Dynamic Link Libraries

CHAPTER

9

IN THIS CHAPTER

This chapter discusses Win32 dynamic link libraries, otherwise known as *DLLs*. DLLs are a key component to writing any Windows application. This chapter discusses several aspects of using and creating DLLs. It gives you an overview of how DLLs work and discusses how to create and use DLLs. You learn different methods of loading DLLs and linking to the procedures and functions they export. This chapter also covers the use of callback functions and illustrates how to share DLL data among different calling processes.

What Exactly Is a DLL?

Dynamic link libraries are program modules that contain code, data, or resources that can be shared among many Windows applications. One of the primary uses of DLLs is to enable applications to load code to execute at runtime instead of linking that code to the application at compile time. Therefore, multiple applications can simultaneously use the same code provided by the DLL. In fact, the files `Kernel32.dll`, `User32.dll`, and `GDI32.dll` are three DLLs on which Win32 relies heavily. `Kernel32.dll` is responsible for memory, process, and thread management. `User32.dll` contains routines for the user interface that deal with the creation of windows and the handling of Win32 messages. `GDI32.dll` deals with graphics. You'll also hear of other system DLLs, such as `AdvAPI32.dll` and `ComDlg32.dll`, which deal with object security/Registry manipulation and common dialog boxes, respectively.

Another advantage to using DLLs is that your applications become modular. This simplifies updating your applications because you need to replace only DLLs instead of replacing the entire application. The Windows environment presents a typical example of this type of modularity. Each time you install a new device, you also install a device driver DLL to enable that device to communicate with Windows. The advantage to modularity becomes obvious when you imagine having to reinstall Windows each time you install a new device to your system.

On disk, a DLL is basically the same as a Windows EXE file. One major difference is that a DLL isn't an independently executable file, although it may contain executable code. The most common DLL file extension is `.dll`. Other file extensions are `.drv` for device drivers, `.sys` for system files, and `.fon` for font resources, which contain no executable code.

NOTE

Delphi introduces a special-purpose DLL known as a *package*, which is used in the Delphi and C++Builder environments. We'll go into greater depth on packages in Chapter 21, "Writing Delphi Custom Components."

DLLs share their code with other applications through a process called *dynamic linking*, which is discussed later in this chapter. In general, when an application uses a DLL, the Win32 system ensures that only one copy of that DLL resides in memory. It does this by using *memory-mapped files*. The DLL is first loaded into the Win32 system's global heap. It's then mapped

into the address space of the calling process. In the Win32 system, each process is given its own 32-bit linear address space. When the DLL is loaded by multiple processes, each process receives its own image of the DLL. Therefore, processes don't share the same physical code, data, or resources, as was the case in 16-bit Windows. In Win32, the DLL appears as though it's actually code belonging to the calling process. For more information on Win32 constructs, you can refer to Chapter 3, "The Win32 API."

This doesn't mean that when multiple processes load a DLL, the physical memory is consumed by each usage of the DLL. The DLL image is placed into each process's address space by mapping its image from the system's global heap to the address space of each process that uses the DLL, at least in the ideal scenario (see the sidebar "Setting a DLL's Preferred Base Address").

Setting a DLL's Preferred Base Address

DLL code is only shared between processes if the DLL can be loaded into the process address space of all interested clients at the DLL's preferred base address. If the preferred base address and range of the DLL overlaps with something already allocated in a process, the Win32 loader has to relocate the entire DLL image to some other base address. When that happens, none of the relocated DLL image is shared with any other process in the system—each relocated DLL instance consumes its own chunk of physical memory and swap file space.

It's critical that you set the base address of every DLL you produce to a value that doesn't conflict with or overlap other address ranges used by your application by using the `$IMAGEBASE` directive.

If your DLL will be used by multiple applications, choose a unique base address that's unlikely to collide with application addresses at the low end of the process virtual address range or common DLLs (such as VCL packages) at the high end of the address range. The default base address for all executable files (EXEs and DLLs) is `$400000`, which means unless you change your DLL base address, it will always collide with the base address of its host EXE and therefore never be shared between processes.

There's another side benefit to base address loading. Because the DLL doesn't require relocation or fixes (which is usually the case) and because it's stored on a local disk drive, the DLL's memory pages are mapped directly onto the DLL file on disk. The DLL code does not consume any space in the system's page file (as called a *swap file*). This is why the system's total committed page count and size statistics can be much larger than the system swap file plus RAM.

You'll find detailed information on using the `$IMAGEBASE` directive by looking up "Image Base Address" in the Delphi 5 online help.

Following are some terms you'll need to know in regard to DLLs:

- *Application*. A Windows program residing in an `.exe` file.
- *Executable*. A file containing executable code. Executable files include `.dll` and `.exe` files.
- *Instance*. When referring to applications and DLLs, an *instance* is the occurrence of an executable. Each instance can be referred to by an *instance handle*, which is assigned by the Win32 system. When an application is run twice, for example, there are two instances of that application and, therefore, two instance handles. When a DLL is loaded, there's an instance of that DLL as well as a corresponding instance handle. The term *instance*, as used here, should not be confused with the instance of a class.
- *Module*. In 32-bit Windows, *module* and *instance* can be used synonymously. This differs from 16-bit Windows, in which the system maintains a database to manage modules and provides a module handle for each module. In Win32, each instance of an application gets its own address space; therefore, there's no need for a separate module identifier. However, Microsoft still uses the term in its own documentation. Just be aware that *module* and *instance* are one and the same.
- *Task*. Windows is a multitasking (or task-switching) environment. It must be able to allocate system resources and time to the various instances running under it. It does this by maintaining a task database that maintains instance handles and other necessary information to enable it to perform its task-switching functions. The *task* is the element to which Windows grants resources and time blocks.

Static Linking Versus Dynamic Linking

Static linking refers to the method by which the Delphi compiler resolves a function or procedure call to its executable code. The function's code can exist in the application's `.dpr` file or in a unit. When linking your applications, these functions and procedures become part of the final executable file. In other words, on disk, each function will reside at a specific location in the program's `.exe` file.

A function's location also is predetermined at a location relative to where the program is loaded in memory. Any calls to that function cause program execution to jump to where the function resides, execute the function, and then return to the location from which it was called. The relative address of the function is resolved during the linking process.

This is a loose description of a more complex process that the Delphi compiler uses to perform static linking. However, for the purpose of this book, you don't need to understand the underlying operations that the compiler performs to use DLLs effectively in your applications.

NOTE

Delphi implements a *smart linker* that automatically removes functions, procedures, variables, and typed constants that never get referenced in the final project. Therefore, functions that reside in large units that never get used don't become a part of your EXE file.

Suppose you have two applications that use the same function that resides in a unit. Both applications, of course, would have to include the unit in their `uses` statements. If you ran both applications simultaneously in Windows, the function would exist twice in memory. If you had a third application, there would be a third instance of the function in memory, and you would be using up three times its memory space. This small example illustrates one of the primary reasons for dynamic linking. Through dynamic linking, this function resides in a DLL. Then, when an application loads the function into memory, all other applications that need to reference it can share its code by mapping the image of the DLL into their own process memory space. The end result is that the DLL's function exists only once in memory—theoretically.

With *dynamic linking*, the link between a function call and its executable code is resolved at runtime by using an external reference to the DLL's function. These references can be declared in the application, but usually they're placed in a separate `import` unit. The `import` unit declares the imported functions and procedures and defines the various types required by DLL functions.

For example, suppose you have a DLL named `MaxLib.dll` that contains a function:

```
function Max(i1, I2: integer): integer;
```

This function returns the higher of the two integers passed to it. A typical `import` unit would look like this:

```
unit MaxUnit;
interface
function Max(I1, I2: integer): integer;
implementation
function Max; external 'MAXLIB';
end.
```

You'll notice that although this looks somewhat like a typical unit, it doesn't define the function `Max()`. The keyword `external` simply says that the function resides in the DLL of the name that follows it. To use this unit, an application would simply place `MaxUnit` in its `uses` statement. When the application runs, the DLL is loaded into memory automatically, and any calls to `Max()` are linked to the `Max()` function in the DLL.

This illustrates one of two ways to load a DLL; it's called *implicit loading*, which causes Windows to automatically load the DLL when the application loads. Another method is to *explicitly load* the DLL; this is discussed later in this chapter.

Why Use DLLs?

There are several reasons for using DLLs, some of which were mentioned earlier. In general, you use DLLs to share code or system resources, to hide your code implementation or low-level system routines, or to design custom controls. We discuss these topics in the following sections.

Sharing Code, Resources, and Data with Multiple Applications

Earlier in this chapter, you learned that the most common reason for creating a DLL is to share code. Unlike units, which enable you to share code with different Delphi applications, DLLs enable you to share code with any Windows application that can call functions from DLLs.

Additionally, DLLs provide a way for you to share resources such as bitmaps, fonts, icons, and so on that you normally would put into a resource file and link directly into your application. If you place these resources into a DLL, many applications can make use of them without using up the memory required to load them more often.

Back in 16-bit Windows, DLLs had their own data segment, so all applications that used a DLL could access the same data-global and static variables. In the Win32 system, this is a different story. Because the DLL image is mapped to each process's address space, all data in the DLL belongs to that process. One thing worth mentioning here is that although the DLL's data isn't shared between different processes, it's shared by multiple threads within the same process. Because threads execute independently of one another, you must take precautions not to cause conflicts when accessing a DLL's global data.

This doesn't mean that there aren't ways to make multiple processes share data made accessible through a DLL. One technique would be to create a shared memory area (using a memory-mapped file) from within the DLL. Each application using that DLL would be able to read the data stored in the shared memory area. This technique is shown later in the chapter.

Hiding Implementation

In some cases, you might want to hide the details of the routines that you make available from a DLL. Regardless of your reason for deciding to hide your code's implementation, a DLL provides a way for you to make your functions available to the public and not give away your source code in doing so. All you need to do is provide an interface unit to enable others to access your DLL. If you're thinking that this is already possible with Delphi compiled units (DCUs), consider that DCUs apply only to other Delphi applications that are created with the same version of Delphi. DLLs are language-independent, so you can create a DLL that can be used by C++, VB, or any other language that supports DLLs.

The `Windows` unit is the interface unit to the Win32 DLLs. The Win32 API unit source files are included with Delphi 5. One of the files you get is `Windows.pas`, the source to the Windows unit. In `Windows.pas`, you find function definitions such as the following in the `interface` section:

```
function ClientToScreen(Hwnd: HWND; var lpPoint: TPoint): BOOL; stdcall;
```

The corresponding link to the DLL is in the `implementation` section, as in the following example:

```
function ClientToScreen; external user32 name 'ClientToScreen';
```

This basically says that the procedure `ClientToScreen()` exists in the dynamic link library `User32.dll`, and its name is `ClientToScreen`.

Custom Controls

Custom controls usually are placed in DLLs. These controls aren't the same as Delphi custom components. Custom controls are registered under Windows and can be used by any Windows development environment. These types of custom controls are placed in DLLs to conserve memory by having only one copy of the control's code in memory when multiple copies of the control are being used.

NOTE

The old custom control DLL mechanism is extremely crude and inflexible, which is why Microsoft now uses OLE and ActiveX controls. These older forms of custom controls are rare.

Creating and Using DLLs

The following sections take you through the process of actually creating a DLL with Delphi. You'll see how to create an interface unit so that you can make your DLLs available to other programs. You'll also learn how to incorporate Delphi forms into DLLs before going on to using DLLs in Delphi.

Counting Your Pennies (A Simple DLL)

The following DLL example illustrates placing a routine that's a favorite of many computer science professors into a DLL. The routine converts a monetary amount in pennies to the minimum number of nickels, dimes, or quarters needed to match the total number of pennies.

A Basic DLL

The library contains the `PenniesToCoins()` method. Listing 9.1 shows the complete DLL project.

LISTING 9.1 PenniesLib.dpr, a DLL to Convert Pennies to Other Coins

```
library PenniesLib;
{$DEFINE PENNIESLIB}
uses
  SysUtils,
  Classes,
  PenniesInt;

function PenniesToCoins(TotPennies: word;
   CoinsRec: PCoinsRec): word; StdCall;
begin
  Result := TotPennies;  // Assign value to Result
  { Calculate the values for quarters, dimes, nickels, pennies }
  with CoinsRec^ do
  begin
    Quarters    := TotPennies div 25;
    TotPennies  := TotPennies - Quarters * 25;
    Dimes       := TotPennies div 10;
    TotPennies  := TotPennies - Dimes * 10;
    Nickels     := TotPennies div 5;
    TotPennies  := TotPennies - Nickels * 5;
    Pennies     := TotPennies;
  end;
end;

{ Export the function by name }
exports
  PenniesToCoins;
end.
```

Notice that this library uses the unit PenniesInt. We'll discuss this in more detail momentarily.

The exports clause specifies which functions or procedures in the DLL get exported and made available to calling applications.

Defining an Interface Unit

Interface units enable users of your DLL to statically import your DLL's routines into their applications by just placing the import unit's name in their module's uses statement. Interface units also allow the DLL writer to define common structures used by both the library and the calling application. We demonstrate that here with the interface unit. Listing 9.2 shows the source code to PenniesInt.pas.

LISTING 9.2 PenniesInt.pas, the interface Unit for PenniesLib.Dll

```
unit PenniesInt;
{ Interface routine for PENNIES.DLL }

interface
type

  { This record will hold the denominations after the conversions have
    been made }
  PCoinsRec = ^TCoinsRec;
  TCoinsRec = record
    Quarters,
    Dimes,
    Nickels,
    Pennies: word;
  end;

{$IFNDEF PENNIESLIB}
{ Declare function with export keyword }

function PenniesToCoins(TotPennies: word;
  CoinsRec: PCoinsRec): word; StdCall;
{$ENDIF}

implementation

{$IFNDEF PENNIESLIB}
{ Define the imported function }
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
{$ENDIF}

end.
```


In the type section of this project, you declare the record TCoinsRec as well as a pointer to this record. This record will hold the denominations that will make up the penny amount passed into the PenniesToCoins() function. The function takes two parameters—the total amount of money in pennies and a pointer to a TCoinsRec variable. The result of the function is the amount of pennies passed in.

PenniesInt.pas declares the function that the PenniesLib.dll exports in its interface section. The definition of the PenniesToCoins() function is placed in the implementation section. This definition specifies that the function is an external function existing in the DLL file

PenniesLib.dll. It links to the DLL function by the name of the function. Notice that you used a compiler directive PENNIESLIB to conditionally compile the declaration of the PenniesToCoins() function. You do this because it's not necessary to link this declaration when compiling the interface unit for the library. This allows you to share the interface unit's type definitions with both the library and any applications that intend to use the library. Any changes to the structures used by both only have to be made in the interface unit.

TIP

To define an application-wide conditional directive, specify the conditional in the Directories/Conditionals page of the Project, Options dialog box. Note that you must rebuild your project for changes to conditional defines to take effect because Make logic doesn't reevaluate conditional defines.

NOTE

The following definition shows one of two ways to import a DLL function:

```
function PenniesToCoins; external 'PENNIESLIB.DLL' index 1;
```

This method is called *importing by ordinal*. The other method by which you can import DLL functions is *by name*:

```
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
```

The by-name method uses the name specified after the name keyword to determine which function to link to in the DLL.

The by-ordinal method reduces the DLL's load time because it doesn't have to look up the function name in the DLL's name table. However, this method isn't the preferred method in Win32. Importing by name is the preferred technique so that applications won't be hypersensitive to relocation of DLL entry points as DLLs get updated over time. When you import by ordinal, you are binding to a place in the DLL. When you import by name, you're binding to the function name, regardless of where it happens to be placed in the DLL.

If this were an actual DLL that you planned to deploy, you would provide both PenniesLib.dll and PenniesInt.pas to your users. This would enable them to use the DLL by defining the types and functions in PenniesInt.pas that PenniesLib.dll requires. Additionally, programmers using different languages, such as C++, could convert PenniesInt.pas to their languages, thus enabling them to use your DLL in their development environments. You'll find a sample project that uses PenniesLib.dll on the CD that accompanies this book.

Displaying Modal Forms from DLLs

This section shows you how to make modal forms available from a DLL. One reason why placing commonly used forms in a DLL is beneficial is that it enables you to extend your forms for use with any Windows application or development environment, such as C++ and Visual Basic.

To do this, you remove your DLL-based form from the list of autocreated forms.

We've created such a form that contains a `TCalendar` component on the main form. The calling application will call a DLL function that will invoke this form. When the user selects a day on the calendar, the date will be returned to the calling application.

Listing 9.3 shows the source for `CalendarLib.dpr`, the DLL project file. Listing 9.4, in the section, "Displaying Modeless Forms from DLLs," shows the source code for `DllFrm.pas`, the DLL form's unit, which illustrates how to encapsulate the form into a DLL.

LISTING 9.3 Library Project Source—`CalendarLib.dpr`

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
    procedure calDllCalendarDblClick(Sender: TObject);
  end;

{ Declare the export function }
function ShowCalendar(AHandle: THandle; ACaption: String):
  TDateTime; StdCall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime;
var
  DLLForm: TDllForm;
```

continues

LISTING 9.3 Continued

```
begin
  // Copy application handle to DLL's TApplication object
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  try
    DLLForm.Caption := ACaption;
    DLLForm.ShowModal;
    // Pass the date back in Result
    Result := DLLForm.calDLLCalendar.CalendarDate;
  finally
    DLLForm.Free;
  end;
end;

procedure TDLLForm.calDllCalendarDblClick(Sender: TObject);
begin
  Close;
end;

end.
```

The main form in this DLL is incorporated into the exported function. Notice that the `DLLForm` declaration was removed from the `interface` section and declared inside the function instead.

The first thing that the DLL function does is to assign the `AHandle` parameter to the `Application.Handle` property. Recall from Chapter 4, "Application Frameworks and Design Concepts," that Delphi projects, including library projects, contain a global `Application` object. In a DLL, this object is separate from the `Application` object that exists in the calling application. For the form in the DLL to truly act as a modal form for the calling application, you must assign the handle of the calling application to the DLL's `Application.Handle` property, as has been illustrated. Not doing so will result in erratic behavior, especially when you start minimizing the DLL's form. Also, as shown, you must make sure not to pass `nil` as the owner of the DLL's form.

After the form is created, you assign the `ACaption` string to the `Caption` of the DLL form. It's then displayed modally. When the form closes, the date selected by the user in the `TCalendar` component is passed back to the calling function. The form closes after the user double-clicks the `TCalendar` component.

CAUTION

ShareMem must be the first unit in your library's uses clause and your project's (select View, Project Source) uses clause if your DLL exports any procedures or functions that pass strings or dynamic arrays as parameters or function results. This applies to all strings passed to and from your DLL—even those nested in records and classes. ShareMem is the interface unit to the Borlndmm.dll shared memory manager, which must be deployed along with your DLL. To avoid using Borlndmm.dll, pass string information using PChar or ShortString parameters.

ShareMem is only required when heap-allocated strings or dynamic arrays are passed between modules, and such transfers also transfer ownership of that string memory. Typecasting an internal string to a PChar and passing it to another module as a PChar does not transfer ownership of the string memory to the calling module, so ShareMem is not required.

Note that this ShareMem issue applies only to DelphiC++Builder DLLs that pass strings or dynamic arrays to other Delphi/BCB DLLs or EXEs. You should never expose Delphi strings or dynamic arrays (as parameters or function results of DLL exported functions) to non-Delphi DLLs or host apps. They won't know how to dispose of the Delphi items correctly.

Also, ShareMem is never required between modules built with packages. The memory allocator is implicitly shared between packaged modules.

This is all that's required when encapsulating a modal form into a DLL. In the next section, we'll discuss displaying a modeless form in a DLL.

Displaying Modeless Forms from DLLs

To illustrate placing modeless forms in a DLL, we'll use the same calendar form as the previous section.

When displaying modeless forms from a DLL, the DLL must provide two routines. The first routine must take care of creating and displaying the form. A second routine is required to free the form. Listing 9.4 displays the source code for the illustration of a modeless form in a DLL.

LISTING 9.4 A Modeless Form in a DLL

```
unit DLLFrm;

interface

uses
```

continues

LISTING 9.4 Continued

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
  end;

{ Declare the export function }
function ShowCalendar(AHandle: THandle; ACaption: String):
  Longint; stdCall;
procedure CloseCalendar(AFormRef: Longint); stdcall;


implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): Longint;
var
  DLLForm: TDllForm;
begin
  // Copy application handle to DLL's TApplication object
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  Result := Longint(DLLForm);
  DLLForm.Caption := ACaption;
  DLLForm.Show;
end;

procedure CloseCalendar(AFormRef: Longint);
begin
  if AFormRef > 0 then
    TDLLForm(AFormRef).Release;
end;

end.
```

This listing displays the routines `ShowCalendar()` and `CloseCalendar()`. `ShowCalendar()` is similar to the same function in the modal form example in that it makes the assignment of the calling application's application handle to the DLL's application handle and creates the form.

Instead of calling `ShowModal()`, however, this routine calls `Show()`. Notice that it doesn't free the form. Also, notice that the function returns a `longint` value to which you assign the `DLLForm` instance. This is because a reference of the created form must be maintained, and it's best to have the calling application maintain this instance. This would take care of any issues regarding other applications calling this DLL and creating another instance of the form.

In the `CloseCalendar()` procedure, you simply check for a valid reference to the form and invoke its `Release()` method. Here, the calling application should pass back the same reference that was returned to it from `ShowCalendar()`.

When using such a technique, you must be careful that your DLL never frees the form independently of the host. If it does (for example, returning `caFree` in `CanClose()`), the call to `CloseCalendar()` will crash.

Demos of both the modal and modeless forms are on the CD that accompanies this book.

Using DLLs in Your Delphi Applications

Earlier in this chapter, you learned that there are two ways to load or import DLLs: implicitly and explicitly. Both techniques are illustrated in this section with the DLLs just created.

The first DLL created in this chapter included an `interface` unit. You'll use this `interface` unit in the following example to illustrate implicit linking of a DLL. The sample project's main form has a `TMaskEdit`, `TButton`, and nine `TLabel` components.

In this application, the user enters an amount of pennies. Then, when the user clicks the button, the labels will show the breakdown of denominations of change adding up to that amount. This information is obtained from the `PenniesLib.dll` exported function `PenniesToCoins()`.

The main form is defined in the unit `MainFrm.pas` shown in Listing 9.5.

LISTING 9.5 Main Form for the Pennies Demo

```
unit MainFrm;

interface

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

type

  TMainForm = class(TForm)
    lblTotal: TLabel;
    lblQlbl: TLabel;
```

continues

LISTING 9.5 Continued

```
    lblDlbl: TLabel;
    lblNlbl: TLabel;
    lblPlbl: TLabel;
    lblQuarters: TLabel;
    lblDimes: TLabel;
    lblNickels: TLabel;
    lblPennies: TLabel;
    btnMakeChange: TButton;
    meTotalPennies: TMaskEdit;
    procedure btnMakeChangeClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
uses PenniesInt;  // Use an interface unit

{$R *.DFM}

procedure TMainForm.btnMakeChangeClick(Sender: TObject);
var
  CoinsRec: TCoinsRec;
  TotPennies: word;
begin
  { Call the DLL function to determine the minimum coins required
    for the amount of pennies specified. }
  TotPennies := PenniesToCoins(StrToInt(meTotalPennies.Text), @CoinsRec);
  with CoinsRec do
  begin
    { Now display the coin information }
    lblQuarters.Caption := IntToStr(Quarters);
    lblDimes.Caption    := IntToStr(Dimes);
    lblNickels.Caption  := IntToStr(Nickels);
    lblPennies.Caption  := IntToStr(Pennies);
  end
end;

end.
```

Notice that `MainFrm.pas` uses the unit `PenniesInt`. Recall that `PenniesInt.pas` includes the external declarations to the functions existing in `PenniesLib.dpr`. When this application runs, the Win32 system automatically loads `PenniesLib.dll` and maps it to the process address space for the calling application.

Usage of an `import` unit is optional. You can remove `PenniesInt` from the `uses` statement and place the `external` declaration to `PenniesToCoins()` in the `implementation` section of `MainFrm.pas`, as in the following code:

```
implementation

function PenniesToCoins(TotPennies: word; ChangeRec: PChangeRec): word;
   ➥StdCall external 'PENNIESLIB.DLL';
```

You also would have to define `PChangeRec` and `TChangeRec` again in `MainFrm.pas`, or you can compile your application using the compiler directive `PENNIESLIB`. This technique is fine in the case where you only need access to a few routines from a DLL. In many cases, you'll find that you require not only the external declarations to the DLL's routines but also access to the types defined in the `interface` unit.

NOTE

Many times, when using another vendor's DLL, you won't have a Pascal `interface` unit; instead, you'll have a C/C++ import library. In this case, you have to translate the library to a Pascal equivalent `interface` unit.

You'll find this demo on the accompanying CD.

Loading DLLs Explicitly

Although loading DLLs implicitly is convenient, it isn't always the most desired method. Suppose you have a DLL that contains many routines. If it's likely that your application will never call any of the DLL's routines, it would be a waste of memory to load the DLL every time your application runs. This is especially true when using multiple DLLs with one application. Another example is when using DLLs as large objects: a standard list of functions that are implemented by multiple DLLs but do slightly different things, such as printer drivers and file format readers. In this situation, it would be beneficial to load the DLL when specifically requested to do so by the application. This is referred to as *explicitly loading* a DLL.

To illustrate explicitly loading a DLL, we return to the sample DLL with a modal form. Listing 9.6 shows the code for the main form of the application that demonstrates explicitly loading this DLL. The project file for this application is on the accompanying CD.

LISTING 9.6 Main Form for Calendar DLL Demo Application

```
unit MainFfm;

interface

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

type
  { First, define a procedural data type, this should reflect the
    procedure that is exported from the DLL. }
  TShowCalendar = function (AHandle: THandle; ACaption: String):
    TDateTime; StdCall;

  { Create a new exception class to reflect a failed DLL load }
  EDLLLoadError = class(Exception);

  TMainForm = class(TForm)
    lblDate: TLabel;
    btnGetCalendar: TButton;
    procedure btnGetCalendarClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnGetCalendarClick(Sender: TObject);
var
  LibHandle   : THandle;
  ShowCalendar: TShowCalendar;
begin

  { Attempt to load the DLL }
  LibHandle := LoadLibrary('CALENDARLIB.DLL');
  try
    { If the load failed, LibHandle will be zero.
      If this occurs, raise an exception. }
```

```
    if LibHandle = 0 then
      raise EDLLLoadError.Create('Unable to Load DLL');
    { If the code makes it here, the DLL loaded successfully, now obtain
      the link to the DLL's exported function so that it can be called. }
    @ShowCalendar := GetProcAddress(LibHandle, 'ShowCalendar');
    { If the function is imported successfully, then set
      lblDate.Caption to reflect the returned date from
      the function. Otherwise, show the return raise an exception. }
    if not (@ShowCalendar = nil) then
      lblDate.Caption := DateToStr(ShowCalendar(Application.Handle, Caption))
    else
      RaiseLastWin32Error;
  finally
    FreeLibrary(LibHandle); // Unload the DLL.
  end;
end;

end.
```

This unit first defines a procedural data type, `TShowCalendar`, that reflects the definition of the function it will be using from `CalendarLib.dll`. It then defines a special exception, which is raised when there's a problem loading the DLL. In the `btnGetCalendarClick()` event handler, you'll notice the use of three Win32 API functions: `LoadLibrary()`, `FreeLibrary()`, and `GetProcAddress()`.

`LoadLibrary()` is defined this way:

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

This function loads the DLL module specified by `lpLibFileName` and maps it into the address space of the calling process. If this function succeeds, it returns a handle to the module. If it fails, it returns the value `0`, and an exception is raised. You can look up `LoadLibrary()` in the online help for detailed information on its functionality and possible return error values.

`FreeLibrary()` is defined like this:

```
function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;
```

`FreeLibrary()` decrements the instance count of the library specified by `LibModule`. It removes the library from memory when the library's instance count is zero. The instance count keeps track of the number of tasks using the DLL.

Here's how `GetProcAddress()` is defined:

```
function GetProcAddress(hModule: HMODULE; lpProcName: LPCSTR):
  FARPROC; stdcall
```

GetProcAddress() returns the address of a function within the module specified in its first parameter, hModule. hModule is the THandle returned from a call to LoadLibrary(). If GetProcAddress() fails, it returns nil. You must call GetLastError() for extended error information.

In Button1's OnClick event handler, LoadLibrary() is called to load CALDLL. If it fails to load, an exception is raised. If the call is successful, a call to the window's GetProcAddress() is made to get the address of the function ShowCalendar(). Prepending the procedural data type variable ShowCalendar with the address of operator (@) character prevents the compiler from issuing a type mismatch error due to its strict type-checking. After obtaining the address of ShowCalendar(), you can use it as defined by TShowCalendar. Finally, FreeLibrary() is called within the finally block to ensure that the library is freed from memory when no longer required.

You can see that the library is loaded and freed each time this function is called. If this function was called only once during the run of an application, it becomes apparent how explicit loading can save much-needed and often limited memory resources. On the other hand, if this function were called frequently, the DLL loading and unloading would add a lot of overhead.

The Dynamically Linked Library Entry/Exit Function

You can provide optional entry and exit code for your DLLs when required under various initialization and shutdown operations. These operations can occur during process or thread initialization/termination.

Process/Thread Initialization and Termination Routines

Typical initialization operations include registering Windows classes, initializing global variables, and initializing an entry/exit function. This occurs during the method of entry for the DLL, which is referred to as the DLLEntryPoint function. This function is actually represented by the begin..end block of the DLL project file. This is the location where you would set up an entry/exit procedure. This procedure must take a single parameter of the type DWord.

The global DLLProc variable is a procedural pointer to which you can assign the entry/exit procedure. This variable is initially nil unless you set up your own procedure. By setting up an entry/exit procedure, you can respond to the events listed in Table 9.1.

TABLE 9.1 DLL Entry/Exit Events

Event	*Purpose*
DLL_PROCESS_ATTACH	The DLL is attaching to the address space of the current process when the process starts up or when a call to LoadLibrary() is made. DLLs initialize any instance data during this event.

Event	*Purpose*
`DLL_PROCESS_DETACH`	The DLL is detaching from the address space of the calling process. This occurs during a clean process exit or when a call to `FreeLibrary()` is made. The DLL can uninitialize any instance data during this event.
`DLL_THREAD_ATTACH`	This event occurs when the current process creates a new thread. When this occurs, the system calls the entry-point function of any DLLs attached to the process. This call is made in the context of the new thread and can be used to allocate any thread-specific data.
`DLL_THREAD_DETACH`	This event occurs when the thread is exiting. During this event, the DLL can free any thread-specific initialized data.

CAUTION

Threads terminated abnormally—by calling `TerminateThread()`—are not guaranteed to call `DLL_THREAD_DETACH`.

DLL Entry/Exit Example

Listing 9.7 illustrates how you would install an entry/exit procedure to the DLL's `DLLProc` variable.

LISTING 9.7 The Source Code for `DllEntry.dpr`

```
library DllEntry;
uses
  SysUtils,
  Windows,
  Dialogs,
  Classes;
procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
    DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
    DLL_THREAD_ATTACH:  MessageBeep(0);
    DLL_THREAD_DETACH:  MessageBeep(0);
  end;
```

continues

LISTING 9.7 Continued

```
end;
begin
  { First, assign the procedure to the DLLProc variable }
  DllProc := @DLLEntryPoint;
  { Now invoke the procedure to reflect that the DLL is attaching to the
    process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

The entry/exit procedure is assigned to the DLL's `DLLProc` variable in the `begin..end` block of the DLL project file. This procedure, `DLLEntryPoint()`, evaluates its `word` parameter to determine which event is being called. These events correspond to the events listed in Table 9.1. For illustration purposes, we have each event display a message box when the DLL is being loaded or destroyed. When a thread in the calling application is being created or destroyed, a message beep occurs.

To illustrate the use of this DLL, examine the code shown in Listing 9.8.

LISTING 9.8 Sample Code for DLL Entry/Exit Demo

```
unit MainFrm;

interface

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

type

  { Define a TThread descendant }

TTestThread = class(TThread)
    procedure Execute; override;
    procedure SetCaptionData;
  end;

  TMainForm = class(TForm)
    btnLoadLib: TButton;
    btnFreeLib: TButton;
    btnCreateThread: TButton;
```

```
    btnFreeThread: TButton;
    lblCount: TLabel;
    procedure btnLoadLibClick(Sender: TObject);
    procedure btnFreeLibClick(Sender: TObject);
    procedure btnCreateThreadClick(Sender: TObject);
    procedure btnFreeThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    LibHandle   : THandle;
    TestThread  : TTestThread;
    Counter     : Integer;
    GoThread    : Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TTestThread.Execute;
begin
  while MainForm.GoThread do
  begin
    Synchronize(SetCaptionData);
    Inc(MainForm.Counter);
  end;
end;

procedure TTestThread.SetCaptionData;
begin
  MainForm.lblCount.Caption := IntToStr(MainForm.Counter);
end;

procedure TMainForm.btnLoadLibClick(Sender: TObject);
{ This procedure loads the library DllEntryLib.DLL }
begin
  if LibHandle = 0 then
  begin
    LibHandle := LoadLibrary('DLLENTRYLIB.DLL');
    if LibHandle = 0 then
```

continues

LISTING 9.8 Continued

```
      raise Exception.Create('Unable to Load DLL');
  end
  else
    MessageDlg('Library already loaded', mtWarning, [mbok], 0);
end;

procedure TMainForm.btnFreeLibClick(Sender: TObject);
{ This procedure frees the library }
begin
  if not (LibHandle = 0) then
  begin
    FreeLibrary(LibHandle);
    LibHandle := 0;
  end;
end;

procedure TMainForm.btnCreateThreadClick(Sender: TObject);
{ This procedure creates the TThread instance. If the DLL is loaded a
  message beep will occur. }
begin
  if TestThread = nil then
  begin
    GoThread   := True;
    TestThread := TTestThread.Create(False);
  end;
end;

procedure TMainForm.btnFreeThreadClick(Sender: TObject);
{ In freeing the TThread a message beep will occur if the DLL is loaded. }
begin
  if not (TestThread = nil) then
  begin
    GoThread   := False;
    TestThread.Free;
    TestThread := nil;
    Counter    := 0;
  end;

end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  LibHandle  := 0;
  TestThread := nil;
end;

end.
```

This project consists of a main form with four `TButton` components. `BtnLoadLib` loads the DLL `DllEntryLib.dll`. `BtnFreeLib` frees the library from the process. `BtnCreateThread` creates a `TThread` descendant object, which in turn creates a thread. `BtnFreeThread` destroys the `TThread` object. The `lblCount` is used just to show the thread execution.

The `btnLoadLibClick()` event handler calls `LoadLibrary()` to load `DllEntryLib.dll`. This causes the DLL to load and be mapped to the process's address space. Additionally, the initialization code in the DLL gets executed. Again, this is the code that appears in the `begin..end` block of the DLL, which performs the following to set up an entry/exit procedure for the DLL:

```
begin
  { First, assign the procedure to the DLLProc variable }
  DllProc := @DLLEntryPoint;
  { Now invoke the procedure to reflect that the DLL is attaching to the
    process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

This initialization section will only be called once per process. If another process loads this DLL, this section will be called again, except in the context of the separate process—processes don't share DLL instances.

The `btnFreeLibClick()` event handler unloads the DLL by calling `FreeLibrary()`. When this happens, the procedure to which the `DLLProc` points, `DLLEntryProc()`, gets called with the value of `DLL_PROCESS_DETACH` passed as the parameter.

The `btnCreateThreadClick()` event handler creates the `TThread` descendant object. This causes the `DLLEntryProc()` to get called, and the `DLL_THREAD_ATTACH` value is passed as the parameter. The `btnFreeThreadClick()` event handler invokes `DLLEntryProc` again but passes `DLL_THREAD_DETACH` as the value to the procedure.

Although you invoke only a message box when the events occur, you'll use these events to perform any process or thread initialization or cleanup that might be necessary for your application. Later, you'll see an example of using this technique to set up sharable DLL global data. You can look at the demo of this DLL in the project `DLLEntryTest.dpr` on the CD.

Exceptions in DLLs

This section discusses issues regarding DLLs and Win32 exceptions.

Capturing Exceptions in 16-Bit Delphi

Back in the 16-bit days with Delphi 1, Delphi exceptions were language specific. Therefore, if exceptions were raised in a DLL, you were required to capture the exception before it escaped from the DLL so that it wouldn't creep up the calling modules stack, causing it to crash. You had to wrap every DLL entry point with an exception handler, like this:

```
procedure SomeDLLProc;
begin
  try
    { Do your stuff }
  except
     on Exception do
        { Don't let it get away, handle it and don't re-raise it }
  end;
end;
```

This is no longer the case as of Delphi 2. Delphi 5 exceptions map themselves to Win32 exceptions. Exceptions raised in DLLs are no longer a compiler/language feature of Delphi but rather a feature of the Win32 system.

For this to work, however, you must make sure that `SysUtils` is included in the DLL's `uses` clause. Not including `SysUtils` disables Delphi's exception support inside the DLL.

CAUTION

Most Win32 applications are not designed to handle exceptions, so even though Delphi language exceptions get turned into Win32 exceptions, exceptions that you let escape from a DLL into the host application are likely to shut down the application.

If the host application is built with Delphi or C++Builder, this shouldn't be much of an issue, but there's still a lot of raw C and C++ code out there that doesn't like exceptions.

Therefore, to make your DLLs bulletproof, you might still consider using the 16-bit method of protecting DLL entry points with `try..except` blocks to capture exceptions raised in your DLLs.

NOTE

When a non-Delphi application uses a DLL written in Delphi, it won't be able to utilize the Delphi language-specific exception classes. However, it can be handled as a Win32 system exception given the exception code of `$0EEDFACE`. The exception address will be the first entry in the `ExceptionInformation` array of the Win32 system `EXCEPTION_RECORD`. The second entry contains a reference to the Delphi exception object. Look up `EXCEPTION_RECORD` in the Delphi online help for additional information.

Exceptions and the Safecall Directive

`Safecall` functions are used for COM and exception handling. They guarantee that any exception will propagate to the caller of the function. A `Safecall` function converts an exception into an `HResult` return value. `Safecall` also implies the `StdCall` calling convention. Therefore, a `Safecall` function declared as

```
function Foo(i: integer): string; Safecall;
```

really looks like this according to the compiler:

```
function Foo(i: integer): string; HResult; StdCall;
```

The compiler then inserts an implicit `try..except` block that wraps the entire function contents and catches any exceptions raised. The `except` block invokes a call to `SafecallExceptionHandler()` to convert the exception into an `HResult`. This is somewhat similar to the 16-bit method of capturing exceptions and passing back error values.

Callback Functions

A *callback function* is a function in your application called by Win32 DLLs or other DLLs. Basically, Windows has several API functions that require a callback function. When calling these functions, you pass in an address of a function defined by your application that Windows can call. If you're wondering how this all relates to DLLs, remember that the Win32 API is really several routines exported from system DLLs. Essentially, when you pass a callback function to a Win32 function, you're passing this function to a DLL.

One such function is the `EnumWindows()` API function, which enumerates through all top-level windows. This function passes the handle of each window in the enumeration to your application-defined callback function. You're required to define and pass the callback function's address to the `EnumWindows()` function. The callback function that you must provide to `EnumWindows()` is defined this way:

```
function EnumWindowsProc(Hw: HWnd; lp: lParam): Boolean; stdcall;
```

We illustrate the use of the `EnumWindows()` function in the `CallBack.dpr` project on the CD accompanying this book and shown in Listing 9.9.

LISTING 9.9 `MainForm.pas`, Source to Callback Example

```
unit MainFrm;

interface

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

type

  { Define a record/class to hold the window name and class name for
    each window. Instances of this class will get added to ListBox1 }
  TWindowInfo = class
    WindowName,          // The window name
    WindowClass: String; // The window's class name
  end;

  TMainForm = class(TForm)
    lbWinInfo: TListBox;
    btnGetWinInfo: TButton;
    hdWinInfo: THeaderControl;
    procedure btnGetWinInfoClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure lbWinInfoDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure hdWinInfoSectionResize(HeaderControl: THeaderControl;
      Section: THeaderSection);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
function EnumWindowsProc(Hw: HWnd; AMainForm: TMainForm):
   Boolean; stdcall;
```

```
{ This procedure is called by the User32.DLL library as it enumerates
  through windows active in the system. }
var
  WinName, CName: array[0..144] of char;
  WindowInfo: TWindowInfo;
begin
  { Return true by default which indicates not to stop enumerating
    through the windows }
  Result := True;
  GetWindowText(Hw, WinName, 144); // Obtain the current window text
  GetClassName(Hw, CName, 144);    // Obtain the class name of the window
  { Create a TWindowInfo instance and set its fields with the values of
    the window name and window class name. Then add this object to
    ListBox1's Objects array. These values will be displayed later by
    the listbox }
  WindowInfo := TWindowInfo.Create;
  with WindowInfo do
  begin
    SetLength(WindowName, strlen(WinName));
    SetLength(WindowClass, StrLen(CName));
    WindowName := StrPas(WinName);
    WindowClass := StrPas(CName);
  end;
  // Add to Objects array
  MainForm.lbWinInfo.Items.AddObject('', WindowInfo); end;

procedure TMainForm.btnGetWinInfoClick(Sender: TObject);
begin
  { Enumerate through all top-level windows being displayed. Pass in the
    call back function EnumWindowsProc which will be called for each
    window }
  EnumWindows(@EnumWindowsProc, 0);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  { Free all instances of TWindowInfo }
  for i := 0 to lbWinInfo.Items.Count - 1 do
    TWindowInfo(lbWinInfo.Items.Objects[i]).Free
end;
```

9
DYNAMIC LINK LIBRARIES

continues

LISTING 9.9 Continued

```
procedure TMainForm.lbWinInfoDrawItem(Control: TWinControl;
  Index: Integer;Rect: TRect; State: TOwnerDrawState);
begin
  { First, clear the rectangle to which drawing will be performed }
  lbWinInfo.Canvas.FillRect(Rect);
  { Now draw the strings of the TWindowInfo record stored at the
    Index'th position of the listbox. The sections of HeaderControl
    will give positions to which to draw each string }
  with TWindowInfo(lbWinInfo.Items.Objects[Index]) do
  begin
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowName),
      Length(WindowName), Rect,dt_Left or dt_VCenter);
    { Shift the drawing rectangle over by using the size
      HeaderControl1's sections to determine where to draw the next
      string }
    Rect.Left := Rect.Left + hdWinInfo.Sections[0].Width;
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowClass),
      Length(WindowClass), Rect, dt_Left or dt_VCenter);
  end;
end;

procedure TMainForm.hdWinInfoSectionResize(HeaderControl:
   THeaderControl; Section: THeaderSection);
begin
  lbWinInfo.Invalidate; // Force ListBox1 to redraw itself.
end;

end.
```

This application uses the `EnumWindows()` function to extract the window name and class name of all top-level windows and adds them to the owner-draw list box on the main form. The main form uses an owner-draw list box to make both the window name and window class name appear in a columnar fashion. First we'll explain the use of the callback function. Then we'll explain how we created the columnar list box.

Using the Callback Function

You saw in Listing 9.9 that we defined a procedure, `EnumWindowsProc()`, that takes a window handle as its first parameter. The second parameter is user-defined data, so you may pass whatever data you deem necessary as long as its size is the equivalent to an integer data type.

EnumWindowsProc() is the callback procedure that you'll pass to the EnumWindows() Win32 API function. It must be declared with the StdCall directive to specify that it uses the Win32 calling convention. When passing this procedure to EnumWindows(), it will get called for each top-level window whose window handle gets passed as the first parameter. You use this window handle to obtain both the window name and class name of each window. You then create an instance of the TWindowInfo class and set its fields with this information. The TWindowInfo class instance is then added to the lbWinInfo.Objects array. The data in this list box will be used when the list box is drawn to show this data in a columnar fashion.

Notice that, in the main form's OnDestroy event handler, you make sure to clean up any allocated instances of the TWindowInfo class.

The btnGetWinInfoClick()event handler calls the EnumWindows() procedure and passes EnumWindowsProc() as its first parameter.

When you run the application and click the button, you'll see that the information is obtained from each window and is shown in the list box.

Drawing an Owner-Draw List Box

The window names and class names of top-level windows are drawn in a columnar fashion in lbWinInfo from the previous project. This was done by using a TListBox with its Style property set to lbOwnerDraw. When this style is set as such, the TListBox.OnDrawItem event is called each time the TListBox is to draw one of its items. You're responsible for drawing the items as illustrated in the example.

In Listing 9.9, the event handler lbWinInfoDrawItem() contains the code that performs the drawing of list box items. Here, you draw the strings contained in the TWindowInfo class instances, which are stored in the lbWinInfo.Objects array. These values are obtained from the callback function EnumWindowsProc(). You can refer to the code commentary to determine what this event handler does.

Calling Callback Functions from Your DLLs

Just as you can pass callback functions to DLLs, you can also have your DLLs call callback functions. This section illustrates how you can create a DLL whose exported function takes a callback procedure as a parameter. Then, based on whether the user passes in a callback procedure, the procedure gets called. Listing 9.10 contains the source code to this DLL.

LISTING 9.10 Calling a Callback Demo: Source Code for StrSrchLib.dll

```
library StrSrchLib;

uses
  Wintypes,
  WinProcs,
```

continues

LISTING 9.10 Continued

```
  SysUtils,
  Dialogs;

type
 { declare the callback function type }
 TFoundStrProc = procedure(StrPos: PChar); StdCall;

function SearchStr(ASrcStr, ASearchStr: PChar;  AProc: TFarProc):
  Integer; StdCall;
{ This function looks for ASearchStr in ASrcStr. When founc ASearchStr,
  the callback procedure referred to by AProc is called if one has been
  passed in. The user may pass nil as this parameter. }
var
  FindStr: PChar;
begin
  FindStr := ASrcStr;
  FindStr := StrPos(FindStr, ASearchStr);
  while FindStr <> nil do
  begin
    if AProc <> nil then
      TFoundStrProc(AProc)(FindStr);
    FindStr := FindStr + 1;
    FindStr := StrPos(FindStr, ASearchStr);
  end;
end;

exports
  SearchStr;
begin

end.
```

The DLL also defines a procedural type, `TFoundStrProc`, for the callback function, which will be used to typecast the callback function when it's called.

The exported procedure `SearchStr()` is where the callback function is called. The commentary in the listing explains what this procedure does.

An example of this DLL's usage is given in the project `CallBackDemo.dpr` in the `\DLLCallBack` directory on the CD. The source for the main form of this demo is shown in Listing 9.11.

LISTING 9.11 The Main Form for the DLL Callback Demo

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    btnCallDLLFunc: TButton;
    edtSearchStr: TEdit;
    lblSrchWrd: TLabel;
    memStr: TMemo;
    procedure btnCallDLLFuncClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  Count: Integer;

implementation

{$R *.DFM}

{ Define the DLL's exported procedure }
function SearchStr(ASrcStr, ASearchStr: PChar; AProc: TFarProc):
   Integer; StdCall external
  'STRSRCHLIB.DLL';

{ Define the callback procedure, make sure to use the StdCall directive }
procedure StrPosProc(AStrPsn: PChar); StdCall;
begin
  inc(Count); // Increment the Count variable.
end;

procedure TMainForm.btnCallDLLFuncClick(Sender: TObject);
var
  S: String;
  S2: String;
begin
```

continues

LISTING 9.11 Continued

```
  Count := 0; // Initialize Count to zero.
  { Retrieve the length of the text on which to search. }
  SetLength(S, memStr.GetTextLen);
  { Now copy the text to the variable S }
  memStr.GetTextBuf(PChar(S), memStr.GetTextLen);
  { Copy Edit1's Text to a string variable so that it can be passed to
    the DLL function }
  S2 := edtSearchStr.Text;
  { Call the DLL function }
  SearchStr(PChar(S), PChar(S2), @StrPosProc);
  { Show how many times the word occurs in the string. This has been
    stored in the Count variable which is used by the callback function }
  ShowMessage(Format('%s %s %d %s', [edtSearchStr.Text,
    'occurs', Count, 'times.']));
end;

end.
```

This application contains a `TMemo` control. `EdtSearchStr.Text` contains a string that will be searched for in `memStr`'s contents. `memStr`'s contents are passed as the source string to the DLL function `SearchStr()`, and `edtSearchStr.Text` is passed as the search string.

The function `StrPosProc()` is the actual callback function. This function increments the value of the global variable `Count`, which you use to hold the number of times the search string occurs in `memStr`'s text.

Sharing DLL Data Across Different Processes

Back in the world of 16-bit Windows, DLL memory was handled differently than it is in the 32-bit world of Win32. One often-used trait of 16-bit DLLs is that they share global memory among different applications. In other words, if you declare a global variable in a 16-bit DLL, any application that uses that DLL will have access to that variable, and changes made to that variable by an application will be seen by other applications.

In some ways, this behavior can be dangerous because one application can overwrite data on which another application is dependent. In other ways, developers have made use of this characteristic.

In Win32, this sharing of DLL global data no longer exists. Because each application process maps the DLL to its own address space, the DLL's data also gets mapped to that same address space. This results in each application getting its own instance of DLL data. Changes made to the DLL global data by one application won't be seen from another application.

If you're planning on porting a 16-bit application that relies on the sharable behavior of DLL global data, you can still provide a means for applications to share data in a DLL with other applications. The process isn't automatic, and it requires the use of memory-mapped files to store the shared data. Memory-mapped files are covered in Chapter 12, "Working with Files." We'll use them here to illustrate this method; however, you'll probably want to return to and review this section when you have a more thorough understanding of memory-mapped files after reading Chapter 12.

Creating a DLL with Shared Memory

Listing 9.12 shows a DLL project file that contains the code to allow applications using this DLL to share its global data. This global data is stored in the variable appropriately named `GlobalData`.

LISTING 9.12 `ShareLib`: A DLL That Illustrates Sharing Global Data

```
library ShareLib;

uses
  ShareMem,
  Windows,
  SysUtils,
  Classes;
const

  cMMFileName: PChar = 'SharedMapData';

{$I DLLDATA.INC}

var
  GlobalData : PGlobalDLLData;
  MapHandle  : THandle;

{ GetDLLData will be the exported DLL function }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall;
begin
  { Point AGlobalData to the same memory address referred to by GlobalData. }
  AGlobalData := GlobalData;
end;

procedure OpenSharedData;
var
   Size: Integer;
```

continues

LISTING 9.12 Continued

```
begin
  { Get the size of the data to be mapped. }
  Size := SizeOf(TGlobalDLLData);

  { Now get a memory-mapped file object. Note the first parameter passes
    the value $FFFFFFFF or DWord(-1) so that space is allocated from
    the system's paging file. This requires that a name for the memory-mapped
    object get passed as the last parameter. }

  MapHandle := CreateFileMapping(DWord(-1), nil, PAGE_READWRITE, 0,
    Size, cMMFileName);

  if MapHandle = 0 then
    RaiseLastWin32Error;
  { Now map the data to the calling process's address space and get a
    pointer to the beginning of this address }
  GlobalData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, Size);
  { Initialize this data }
  GlobalData^.S := 'ShareLib';
  GlobalData^.I := 1;
  if GlobalData = nil then
  begin
    CloseHandle(MapHandle);
    RaiseLastWin32Error;
  end;
end;

procedure CloseSharedData;
{ This procedure un-maps the memory-mapped file and releases the memory-mapped
  file handle }
begin
  UnmapViewOfFile(GlobalData);
  CloseHandle(MapHandle);
end;

procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: OpenSharedData;
    DLL_PROCESS_DETACH: CloseSharedData;
  end;
```

```
end;

exports
  GetDLLData;

begin
  { First, assign the procedure to the DLLProc variable }
  DllProc := @DLLEntryPoint;
  { Now invoke the procedure to reflect that the DLL is attaching
    to the process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

`GlobalData` is of the type `PGlobalDLLData`, which is defined in the include file `DllData.inc`. This include file contains the following type definition (note that the include file is linked by using the include directive `$I`):

```
type

  PGlobalDLLData = ^TGlobalDLLData;
  TGlobalDLLData = record
    S: String[50];
    I: Integer;
  end;
```

In this DLL, you use the same process discussed earlier in the chapter to add entry and exit code to the DLL in the form of an entry/exit procedure. This procedure is called `DLLEntryPoint()`, as shown in the listing. When a process loads the DLL, the `OpenSharedData()` method gets called. When a process detaches from the DLL, the `CloseSharedData()` method is called.

We won't go into too much detail here about memory-mapped file usage because we cover it in more detail in Chapter 12, "Working with Files." However, we'll explain the basics of what's going on so that you understand the purpose of this DLL.

Memory-mapped files provide a means for you to reserve a region of address space in the Win32 system to which physical storage gets committed. This is similar to allocating memory and referring to that memory with a pointer. With memory-mapped files, however, you can map a disk file to this address space and refer to the space within the file as though you were just referencing an area of memory with a pointer.

With memory-mapped files, you must first get a handle to an existing file on disk to which a memory-mapped object will be mapped. You then map the memory-mapping object to that file. At the beginning of the chapter, we told you how the system shares DLLs with multiple applications by first loading the DLL into memory and then giving each application its own image of the DLL so that it appears that each application has loaded a separate instance of the DLL.

In reality, however, the DLL exists in memory only once. This is done by using memory-mapped files. You can use the same process to give access to data files. You just make necessary Win32 API calls that deal with creating and accessing memory-mapped files.

Now, consider this scenario: Suppose an application, which we'll call `App1`, creates a memory-mapped file that gets mapped to a file on disk, `MyFile.dat`. `App1` can now read and write data in that file. If, while `App1` is running, `App2` also maps to that same file, changes made to the file by `App1` will be seen by `App2`. Actually, it's a bit more complex; certain flags must be set so that changes to the file are immediately set and so forth. For this discussion, it suffices to say that changes will be realized by both applications because this is possible.

One of the ways in which memory-mapped files can be used is to create a file mapping from the Win32 paging file rather than an existing file. This means that instead of mapping to an existing file on disk, you can reserve an area of memory to which you can refer as though it were a disk file. This prevents you from having to create and destroy a temporary file if all you want to do is to create an address space that can be accessed by multiple processes. The Win32 system manages its paging file, so when memory is no longer required of the paging file, this memory gets released.

In the preceding paragraphs, we presented a scenario that illustrated how two applications can access the same file data by using a memory-mapped file. The same can be done between an application and a DLL. In fact, if the DLL creates the memory-mapped file when it's loaded by an application, it will use the same memory-mapped file when loaded by another application. There will be two images of the DLL, one for each calling application, both of which use the same memory-mapped file instance. The DLL can make the data referred to by the file mapping available to its calling application. When one application makes changes to this data, the second application will see these changes because they're referring to the same data, mapped by two different memory-mapped object instances. We use this technique in the example.

In Listing 9.12, `OpenSharedData()` is responsible for creating the memory-mapped file. It uses the `CreateFileMapping()` function to first create the file-mapping object, which it then passes to the `MapViewOfFile()` function. The `MapViewOfFile()` function maps a view of the file into the address space of the calling process. The return value of this function is the beginning of that address space. Now remember, this is the address space of the calling process. For two different applications using this DLL, this address location might be different, although the data to which they refer will be the same.

NOTE

The first parameter to `CreateFileMapping()` is a handle to a file to which the memory-mapped file gets mapped. However, if you're mapping to an address space of the system paging file, pass the value `$FFFFFFFF` (which is the same as `DWord(-1)`) as this parameter value. You must also supply a name for the file-mapping object as the last

parameter to `CreateFileMapping()`. This is the name that the system uses to refer to this file mapping. If multiple processes create a memory-mapped file using the same name, the mapping objects will refer to the same system memory.

After the call to `MapViewOfFile()`, the variable `GlobalData` refers to the address space for the memory-mapped file. The exported function `GetDLLData()` assigns that memory to which `GlobalData` refers to the `AGlobalData` parameter. `AGlobalData` is passed in from the calling application; therefore, the calling application has read/write access to this data.

The `CloseSharedData()` procedure is responsible for unmapping the view of the file from the calling process and releasing the file-mapping object. This doesn't affect other file-mapping objects or file mappings from other applications.

Using a DLL with Shared Memory

To illustrate the use of the shared memory DLL, we've created two applications that make use of it. The first application, `App1.dpr`, allows you to modify the DLL's data. The second application, `App2.dpr`, also refers to the DLL's data and continually updates a couple of `TLabel` components by using a `TTimer` component. When you run both applications, you'll be able to see the sharable access to the DLL data—`App2` will reflect changes made by `App1`.

Listing 9.13 shows the source code for the `APP1` project.

LISTING 9.13 The Main Form for `App1.dpr`

```
unit MainFrmA1;

interface

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

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    edtGlobDataStr: TEdit;
    btnGetDllData: TButton;
    meGlobDataInt: TMaskEdit;
    procedure btnGetDllDataClick(Sender: TObject);
```

continues

LISTING 9.13 Continued

```
    procedure edtGlobDataStrChange(Sender: TObject);
    procedure meGlobDataIntChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

var
  MainForm: TMainForm;

{ Define the DLL's exported procedure }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
  StdCall External 'SHARELIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnGetDllDataClick(Sender: TObject);
begin
  { Get a pointer to the DLL's data }
  GetDLLData(GlobalData);
  { Now update the controls to reflect GlobalData's field values }
  edtGlobDataStr.Text := GlobalData^.S;
  meGlobDataInt.Text  := IntToStr(GlobalData^.I);
end;

procedure TMainForm.edtGlobDataStrChange(Sender: TObject);
begin
  { Update the DLL data with the changes }
  GlobalData^.S := edtGlobDataStr.Text;
end;

procedure TMainForm.meGlobDataIntChange(Sender: TObject);
begin
  { Update the DLL data with the changes }
  if meGlobDataInt.Text = EmptyStr then
    meGlobDataInt.Text := '0';
  GlobalData^.I := StrToInt(meGlobDataInt.Text);
end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  btnGetDllDataClick(nil);
end;

end.
```

This application also links in the include file `DllData.inc`, which defines the `TGlobalDLLData` data type and its pointer. The `btnGetDllDataClick()` event handler gets a pointer to the DLL's data, which is accessed by a memory-mapped file in the DLL. It does this by calling the DLL's `GetDLLData()` function. It then updates its controls with the value of this pointer, `GlobalData`. The `OnChange` event handlers for the edit controls change the values of `GlobalData`. Because `GlobalData` refers to the DLL's data, it modifies the data referred to by the DLL's memory-mapped file.

Listing 9.14 shows the source code for the main form for `App2.dpr`.

LISTING 9.14 The Source Code for Main Form for `App2.dpr`

```
unit MainFrmA2;

interface

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

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    lblGlobDataStr: TLabel;
    tmTimer: TTimer;
    lblGlobDataInt: TLabel;
    procedure tmTimerTimer(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

{ Define the DLL's exported procedure }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
```

continues

LISTING 9.14 Continued

```
  StdCall External 'SHARELIB.DLL';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.tmTimerTimer(Sender: TObject);
begin
  GetDllData(GlobalData);  // Get access to the data
  { Show the contents of GlobalData's fields.}
  lblGlobDataStr.Caption := GlobalData^.S;
  lblGlobDataInt.Caption := IntToStr(GlobalData^.I);
end;

end.
```

This form contains two `TLabel` components, which get updated during the `tmTimer`'s `OnTimer` event. When the user changes the values of the DLL's data from `App1`, `App2` will reflect these changes.

You can run both applications to experiment with them. You'll find them on this book's CD.

Exporting Objects from DLLs

It's possible to access an object and its methods even if that object is contained within a DLL. There are some requirements, however, to how that object is defined within the DLL as well as some limitations as to how the object can be used. The technique we illustrate here is useful in very specific situations. Typically, you can achieve the same functionality by using packages or interfaces.

The following list summarizes the conditions and limitations to exporting an object from a DLL:

- The calling application can only use methods of the object that have been declared as virtual.
- The object instances must be created only within the DLL.
- The object must be defined in both the DLL and calling application with methods defined in the same order.
- You cannot create a descendant object from the object contained within the DLL.

Some additional limitations might exist, but the ones listed are the primary limitations.

To illustrate this technique, we've created a simple, yet illustrative example of an object that we export. This object contains a function that returns the uppercase or lowercase value of a string based on the value of a parameter indicating either uppercase or lowercase. This object is defined in Listing 9.15.

LISTING 9.15 Object to Be Exported from a DLL

```
type

  TConvertType = (ctUpper, ctLower);

  TStringConvert = class(TObject)
{$IFDEF STRINGCONVERTLIB}
  private
    FPrepend: String;
    FAppend : String;
{$ENDIF}
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
String;
      virtual; stdcall; {$IFNDEF STRINGCONVERTLIB} abstract; {$ENDIF}
{$IFDEF STRINGCONVERTLIB}
    constructor Create(APrepend, AAppend: String);
    destructor Destroy; override;
{$ENDIF}
  end;

{ For any application using this class, STRINGCONVERTLIB is not defined and
  therefore, the class definition will be equivalent to:

  TStringConvert = class(TObject)
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
String;
      virtual; stdcall; abstract;
  end;
}
```

Listing 9.15 is actually an include file named `StrConvert.inc`. The reason for placing this object in an include file is to meet the third requirement in the preceding list—that the object be equally defined in both the DLL and in the calling application. By placing the object in an

include file, both the calling application and DLL can include this file. If changes are made to the object, you only have to compile both projects instead of typing the changes twice—once in the calling application and once in the DLL—which is error prone.

Observe the following definition of the `ConvertSring()` method:

```
function ConvertString(AConvertType: TConvertType; AString: String):
   ➥String; virtual; stdcall;
```

The reason you declare this method as virtual is not so that one can create a descendant object that can then override the `ConvertString()` method. Instead, it's declared as virtual so that an entry to the `ConvertString()` method is made in the Virtual Method Table (VMT). We won't go into detail on the VMT here; it's discussed in Chapter 13, "Hard-Core Techniques." For now, think of the VMT as a block of memory that holds pointers to virtual methods of an object. Because of the VMT, the calling application can obtain a pointer to the method of the object. Without declaring the method as virtual, the VMT would not have an entry for the method, and the calling application would have no way of obtaining the pointer to the method. So really, what you have in the calling application is a pointer to the function. Because you've based this pointer on a method type defined in an object, Delphi automatically handles any fix-ups, such as passing the implicit `self` parameter to the method.

Note the conditional define `STRINGCONVERTLIB`. When you're exporting the object, the only methods that need redefinition in the calling application are the methods to be accessed externally from the DLL. Also, these methods can be defined as abstract methods to avoid generating a compile-time error. This is valid because at runtime, these methods will be implemented in the DLL code. The commentary shows what the `TStringConvert` object looks like on the application side.

Listing 9.16 shows the implementation of the `TStringConvert` object.

LISTING 9.16 Implementation of the `TStringConvert` Object

```
unit StringConvertImp;
{$DEFINE STRINGCONVERTLIB}

interface
uses SysUtils;
{$I StrConvert.inc}

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;

implementation

constructor TStringConvert.Create(APrepend, AAppend: String);
begin
```

```
  inherited Create;
  FPrepend := APrepend;
  FAppend  := AAppend;
end;

destructor TStringConvert.Destroy;
begin
  inherited Destroy;
end;

function TStringConvert.ConvertString(AConvertType:
  TConvertType; AString: String): String;
begin
  case AConvertType of
    ctUpper: Result := Format('%s%s%s', [FPrepend, UpperCase(AString),
    FAppend]);
    ctLower: Result := Format('%s%s%s', [FPrepend, LowerCase(AString),
    FAppend]);
  end;
end;

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
begin
  Result := TStringConvert.Create(APrepend, AAppend);
end;

end.
```

As stated in the conditions, the object must be created in the DLL. This is done in a standard DLL exported function `InitStrConvert()`, which takes two parameters that are passed to the constructor. We added this to illustrate how you would pass information to an object's constructor through an interface function.

Also, notice that in this unit you declare the conditional directive `STRINGCONVERTLIB`. The rest of this unit is self-explanatory. Listing 9.17 shows the DLL's project file.

LISTING 9.17 The Project File for `StringConvertLib.dll`

```
library StringConvertLib;
uses
  ShareMem,
  SysUtils,
  Classes,
```

continues

LISTING 9.17 Continued

```
  StringConvertImp in 'StringConvertImp.pas';

exports
  InitStrConvert;
end.
```

Generally, this library doesn't contain anything we haven't already covered. Do note, however, that you used the `ShareMem` unit. This unit must be the first unit declared in the library project file as well as in the calling application's project file. This is an extremely important thing to remember.

Listing 9.18 shows an example of how to use the exported object to convert a string to both uppercase and lowercase. You'll find this demo project on the CD as `StrConvertTest.dpr`.

LISTING 9.18 The Demo Project for the String Conversion Object

```
unit MainFrm;

interface

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

{$I strconvert.inc}

type


  TMainForm = class(TForm)
    btnUpper: TButton;
    edtConvertStr: TEdit;
    btnLower: TButton;
    procedure btnUpperClick(Sender: TObject);
    procedure btnLowerClick(Sender: TObject);
  private
  public
  end;
```

```
var
  MainForm: TMainForm;

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;
  external 'STRINGCONVERTLIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnUpperClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Upper ', ' end');
  try
      ConvStr := edtConvertStr.Text;
      if ConvStr <> EmptyStr then
        edtConvertStr.Text := FStrConvert.ConvertString(ctUpper, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

procedure TMainForm.btnLowerClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Lower ', ' end');
  try
      ConvStr := edtConvertStr.Text;
      if ConvStr <> EmptyStr then
        edtConvertStr.Text := FStrConvert.ConvertString(ctLower, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

end.
```

Summary

DLLs are an essential part of creating Windows applications while focusing in on code reusability. This chapter covered the reasons for creating or using DLLs. The chapter illustrated how to create and use DLLs in your Delphi applications and showed different methods of loading DLLs. The chapter discussed some of the special considerations you must take when using DLLs with Delphi and showed you how to make DLL data sharable with different applications.

With this knowledge under your belt, you should be able to create DLLs with Delphi and use them in your Delphi applications with ease. You'll learn more about DLLs in other chapters.