records. Cannot empty it.
9745	2611	Self-referencing referential integrity must be entered one at a time with no other changes to the table.
9746	2612	Detail table is open.
9747	2613	Cannot make this master a detail of another table if its details are not empty.
9748	2614	Referential integrity fields must be indexed.

Error Code		
Decimal	*Hex*	*Error String*
9749	2615	A table linked by referential integrity requires password to open.
9750	2616	Field(s) linked to more than one master.
9985	2701	Number is out of range.
9986	2702	Invalid parameter.
9987	2703	Invalid filename.
9988	2704	File does not exist.
9989	2705	Invalid option.
9990	2706	Invalid handle to the function.
9991	2707	Unknown table type.
9992	2708	Cannot open file.
9993	2709	Cannot redefine primary key.
9994	270A	Cannot change this `RINTDesc`.
9995	270B	Foreign and primary key do not match.
9996	270C	Invalid modify request.
9997	270D	Index does not exist.
9998	270E	Invalid offset into the BLOB.
9999	270F	Invalid descriptor number.
10000	2710	Invalid field type.
10001	2711	Invalid field descriptor.
10002	2712	Invalid field transformation.
10003	2713	Invalid record structure.
10004	2714	Invalid descriptor.
10005	2715	Invalid array of index descriptors.
10006	2716	Invalid array of validity check descriptors.
10007	2717	Invalid array of referential integrity descriptors.
10008	2718	Invalid ordering of tables during restructure.
10009	2719	Name not unique in this context.
10010	271A	Index name required.
10011	271B	Invalid session handle.
10012	271C	Invalid restructure operation.
10013	271D	Driver not known to system.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
10014	271E	Unknown database.
10015	271F	Invalid password given.
10016	2720	No callback function.
10017	2721	Invalid callback buffer length.
10018	2722	Invalid directory.
10019	2723	Translate Error. Value out of bounds.
10020	2724	Cannot set cursor of one table to another.
10021	2725	Bookmarks do not match table.
10022	2726	Invalid index/tag name.
10023	2727	Invalid index descriptor.
10024	2728	Table does not exist.
10025	2729	Table has too many users.
10026	272A	Cannot evaluate key or key does not pass filter condition.
10027	272B	Index already exists.
10028	272C	Index is open.
10029	272D	Invalid BLOB length.
10030	272E	Invalid BLOB handle in record buffer.
10031	272F	Table is open.
10032	2730	Need to do (hard) restructure.
10033	2731	Invalid mode.
10034	2732	Cannot close index.
10035	2733	Index is being used to order table.
10036	2734	Unknown username or password.
10037	2735	Multilevel cascade is not supported.
10038	2736	Invalid field name.
10039	2737	Invalid table name.
10040	2738	Invalid linked cursor expression.
10041	2739	Name is reserved.
10042	273A	Invalid file extension.
10043	273B	Invalid language driver.
10044	273C	Alias is not currently opened.

Error Code		
Decimal	*Hex*	*Error String*
10045	273D	Incompatible record structures.
10046	273E	Name is reserved by DOS.
10047	273F	Destination must be indexed.
10048	2740	Invalid index type.
10049	2741	Language drivers of table and index do not match.
10050	2742	Filter handle is invalid.
10051	2743	Invalid filter.
10052	2744	Invalid Table Create request.
10053	2745	Invalid Table Delete request.
10054	2746	Invalid Index Create request.
10055	2747	Invalid Index Delete request.
10056	2748	Invalid table specified.
10058	274A	Invalid time.
10059	274B	Invalid date.
10060	274C	Invalid date/time.
10061	274D	Tables in different directories.
10062	274E	Mismatch in the number of arguments.
10063	274F	Function not found in service library.
10064	2750	Must use `baseorder` for this operation.
10065	2751	Invalid procedure name.
10066	2752	The field map is invalid.
10241	2801	Record locked by another user.
10242	2802	Unlock failed.
10243	2803	Table is busy.
10244	2804	Directory is busy.
10245	2805	File is locked.
10246	2806	Directory is locked.
10247	2807	Record already locked by this session.
10248	2808	Object not locked.
10249	2809	Lock timeout.
10250	280A	Key group is locked.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
10251	280B	Table lock was lost.
10252	280C	Exclusive access was lost.
10253	280D	Table cannot be opened for exclusive use.
10254	280E	Conflicting record lock in this session.
10255	280F	A deadlock was detected.
10256	2810	A user transaction is already in progress.
10257	2811	No user transaction is currently in progress.
10258	2812	Record lock failed.
10259	2813	Couldn't perform the edit because another user changed the record.
10260	2814	Couldn't perform the edit because another user deleted or moved the record.
10497	2901	Insufficient field rights for operation.
10498	2902	Insufficient table rights for operation. Password required.
10499	2903	Insufficient family rights for operation.
10500	2904	This directory is read-only.
10501	2905	Database is read-only.
10502	2906	Trying to modify read-only field.
10503	2907	Encrypted dBASE tables not supported.
10504	2908	Insufficient SQL rights for operation.
10753	2A01	Field is not a BLOB.
10754	2A02	BLOB already opened.
10755	2A03	BLOB not opened.
10756	2A04	Operation not applicable.
10757	2A05	Table is not indexed.
10758	2A06	Engine not initialized.
10759	2A07	Attempt to reinitialize engine.
10760	2A08	Attempt to mix objects from different sessions.
10761	2A09	Paradox driver not active.
10762	2A0A	Driver not loaded.
10763	2A0B	Table is read only.
10764	2A0C	No associated index.

Error Code		
Decimal	*Hex*	*Error String*
10765	2A0D	Table(s) open. Cannot perform this operation.
10766	2A0E	Table does not support this operation.
10767	2A0F	Index is read only.
10768	2A10	Table does not support this operation because it is not uniquely indexed.
10769	2A11	Operation must be performed on the current session.
10770	2A12	Invalid use of keyword.
10771	2A13	Connection is in use by another statement.
10772	2A14	Passthrough SQL connection must be shared.
11009	2B01	Invalid function number.
11010	2B02	File or directory does not exist.
11011	2B03	Path not found.
11012	2B04	Too many open files. You may need to increase `MAXFILEHANDLE` limit in `IDAPI` configuration.
11013	2B05	Permission denied.
11014	2B06	Bad file number.
11015	2B07	Memory blocks destroyed.
11016	2B08	Not enough memory.
11017	2B09	Invalid memory block address.
11018	2B0A	Invalid environment.
11019	2B0B	Invalid format.
11020	2B0C	Invalid access code.
11021	2B0D	Invalid data.
11023	2B0F	Device does not exist.
11024	2B10	Attempt to remove current directory.
11025	2B11	Not same device.
11026	2B12	No more files.
11027	2B13	Invalid argument.
11028	2B14	Argument list is too long.
11029	2B15	Execution format error.
11030	2B16	Cross-device link.
11041	2B21	Math argument.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
11042	2B22	Result is too large.
11043	2B23	File already exists.
11047	2B27	Unknown internal operating system error.
11058	2B32	Share violation.
11059	2B33	Lock violation.
11060	2B34	Critical DOS error.
11061	2B35	Drive not ready.
11108	2B64	Not exact read/write.
11109	2B65	Operating system network error.
11110	2B66	Error from Novell file server.
11111	2B67	Novell server out of memory.
11112	2B68	Record already locked by this workstation.
11113	2B69	Record not locked.
11265	2C01	Network initialization failed.
11266	2C02	Network user limit exceeded.
11267	2C03	Wrong NET file version.
11268	2C04	Cannot lock network file.
11269	2C05	Directory is not private.
11270	2C06	Directory is controlled by other NET file.
11271	2C07	Unknown network error.
11272	2C08	Not initialized for accessing network files.
11273	2C09	Share not loaded. It is required to share local files.
11274	2C0A	Not on a network. Not logged in or wrong network driver.
11275	2C0B	Lost communication with SQL server.
11277	2C0D	Cannot locate or connect to SQL server.
11278	2C0E	Cannot locate or connect to network server.
11521	2D01	Optional parameter is required.
11522	2D02	Invalid optional parameter.
11777	2E01	Obsolete.
11778	2E02	Obsolete.
11779	2E03	Ambiguous use of ! (inclusion operator).

Error Code		
Decimal	*Hex*	*Error String*
11780	2E04	Obsolete.
11781	2E05	Obsolete.
11782	2E06	A `SET` operation cannot be included in its own grouping.
11783	2E07	Only numeric and date/time fields can be averaged.
11784	2E08	Invalid expression.
11785	2E09	Invalid `OR` expression.
11786	2E0A	Obsolete.
11787	2E0B	Bitmap.
11788	2E0C	`CALC` expression cannot be used in `INSERT`, `DELETE`, `CHANGETO`, and `SET` rows.
11789	2E0D	Type error in `CALC` expression.
11790	2E0E	`CHANGETO` can be used in only one query form at a time.
11791	2E0F	Cannot modify `CHANGED` table.
11792	2E10	A field can contain only one `CHANGETO` expression.
11793	2E11	A field cannot contain more than one expression to be inserted.
11794	2E12	Obsolete.
11795	2E13	`CHANGETO` must be followed by the new value for the field.
11796	2E14	Checkmark or `CALC` expressions cannot be used in `FIND` queries.
11797	2E15	Cannot perform operation on `CHANGED` table together with a `CHANGETO` query.
11798	2E16	Chunk.
11799	2E17	More than 255 fields in `ANSWER` table.
11800	2E18	`AS` must be followed by the name for the field in the `ANSWER` table.
11801	2E19	`DELETE` can be used in only one query form at a time.
11802	2E1A	Cannot perform operation on `DELETED` table together with a `DELETE` query.
11803	2E1B	Cannot delete from the `DELETED` table.
11804	2E1C	Example element is used in two fields with incompatible types or with a `BLOB`.
11805	2E1D	Cannot use example elements in an `OR` expression.
11806	2E1E	Expression in this field has the wrong type.
11807	2E1F	Extra comma found.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
11808	2E20	Extra `OR` found.
11809	2E21	One or more query rows do not contribute to the `ANSWER`.
11810	2E22	`FIND` can be used in only one query form at a time.
11811	2E23	`FIND` cannot be used with the `ANSWER` table.
11812	2E24	A row with `GROUPBY` must contain `SET` operations.
11813	2E25	`GROUPBY` can be used only in `SET` rows.
11814	2E26	Use only `INSERT`, `DELETE`, `SET`, or `FIND` in leftmost column.
11815	2E27	Use only one `INSERT`, `DELETE`, `SET`, or `FIND` per line.
11816	2E28	Syntax error in expression.
11817	2E29	`INSERT` can be used in only one query form at a time.
11818	2E2A	Cannot perform operation on `INSERTED` table together with an `INSERT` query.
11819	2E2B	`INSERT`, `DELETE`, `CHANGETO`, and `SET` rows may not be checked.
11820	2E2C	Field must contain an expression to insert (or be blank).
11821	2E2D	Cannot insert into the `INSERTED` table.
11822	2E2E	Variable is an array and cannot be accessed.
11823	2E2F	Label.
11824	2E30	Rows of example elements in `CALC` expression must be linked.
11825	2E31	Variable name is too long.
11826	2E32	Query may take a long time to process.
11827	2E33	Reserved word or one that can't be used as a variable name.
11828	2E34	Missing comma.
11829	2E35	Missing right parenthesis.
11830	2E36	Missing right quote.
11831	2E37	Cannot specify duplicate column names.
11832	2E38	Query has no checked fields.
11833	2E39	Example element has no defining occurrence.
11834	2E3A	No grouping is defined for `SET` operation.
11835	2E3B	Query makes no sense.
11836	2E3C	Cannot use patterns in this context.
11837	2E3D	Date does not exist.

Error Code		
Decimal	*Hex*	*Error String*
11838	2E3E	Variable has not been assigned a value.
11839	2E3F	Invalid use of example element in summary expression.
11840	2E40	Incomplete query statement. Query only contains a SET definition.
11841	2E41	Example element with `!` makes no sense in expression.
11842	2E42	Example element cannot be used more than twice with a `!` query.
11843	2E43	Row cannot contain expression.
11844	2E44	Obsolete.
11845	2E45	Obsolete.
11846	2E46	No permission to insert or delete records.
11847	2E47	No permission to modify field.
11848	2E48	Field not found in table.
11849	2E49	Expecting a column separator in table header.
11850	2E4A	Expecting a column separator in table.
11851	2E4B	Expecting a column name in table.
11852	2E4C	Expecting table name.
11853	2E4D	Expecting consistent number of columns in all rows of table.
11854	2E4E	Cannot open table.
11855	2E4F	Field appears more than once in table.
11856	2E50	This `DELETE`, `CHANGE`, or `INSERT` query has no `ANSWER`.
11857	2E51	Query is not prepared. Properties unknown.
11858	2E52	`DELETE` rows cannot contain quantifier expression.
11859	2E53	Invalid expression in `INSERT` row.
11860	2E54	Invalid expression in `INSERT` row.
11861	2E55	Invalid expression in `SET` definition.
11862	2E56	Row use.
11863	2E57	`SET` keyword expected.
11864	2E58	Ambiguous use of example element.
11865	2E59	Obsolete.
11866	2E5A	Obsolete.
11867	2E5B	Only numeric fields can be summed.
11868	2E5C	Table is write protected.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
11869	2E5D	Token not found.
11870	2E5E	Cannot use example element with `!` more than once in a single row.
11871	2E5F	Type mismatch in expression.
11872	2E60	Query appears to ask two unrelated questions.
11873	2E61	Unused `SET` row.
11874	2E62	`INSERT`, `DELETE`, `FIND`, and `SET` can be used only in the leftmost column.
11875	2E63	`CHANGETO` cannot be used with `INSERT`, `DELETE`, `SET`, or `FIND`.
11876	2E64	Expression must be followed by an example element defined in a `SET`.
11877	2E65	Lock failure.
11878	2E66	Expression is too long.
11879	2E67	Refresh exception during query.
11880	2E68	Query canceled.
11881	2E69	Unexpected database engine error.
11882	2E6A	Not enough memory to finish operation.
11883	2E6B	Unexpected exception.
11884	2E6C	Feature not implemented yet in query.
11885	2E6D	Query format is not supported.
11886	2E6E	Query string is empty.
11887	2E6F	Attempted to prepare an empty query.
11888	2E70	Buffer too small to contain query string.
11889	2E71	Query was not previously parsed or prepared.
11890	2E72	Function called with bad query handle.
11891	2E73	QBE syntax error.
11892	2E74	Query extended syntax field count error.
11893	2E75	Field name in sort or field clause not found.
11894	2E76	Table name in sort or field clause not found.
11895	2E77	Operation is not supported on BLOB fields.
11896	2E78	General BLOB error.
11897	2E79	Query must be restarted.
11898	2E7A	Unknown answer table type.

Error Code		
Decimal	*Hex*	*Error String*
11926	2E96	Blob cannot be used as grouping field.
11927	2E97	Query properties have not been fetched.
11928	2E98	Answer table is of unsuitable type.
11929	2E99	Answer table is not yet supported under server alias.
11930	2E9A	Non-null BLOB field required. Can't insert records.
11931	2E9B	Unique index required to perform `CHANGETO`.
11932	2E9C	Unique index required to delete records.
11933	2E9D	Update of table on the server failed.
11934	2E9E	Can't process this query remotely.
11935	2E9F	Unexpected end of command.
11936	2EA0	Parameter not set in query string.
11937	2EA1	Query string is too long.
11946	2EAA	No such table or correlation name.
11947	2EAB	Expression has ambiguous data type.
11948	2EAC	Field in `ORDER BY` must be in resultset.
11949	2EAD	General parsing error.
11950	2EAE	Record or field constraint failed.
11951	2EAF	Field in group by must be in resultset.
11952	2EB0	User-defined function is not defined.
11953	2EB1	Unknown error from user-defined function.
11954	2EB2	Single-row subquery produced more than one row.
11955	2EB3	Expressions in `GROUP BY` are not supported.
11956	2EB4	Queries on text or ASCII tables is not supported.
11957	2EB5	`ANSI` join keywords `USING` and `NATURAL` are not supported in this release.
11958	2EB6	`SELECT DISTINCT` may not be used with `UNION` unless `UNION ALL` is used.
11959	2EB7	`GROUP BY` is required when both aggregate and nonaggregate fields are used in resultset.
11960	2EB8	`INSERT` and `UPDATE` operations are not supported on autoincrement field type.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
11961	2EB9	`UPDATE` on primary key of a master table may modify more than one record.
12033	2F01	Interface mismatch. Engine version different.
12034	2F02	Index is out of date.
12035	2F03	Older version (see context).
12036	2F04	VAL file is out of date.
12037	2F05	BLOB file version is too old.
12038	2F06	Query and engine DLLs are mismatched.
12039	2F07	Server is incompatible version.
12040	2F08	Higher table level required.
12289	3001	Capability not supported.
12290	3002	Not implemented yet.
12291	3003	SQL replicas not supported.
12292	3004	Non-BLOB column in table required to perform operation.
12293	3005	Multiple connections not supported.
12294	3006	Full dBASE expressions not supported.
12545	3101	Invalid database alias specification.
12546	3102	Unknown database type.
12547	3103	Corrupt system configuration file.
12548	3104	Network type unknown.
12549	3105	Not on the network.
12550	3106	Invalid configuration parameter.
12801	3201	Object implicitly dropped.
12802	3202	Object may be truncated.
12803	3203	Object implicitly modified.
12804	3204	Should field constraints be checked?
12805	3205	Validity check field modified.
12806	3206	Table level changed.
12807	3207	Copy linked tables?
12809	3209	Object implicitly truncated.
12810	320A	Validity check will not be enforced.
12811	320B	Multiple records found, but only one was expected.

Error Code		
Decimal	*Hex*	*Error String*
12812	320C	Field will be trimmed. Cannot put master records into problem table.
13057	3301	File already exists.
13058	3302	BLOB has been modified.
13059	3303	General SQL error.
13060	3304	Table already exists.
13061	3305	Paradox 1.0 tables are not supported.
13062	3306	Update aborted.
13313	3401	Different sort order.
13314	3402	Directory in use by earlier version of Paradox.
13315	3403	Needs Paradox 3.5–compatible language driver.
13569	3501	Data dictionary is corrupt.
13570	3502	Data dictionary info BLOB corrupted.
13571	3503	Data dictionary schema is corrupt.
13572	3504	Attribute type exists.
13573	3505	Invalid object type.
13574	3506	Invalid relation type.
13575	3507	View already exists.
13576	3508	No such view exists.
13577	3509	Invalid record constraint.
13578	350A	Object is in a logical DB.
13579	350B	Dictionary already exists.
13580	350C	Dictionary does not exist.
13581	350D	Dictionary database does not exist.
13582	350E	Dictionary info is out of date. Needs refreshed.
13584	3510	Invalid dictionary name.
13585	3511	Dependent objects exist.
13586	3512	Too many relationships for this object type.
13587	3513	Relationships to the object exist.
13588	3514	Dictionary exchange file is corrupt.
13589	3515	Dictionary exchange file version mismatch.
13590	3516	Dictionary object type mismatch.

continues

TABLE B.1 Continued

Error Code		
Decimal	*Hex*	*Error String*
13591	3517	Object exists in target dictionary.
13592	3518	Cannot access data dictionary.
13593	3519	Cannot create data dictionary.
13594	351A	Cannot open database.
15873	3E01	Wrong driver name.
15874	3E02	Wrong system version.
15875	3E03	Wrong driver version.
15876	3E04	Wrong driver type.
15877	3E05	Cannot load driver.
15878	3E06	Cannot load language driver.
15879	3E07	Vendor initialization failed.
15880	3E08	Your application is not enabled for use with this driver.