Hard-Core Techniques

CHAPTER

13

IN THIS CHAPTER

There comes a time when you must step off the beaten path to accomplish a particular goal. This chapter teaches you some advanced techniques you can use in your Delphi applications. You get much closer to the Win32 API in this chapter than you do in most of the other chapters, and you explore some things that aren't obvious or aren't provided under the Visual Component Library (VCL). You learn about concepts such as window procedures, multiple program instances, Windows hooks, and sharing Delphi and C++ code.

Advanced Application Message Handling

As discussed in Chapter 5, "Understanding Windows Messaging," a *window procedure* is a function that Windows calls whenever a particular window receives a message. Because the `Application` object contains a window, it has a window procedure that's called to receive all the messages sent to your application. The `TApplication` class even comes equipped with an `OnMessage` event that notifies you whenever one of these messages comes down the pike.

Well…not exactly.

`TApplication.OnMessage` fires only when a message is retrieved from the application's message queue (again, refer to Chapter 5, for a discussion of all this message terminology). Messages found in the application queue are typically those dealing with window management (`WM_PAINT` and `WM_SIZE`, for example) and those posted to the window by using an API function such as `PostMessage()`, `PostAppMessage()`, or `BroadcastSystemMessage()`. The problem arises when other types of messages are sent directly to the window procedure by Windows or by the `SendMessage()` function. When this occurs, the `TApplication.OnMessage` event never happens, and there's no way to know whether the message occurred based on this event.

Subclassing

To know when a message is sent to your application, you must replace the `Application` window's procedure with your own. In your window procedure, you should do whatever processing or message handling you need to do before passing the message to the original window procedure. This process is known as *subclassing* a window.

You can use the `SetWindowLong()` Win32 API function with the `GWL_WNDPROC` constant to set a new window procedure function for a window. The window procedure function itself can have one of two formats: It can follow the API definition of a window procedure, or you can take advantage of some Delphi helper functions and make the window procedure a special method referred to as a *window method*.

CAUTION

A problem that can arise when you subclass the window procedure of a VCL window is that the handle of the window can be re-created beneath you, thus causing your application to fail. Beware of using this technique if there's a chance the window handle of the window you're subclassing will be re-created. A safer technique is to use `Application.HookMainWindow()`, which is shown later in this chapter.

A Win32 API Window Procedure

An API window procedure must have the following declaration:

```
function AWndProc(Handle: hWnd; Msg, wParam, lParam: Longint):
    Longint; stdcall;
```

The `Handle` parameter identifies the destination window, the `Msg` parameter is the window message, and the `wParam` and `lParam` parameters contain additional message-specific information. This function returns a value that depends on the message received. Note carefully that this function must use the `stdcall` calling convention.

You can use the `SetWindowLong()` function to set the window procedure of `Application`'s window, as shown here:

```
var
  WProc: Pointer;
begin
  WProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
   Integer(@NewWndProc)));
```

After this call, `WProc` will hold a pointer to the old window procedure. It's necessary to save this value because you must pass on any messages you don't handle yourself to the old window procedure using the `CallWindowProc()` API function. The following code gives you an idea of the implementation of the window procedure:

```
function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint):
    Longint; stdcall;
begin
  { Check value of Msg, and perform whatever type of action you'd }
  { like depending on the value of the message.  For messages you }
  {don't explicitly handle, you must pass the message information }
  {on to the original window procedure as shown below: }
  Result := CallWindowProc(WProc, Application.Handle, Msg, wParam,
    lParam);
end;
```

Listing 13.1 shows the ScWndPrc.pas unit, which subclasses Application's window procedure to handle a user-defined message called DDGM_FOOMSG.

LISTING 13.1 ScWndPrc.pas

```
unit ScWndPrc;

interface

uses Forms, Messages;

const
  DDGM_FOOMSG = WM_USER;

implementation

uses Windows, SysUtils, Dialogs;

var
  WProc: Pointer;

function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint): Longint;
  stdcall;
{ This is a Win32 API-level window procedure. It handles the messages }
{ received by the Application window. }
begin
  if Msg = DDGM_FOOMSG then
    { If it's our user-defined message, then alert the user. }
    ShowMessage(Format('Message seen by WndProc! Value is: $%x', [Msg]));
  { Pass message on to old window procedure }
  Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
end;

initialization
  { Set window procedure of Application window. }
  WProc := Pointer(SetWindowLong(Application.Handle, gwl_WndProc,
    Integer(@NewWndProc)));
end.
```

CAUTION

Be sure to save the old window procedure returned by GetWindowLong(). If you don't call the old window procedure inside your subclassed window procedure for messages that you don't want to handle, you're likely to crash your application, and you might even crash the operating system.

A Delphi Window Method

Delphi provides a function called `MakeObjectInstance()` that bridges the gap between an API window procedure and a Delphi method. `MakeObjectInstance()` enables you to create a method of type `TWndMethod` to serve as the window procedure. `MakeObjectInstance()` is declared in the `Forms` unit as follows:

```
function MakeObjectInstance(Method: TWndMethod): Pointer;
```

`TWndMethod` is defined in the `Forms` unit as follows:

```
type
  TWndMethod = procedure(var Message: TMessage) of object;
```

The return value of `MakeObjectInstance()` is a `Pointer` to the address of the newly created window procedure. This is the value you pass as the last parameter to `SetWindowLong()`. You should free any window methods created with `MakeObjectInstance()` by using the `FreeObjectInstance()` function.

As an illustration, the project called `WinProc.dpr` demonstrates both techniques for subclassing the `Application` window procedure and its advantages over `Application.OnMessage`. The main form for this project is shown in Figure 13.1.

FIGURE 13.1
WinProc's main form.

Listing 13.2 shows the source code for `Main.pas`, the main unit for the WinProc project.

LISTING 13.2 The Source Code for `Main.pas`

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    SendBtn: TButton;
    PostBtn: TButton;
```

continues

LISTING 13.2 Continued

```
    procedure SendBtnClick(Sender: TObject);
    procedure PostBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    OldWndProc: Pointer;
    WndProcPtr: Pointer;
    procedure WndMethod(var Msg: TMessage);
    procedure HandleAppMessage(var Msg: TMsg; var Handled: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses ScWndPrc;


procedure TMainForm.HandleAppMessage(var Msg: TMsg;
  var Handled: Boolean);
{ OnMessage handler for Application object. }
begin
  if Msg.Message = DDGM_FOOMSG then
    { if it's the user-defined message, then alert the user. }
    ShowMessage(Format('Message seen by OnMessage! Value is: $%x',
      [Msg.Message]));
end;

procedure TMainForm.WndMethod(var Msg: TMessage);
begin
  if Msg.Msg = DDGM_FOOMSG then
    { if it's the user-defined message, then alert the user. }
    ShowMessage(Format('Message seen by WndMethod! Value is: $%x',
      [Msg.Msg]));
  with Msg do
    { Pass message on to old window procedure. }
    Result := CallWindowProc(OldWndProc, Application.Handle, Msg, wParam,
      lParam);
end;

procedure TMainForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
```

```
end;

procedure TMainForm.PostBtnClick(Sender: TObject);
begin
  PostMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := HandleAppMessage;     // set OnMessage handler
  WndProcPtr := MakeObjectInstance(WndMethod);   // make window proc
  { Set window procedure of application window. }
  OldWndProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Integer(WndProcPtr)));
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  { Restore old window procedure for Application window }
  SetWindowLong(Application.Handle, GWL_WNDPROC, Longint(OldWndProc));
  { Free our user-created window procedure }
  FreeObjectInstance(WndProcPtr);
end;

end.
```


When `SendBtn` is clicked, the `SendMessage()` API function is used to send the message `DDGM_FOOMSG` to `Application`'s window handle. When `PostBtn` is clicked, the same message is posted to `Application` using the `PostMessage()` API function.

The `HandleAppMessage()` is assigned to handle the `Application.OnMessage` event. This procedure simply uses `ShowMessage()` to invoke a dialog box indicating that it sees a message. The `OnMessage` event is assigned in the `OnCreate` event handler for the main form.

Notice that the `OnDestroy` handler for the main form resets `Application`'s window procedure to the original value (`OldWndProc`) before calling `FreeObjectInstance()` to free the procedure created with `MakeProcInstance()`. If the old window procedure isn't first reinstated, the effect would be that of "unplugging" the window procedure from an active window—effectively removing the window's capability to handle messages. That's bad news because doing so could potentially crash the application or the OS.

Just for kicks, the `ScWndPrc` unit, shown earlier in this chapter, is included in `Main`. This means that the `Application` window will be subclassed twice: once by `ScWndPrc` using the API technique and once by `Main` using the window method technique. There's absolutely no danger in doing this as long as you remember to use `CallWindowProc()` in the window procedure and method to pass messages down to the old window procedures.

When you run this application, you'll be able to see that the `ShowMessage()` dialog box is shown from both the window procedure and method no matter which button is pushed. What's more, you'll see that `Application.OnMessage` sees only the messages posted to the window.

HookMainWindow()

Another perhaps more VCL-friendly technique for intercepting messages meant for the `Application` window is `TApplication`'s `HookMainWindow()` method. This method allows you to insert your own message handler at the top of `TApplication`'s `WndProc()` method to perform special message processing or prevent `TApplication` from processing certain messages. `HookMainWindow()` is defined as follows:

```
procedure HookMainWindow(Hook: TWindowHook);
```

The parameter for this method is of type `TWindowHook`, which is defined as this:

```
type
  TWindowHook = function (var Message: TMessage): Boolean of object;
```

There isn't much to using this method; just call `HookMainWindow()`, passing your own method in the `Hook` parameter. This adds your method to a list of window hook methods that will be called prior to the normal message processing that occurs in `TApplication.WndProc()`. If a window hook method returns `True`, the message is considered handled, and the `WndProc()` method will immediately exit.

When you're through processing messages, call the `UnhookMainWindow()` method to remove your method from the window hook method list. This method is similarly defined as follows:

```
procedure UnhookMainWindow(Hook: TWindowHook);
```

Listing 13.3 shows the main form for a simple one-form VCL project that employs this technique, and Figure 13.2 shows this application in action.

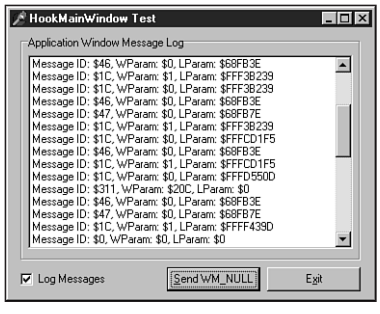

FIGURE 13.2

Spying on the `Application` with the HookWnd project.

LISTING 13.3 Main.pas for the HookWnd Project

```
unit HookMain;

interface

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

type
  THookForm = class(TForm)
    SendBtn: TButton;
    GroupBox1: TGroupBox;
    LogList: TListBox;
    DoLog: TCheckBox;
    ExitBtn: TButton;
    procedure SendBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
    function AppWindowHook(var Message: TMessage): Boolean;
  end;

var
  HookForm: THookForm;

implementation

{$R *.DFM}

procedure THookForm.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(AppWindowHook);
end;

procedure THookForm.FormDestroy(Sender: TObject);
begin
  Application.UnhookMainWindow(AppWindowHook);
end;

function THookForm.AppWindowHook(var Message: TMessage): Boolean;
const
  LogStr = 'Message ID: $%x, WParam: $%x, LParam: $%x';
begin
```

continues

LISTING 13.3 Continued

```
  Result := True;
  if DoLog.Checked then
    with Message do
      LogList.Items.Add(Format(LogStr, [Msg, WParam, LParam]));
end;

procedure THookForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, WM_NULL, 0, 0);
end;

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

end.
```

Preventing Multiple Application Instances

Multiple instances means running more than one copy of your program simultaneously. The capability to run multiple instances of an application independently from one another is a feature provided by the Win32 operating system. While this feature is great, there are cases that arise when we only wish for the end user to be able to run one copy of a given application at a time. An example of this type of application might be one that controls a unique resource on the machine, such as a modem or the parellel port. In such cases, it becomes necessary to write some code into your application to solve this problem by allowing only one copy of an application to run at any given time.

This was a fairly simple task in the 16-bit Windows world: The `hPrevInst` system variable can be used to determine whether multiple copies of an application are running simultaneously. If the value of `hPrevInst` is nonzero, another instance of the application is active. However, as explained in Chapter 3, "The Win32 API," Win32 provides a thick layer of R32 insulation between each process, which isolates each from the other. Because of this, the value for `hPrevInst` is always zero for Win32 applications.

Another technique that works for both 16-bit and 32-bit Windows is to use the `FindWindow()` API function to search for an already-active `Application` window. This solution has two disadvantages, however. First, `FindWindow()` allows you to search for a window based only on its class name or caption. Depending on the class name isn't a particularly robust solution because there's no guarantee that the class name of your form is unique throughout the system.

Searching based on the form caption has obvious drawbacks in that the solution breaks down if you attempt to change the caption of the form while it runs (as do applications such as Delphi and Microsoft Word). The second drawback to `FindWindow()` is that it tends to be slow because it must iterate over all top-level windows.

The optimal solution for Win32, then, is to use some type of API object that's persistent across processes. As explained in Chapter 11, "Writing Multithreaded Applications," several of the thread-synchronization objects are persistent across multiple processes. Because of their simplicity of use, mutexes provide an ideal solution to this problem.

The first time an application is run, a mutex is created using the `CreateMutex()` API function. The `lpName` parameter of this function holds a unique string identifier. Subsequent instances of this application should try to open the mutex by name using the `OpenMutex()` function. `OpenMutex()` will succeed only when a mutex has already been created using the `CreateMutex()` function.

Additionally, when you attempt to run a second instance of these applications, the first instance of the application should come into focus. The most elegant approach to focusing the main form of the previous instance is to use a registered window message obtained by the `RegisterWindowMessage()` function to create a message identifier unique to your application. You then can have the initial instance of your application respond to this message by returning its main window handle, which can then be focused by the second instance. This approach is illustrated in Listing 13.4, which shows the source for the `MultInst.pas` unit, and Listing 13.5, `OIMain.pas`, which is the main unit of the OneInst project. The application is shown in all its glory in Figure 13.3.

FIGURE 13.3
The main form for the OneInst project.

LISTING 13.4 The `MultInst.pas` Unit, Which Permits Only One Application Instance

```
unit MultInst;

interface

const
  MI_QUERYWINDOWHANDLE    = 1;
```

continues

LISTING 13.4 Continued

```
  MI_RESPONDWINDOWHANDLE = 2;

  MI_ERROR_NONE          = 0;
  MI_ERROR_FAILSUBCLASS  = 1;
  MI_ERROR_CREATINGMUTEX = 2;

// Call this function to determine if error occurred in startup.
// Value will be one or more of the MI_ERROR_* error flags.
function GetMIError: Integer;

implementation

uses Forms, Windows, SysUtils;

const
  UniqueAppStr = 'DDG.I_am_the_Eggman!';

var
  MessageId: Integer;
  WProc: TFNWndProc;
  MutHandle: THandle;
  MIError: Integer;

function GetMIError: Integer;
begin
  Result := MIError;
end;

function NewWndProc(Handle: HWND; Msg: Integer; wParam, lParam: Longint):
  Longint; stdcall;
begin
  Result := 0;
  // If this is the registered message...
  if Msg = MessageID then
  begin
    case wParam of
      MI_QUERYWINDOWHANDLE:
        // A new instance is asking for main window handle in order
        // to focus the main window, so normalize app and send back
        // message with main window handle.
        begin
          if IsIconic(Application.Handle) then
          begin
            Application.MainForm.WindowState := wsNormal;
            Application.Restore;
```

```
          end;
          PostMessage(HWND(lParam), MessageID, MI_RESPONDWINDOWHANDLE,
            Application.MainForm.Handle);
        end;
      MI_RESPONDWINDOWHANDLE:
        // The running instance has returned its main window handle,
        // so we need to focus it and go away.
        begin
          SetForegroundWindow(HWND(lParam));
          Application.Terminate;
        end;
    end;
  end
  // Otherwise, pass message on to old window proc
  else
    Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
end;

procedure SubClassApplication;
begin
  // We subclass Application window procedure so that
  // Application.OnMessage remains available for user.
  WProc := TFNWndProc(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Longint(@NewWndProc)));
  // Set appropriate error flag if error condition occurred
  if WProc = nil then
    MIError := MIError or MI_ERROR_FAILSUBCLASS;
end;

procedure DoFirstInstance;
// This is called only for the first instance of the application
begin
  // Create the mutex with the (hopefully) unique string
  MutHandle := CreateMutex(nil, False, UniqueAppStr);
  if MutHandle = 0 then
    MIError := MIError or MI_ERROR_CREATINGMUTEX;
end;

procedure BroadcastFocusMessage;
// This is called when there is already an instance running.
var
  BSMRecipients: DWORD;
begin
  // Prevent main form from flashing
  Application.ShowMainForm := False;
  // Post message to try to establish a dialogue with previous instance
```

continues

LISTING 13.4 Continued

```
  BSMRecipients := BSM_APPLICATIONS;
  BroadCastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @BSMRecipients, MessageID, MI_QUERYWINDOWHANDLE,
    Application.Handle);
end;

procedure InitInstance;
begin
  SubClassApplication;   // hook application message loop
  MutHandle := OpenMutex(MUTEX_ALL_ACCESS, False, UniqueAppStr);
  if MutHandle = 0 then
    // Mutex object has not yet been created, meaning that no previous
    // instance has been created.
    DoFirstInstance
  else
    BroadcastFocusMessage;
end;

initialization
  MessageID := RegisterWindowMessage(UniqueAppStr);
  InitInstance;
finalization
  // Restore old application window procedure
  if WProc <> Nil then
    SetWindowLong(Application.Handle, GWL_WNDPROC, LongInt(WProc));
  if MutHandle <> 0 then CloseHandle(MutHandle);  // Free mutex
end.
```

LISTING 13.5 OIMain.pas

```
unit OIMain;

interface

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

type
  TMainForm = class(TForm)
    Label1: TLabel;
    CloseBtn: TButton;
    procedure CloseBtnClick(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

uses MultInst;

{$R *.DFM}

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

end.
```


Using BASM with Delphi

Because Delphi is based on a true compiler, one benefit you receive is the capacity to write assembly code right in the middle of your Object Pascal procedures and functions. This capability is facilitated through Delphi's built-in assembler (BASM). Before you learn about BASM, you should learn when to use assembly language in your Delphi programs. It's great to have such a powerful tool at your disposal, but, like any good thing, BASM can be overdone. If you follow these simple BASM rules, you can help yourself write better, cleaner, and more portable code:

- Never use assembly language for something that can be done in Object Pascal. For example, you wouldn't write assembly language routines to communicate through the serial ports because the Win32 API provides built-in functions for serial communications.
- Don't over-optimize your programs with assembly language. Hand-optimized assembly might run faster than Object Pascal code—but at the price of readability and maintainability. Object Pascal is a language that communicates algorithms so naturally that it's a shame to have that communication muddled by a bunch of low-level register operations. In addition, after all your assembler toils, you might be surprised to find out that Delphi's optimizing compiler often compiles code that executes faster than handwritten assembly code.

- Always comment your assembly code thoroughly. Your code will probably be read in the future by another programmer—or even by you—and lack of comments can make it difficult to understand.
- Don't use BASM to access machine hardware. Although Windows 95/98 will let you get away with this in most cases, Windows NT/2000 won't.
- Where possible, wrap your assembly language code in procedures or functions callable from Object Pascal. This will make your code not only easier to maintain but also easier to port to other platforms when the time comes.

NOTE

This section doesn't teach you assembler programming, but it shows you the Delphi spin on assembler if you're already familiar with the language.

Also, if you programmed in BASM with Delphi 1, bear in mind that in 32-bit Delphi, BASM is a whole new ballgame. Because you must now write 32-bit assembly language, almost all your 16-bit BASM code will have to be rewritten for the new platform. The fact that BASM code can require so much care to maintain is yet another reason to minimize your use of BASM in applications.

How Does BASM Work?

Using assembly code in your Delphi applications is easier than you might think. In fact, it's so simple that it's scary. Just use the `asm` keyword followed by your assembly code and then an `end`. The following code fragment demonstrates how to use assembly code inline:

```
var
  i: integer;
begin
  i := 0;
  asm
    mov eax, i
    inc eax
    mov i, eax
  end;
  { i has incremented by one }
```

This snippet declares a variable `i` and initializes it to `0`. It then moves the value of `i` into the `eax` register, increments the register by one, and moves the value of the `eax` register back into `i`. This illustrates not only how easy it is to use BASM, but, as the usage of the variable `i` shows, how easily you can access your Pascal variables from BASM.

Easy Parameter Access

Not only is it easy to access variables declared globally or locally to a procedure, it's just as easy to access variables passed into procedures, as the following code illustrates:

```
procedure Foo(I: integer);
begin
  { some code }
  asm
    mov eax, I
    inc eax
    mov I, eax
  end;
  { I has incremented by one }
  { some more code }
end;
```

The capability to access parameters by name is important because you don't have to reference variables passed into a procedure through the stack base pointer (`ebp`) register as you would in a normal assembly program. In a regular assembly language procedure, you would have to refer to the variable `I` as `[ebp+4]` (its offset from the stack's base pointer).

NOTE

When you use BASM to reference parameters passed into a procedure, remember that you can access those parameters by name, and you don't have to access them by their offset from the `ebp` register. Accessing by offset from `ebp` makes your code more difficult to maintain.

var Parameters

Remember that when a parameter is declared as `var` in a function or procedure's parameter list, a pointer to that variable is passed instead of the value. This means that when you reference `var` parameters within a BASM block, you must take into account that the parameter is a 32-bit pointer to a variable and not a variable instance. To expand on the earlier sample snippet, the following example shows how you would increment the variable `I` if it were passed in as a `var` parameter:

```
procedure Foo(var I: integer);
begin
  { some code }
  asm
    mov eax, I
    inc dword ptr [eax]
```

```
  end;
  { I has now been incremented by one }
  { some more code }
end;
```

Register Calling Convention

Remember that the default calling convention for Object Pascal functions and procedures is `register`. Taking advantage of this method of parameter passing can help you to optimize your code. The register calling convention dictates that the first three 32-bit parameters are passed in the `eax`, `edx`, and `ecx` registers. This means that for the function declaration

```
function BlahBlah(I1, I2, I3: Integer): Integer;
```

you can count on the fact that the value of `I1` is stored in `eax`, `I2` in `edx`, and `I3` in `ecx`. Consider the following method as another example:

```
procedure TSomeObject.SomeProc(S1, S2: PChar);
```

Here, the value of `S1` will be passed in `ecx`, `S2` in `edx`, and the implicit `Self` parameter will be passed in `eax`.

All-Assembly Procedures

Object Pascal enables you to write procedures and functions entirely in assembly language simply by beginning the function or procedure with the word `asm`, rather than `begin`, as shown here:

```
function IncAnInt(I: Integer): Integer;
asm
  mov eax, I
  inc eax
end;
```

NOTE

If you're poring over 16-bit code, you should know that it's no longer necessary to use the `assembler` directive from Delphi 1 days. That directive is simply ignored by the 32-bit Delphi compiler.

The preceding procedure accepts an integer variable `I` and increments it. Because the variable value is placed in the `eax` register, that's the value returned by the function. Table 13.1 shows how different types of data are returned from a function in Delphi.

TABLE 13.1 How Values Are Returned from Delphi Functions

Return Type	*Return Method*
`Char`, `Byte`	`al` register.
`SmallInt`, `Word`	`ax` register.
`Integer`, `LongWord`, `AnsiString`, `Pointer`, `class`	`eax` register.
`Real48`	`eax` contains a pointer to data on the stack.
`Int64`	`edx:eax` register pair.
`Single`, `Double`, `Extended`, `Comp`	`ST(0)` on 8087's register stack.

NOTE

A `ShortString` type is returned as a pointer to a temporary instance of a string on the stack.

Records

BASM provides a slick shortcut for accessing the fields of a record. You can access the fields of any record in a BASM block using the syntax *`Register.Type.Field`*. For example, consider a record defined as follows:

```
type
  TDumbRec = record
    i: integer;
    c: char;
  end;
```

Also, consider a function that accepts a `TDumbRec` as a reference parameter, as shown here:

```
procedure ManipulateRec(var DR: TDumbRec);
asm
  mov [eax].TDumbRec.i, 24
  mov [eax].TDumbRec.c, 's'
end;
```

Notice the shortcut syntax for accessing the fields of a record. The alternative would be to manually calculate the proper offset into the record to get or set the appropriate value. Use this technique wherever you use records in BASM to make your BASM more resilient to potential changes to data types.

Using Windows Hooks

Windows *hooks* give programmers the means to control the occurrence and handling of system events. A hook offers perhaps the ultimate degree of power for an applications programmer because it enables the programmer to preview and modify system events and messages as well as to prevent system events and messages from occurring systemwide.

Setting the Hook

A Windows hook is set using the `SetWindowsHookEx()` API function:

```
function SetWindowsHookEx(idHook: Integer; lpfn: TFNHookProc; hmod: HINST;
  dwThreadID: DWORD): HHOOK; stdcall;
```

CAUTION

Use only the `SetWindowsHookEx()` function—not the `SetWindowsHook()` function—in your applications. `SetWindowsHook()`, which existed in Windows 3.*x*, is not implemented in the Win32 API.

The `idHook` parameter describes the type of hook to be installed. This can be any one of the predefined hook constants shown in Table 13.2.

TABLE 13.2 Windows Hook Constants

Hook Constant	*Description*
`WH_CALLWNDPROC`	A window procedure filter. The hook procedure is called whenever a message is sent to a window procedure.
`WH_CALLWNDPROCRET*`	Installs a hook procedure that monitors messages after they've been processed by the destination window procedure.
`WH_CBT`	A computer-based training filter. The hook procedure is called before processing most window-management, mouse, and keyboard messages.
`WH_DEBUG`	A debugging filter. The hook function is called before any other Windows hook.
`WH_GETMESSAGE`	A message filter. The hook function is called whenever a message is retrieved from the application queue.
`WH_HARDWARE`	A hardware message filter. The hook function is called whenever a hardware message is retrieved from the application queue.

Hook Constant	*Description*
WH_JOURNALPLAYBACK	The hook function is called whenever a message is retrieved from the system queue. Typically used to insert system events into the queue.
WH_JOURNALRECORD	The hook function is called whenever an event is requested from the system queue. Typically used to "record" system events.
WH_KEYBOARD	A keyboard filter. The hook function is called whenever a WM_KEYDOWN or WM_KEYUP message is retrieved from the application queue.
WH_KEYBOARD_LL*	A low-level keyboard filter.
WH_MOUSE	A mouse message filter. The hook function is called whenever a mouse message is retrieved from the application queue.
WH_MOUSE_LL*	A low-level mouse message filter.
WH_MSGFILTER	A special message filter. The hook function is called whenever an application's dialog box, menu, or message box is about to process a message.
WH_SHELL	A shell application filter. The hook function is called when top-level windows are created and destroyed as well as when the shell application needs to become active.

* = *available only on Windows NT 4.0 and Windows 2000*

The lpfn parameter is the address of the callback function to act as the Windows hook function. This function is of type TFNHookProc, which is defined as follows:

```
TFNHookProc = function (code: Integer; wparam: WPARAM; lparam: LPARAM):
    LRESULT stdcall;
```

The contents of each of the hook function's parameters vary according to the type of hook installed; the parameters are documented in the Win32 API help.

The hMod parameter should be the value of hInstance in the EXE or DLL containing the hook callback.

The dwThreadID parameter identifies the thread with which the hook is to be associated. If this parameter is zero, the hook will be associated with all threads.

The return value is a hook handle that you must save in a global variable for later use.

Windows can have multiple hooks installed at one time, and it can even have the same type of hook installed multiple times.

Note also that some hooks operate with the restriction that they must be implemented from a DLL. Check the Win32 API documentation for details on each specific hook.

CAUTION

One serious limitation for system hooks is that new instances of the hook DLL are loaded into each process address space separately. Because of this, the hook DLL cannot communicate directly with the host application that set the hook. You have to go through messages or shared memory areas (such as the memory mapped files described in Chapter 12, "Working with Files") to communicate with the host application.

Using the Hook Function

The values of the hook function's `Code`, `wParam`, and `lParam` parameters vary depending on the type of hook installed, and they're documented in the Windows API help. These parameters all have one thing in common: Depending on the value of `Code`, you're responsible for calling the next hook in the chain.

To call the next hook, use the `CallNextHookEx()` API function:

```
Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
```

CAUTION

When calling the next hook in the chain, don't call `DefHookProc()`. This is another unimplemented Windows 3.*x* function.

Using the Unhook Function

When you want to release the Windows hook, you just need to call the `UnhookWindowsHookEx()` API function, passing it the hook handle as a parameter. Again, be careful not to call `UnhookWindowsHook()` here because it's another old-style function:

```
UnhookWindowsHookEx(HookHandle);
```

Using SendKeys: A JournalPlayback Hook

If you come to Delphi from an environment such as Visual Basic or Paradox for Windows, you might be familiar with a function called `SendKeys()`. `SendKeys()` enables you to pass it a string of characters that it then plays back as if they were typed from the keyboard, and all the keystrokes are sent to the active window. Because Delphi doesn't have a function like this built in, creating one proves a great opportunity to add a powerful feature to Delphi as well as to demonstrate how to implement a `wh_JournalPlayback` hook from within Delphi.

Deciding Whether to Use a JournalPlayback Hook

There are a number of reasons why a hook is the best way to send keystrokes to your application or another application. You might wonder, "Why not just post `wm_KeyDown` and `wm_KeyUp` messages?" The primary reason is that you might not know the handle of the window to which you want to post messages, or that the handle for that window might periodically change. And, of course, if you don't know the window handle, you can't send a message. Also, some applications call API functions to check the state of the keyboard in addition to looking at messages to obtain information on keystrokes.

Understanding How SendKeys Works

The declaration of the `SendKeys()` function looks like this:

```
function SendKeys(S: String): TSendKeyError; export;
```

The `TSendKeyError` return type is an enumerated type that indicates the error condition. It can be any one of the values shown in Table 13.3.

TABLE 13.3 `Sendkey` Error Codes

Value	*Meaning*
`sk_None`	The function was successful.
`sk_FailSetHook`	The Windows hook couldn't be set.
`sk_InvalidToken`	An invalid token was detected in the string.
`sk_UnknownError`	Some other unknown but fatal error occurred.
`sk_AlreadyPlaying`	The hook is currently active, and keystrokes are already being played back.

`S` can include any alphanumeric character or `@` for the Alt key, `^` for the Ctrl key, or `~` for the Shift key. `SendKeys()` also enables you to specify special keyboard keys in curly braces, as depicted in the `KeyDefs.pas` unit in Listing 13.6.

LISTING 13.6 `KeyDefs.pas`: Special Key Definitions for `SendKeys()`

```
unit KeyDefs;

interface

uses Windows;

const
  MaxKeys = 24;
```

continues

LISTING 13.6 Continued

```
  ControlKey = '^';
  AltKey = '@';
  ShiftKey = '~';
  KeyGroupOpen = '{';
  KeyGroupClose = '}';

type
  TKeyString = String[7];

  TKeyDef = record
    Key: TKeyString;
    vkCode: Byte;
  end;

const
  KeyDefArray : array[1..MaxKeys] of TKeyDef = (
    (Key: 'F1';     vkCode: vk_F1),
    (Key: 'F2';     vkCode: vk_F2),
    (Key: 'F3';     vkCode: vk_F3),
    (Key: 'F4';     vkCode: vk_F4),
    (Key: 'F5';     vkCode: vk_F5),
    (Key: 'F6';     vkCode: vk_F6),
    (Key: 'F7';     vkCode: vk_F7),
    (Key: 'F8';     vkCode: vk_F8),
    (Key: 'F9';     vkCode: vk_F9),
    (Key: 'F10';    vkCode: vk_F10),
    (Key: 'F11';    vkCode: vk_F11),
    (Key: 'F12';    vkCode: vk_F12),
    (Key: 'INSERT'; vkCode: vk_Insert),
    (Key: 'DELETE'; vkCode: vk_Delete),
    (Key: 'HOME';   vkCode: vk_Home),
    (Key: 'END';    vkCode: vk_End),
    (Key: 'PGUP';   vkCode: vk_Prior),
    (Key: 'PGDN';   vkCode: vk_Next),
    (Key: 'TAB';    vkCode: vk_Tab),
    (Key: 'ENTER';  vkCode: vk_Return),
    (Key: 'BKSP';   vkCode: vk_Back),
    (Key: 'PRTSC';  vkCode: vk_SnapShot),
    (Key: 'SHIFT';  vkCode: vk_Shift),
    (Key: 'ESCAPE'; vkCode: vk_Escape));

function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;

implementation
```

```
uses SysUtils;

function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;
{ function searches array for token passed in Key, and returns the }
{ virtual key code in Code. }
var
  i: word;
begin
  Result := False;
  for i := Low(KeyDefArray) to High(KeyDefArray) do
    if UpperCase(Key) = KeyDefArray[i].Key then begin
      Code := KeyDefArray[i].vkCode;
      Result := True;
      Break;
    end;
end;

end.
```

After receiving the string, `SendKeys()` parses the individual key presses out of the string and adds each of the key presses to a list in the form of message records containing `wm_KeyUp` and `wm_KeyDown` messages. These messages then are played back to Windows through a `wh_JournalPlayback` hook.

Creating Key Presses

After each key press is parsed out of the string, the virtual key code and message (the message can be `wm_KeyUp`, `wm_KeyDown`, `wm_SysKeyUp`, or `wm_SysKeyDown`) are passed to a procedure called `MakeMessage()`. `MakeMessage()` creates a new message record for the key press and adds it to a list of messages called `MessageList`. The message record used here isn't the standard `TMessage` that you're familiar with, or even the `TMsg` record discussed in Chapter 5. This record is called a `TEvent` message, and it represents a system queue message. The definition is as follows:

```
type
  { Message Structure used in Journaling }
  PEventMsg = ^TEventMsg;
  TEventMsg = packed record
    message: UINT;
    paramL: UINT;
    paramH: UINT;
    time: DWORD;
    hwnd: HWND;
  end;
```

Table 13.4 shows the values for TEventMsg's fields.

TABLE 13.4 Values for TEventMsg Fields

Field	*Value*
message	The message constant. Can be wm_(Sys)KeyUp or wm_SysKeyDown for a keyboard message. Can be wm_*X*ButtonUp, wm_*X*ButtonDown, or wm_MouseMove for a mouse message.
paramL	If message is a keyboard message, this field holds the virtual key code. If message is a mouse message, wParam contains the x coordinate of the mouse cursor (in screen units).
paramH	If message is a keyboard message, this field holds the scan code of the key. If it's a mouse message, lParam contains the y coordinate of the mouse cursor.
time	The time, in system ticks, that the message occurred.
hwnd	Identifies the window to which the message is posted. This parameter isn't used for wh_JournalPlayback hooks.

Because the table in the KeyDefs unit maps only to the virtual key code, you must find a way to determine the scan code of the key given the virtual key code. Luckily, the Windows API provides a function called MapVirtualKey() that does just that. The following code shows the source for the MakeMessage() procedure:

```
procedure MakeMessage(vKey: byte; M: Cardinal);
{ procedure builds a TEventMsg record that emulates a keystroke and }
{ adds it to message list }
var
  E: PEventMsg;
begin
  New(E);                                  // allocate a message record
  with E^ do begin
    message := M;                          // set message field
    paramL := vKey;                        // vk code in ParamL
    paramH := MapVirtualKey(vKey, 0);      // scan code in ParamH
    time := GetTickCount;                  // set time
    hwnd := 0;                             // ignored
  end;
  MessageList.Add(E);
end;
```

After the entire message list is created, the hook can be set to play back the key sequence. You do this through a procedure called StartPlayback(). StartPlayback primes the pump by placing the first message from the list into a global buffer. It also initializes a global buffer that keeps track of how many messages have been played and the flags that indicate the state of the

Ctrl, Alt, and Shift keys. This procedure then sets the hook. `StartPlayBack()` is shown in the following code:

```
procedure StartPlayback;
{ Initializes globals and sets the hook }
begin
  { grab first message from list and place in buffer in case we }
  { get an hc_GetNext before an hc_Skip }
MessageBuffer := TEventMsg(MessageList.Items[0]^);
  { initialize message count and play indicator }
  MsgCount := 0;
  { initialize Alt, Control, and Shift key flags }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
  { set the hook! }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('Couldn''t set hook')
  else
    Playing := True;
end;
```


As you might notice from the `SetWindowsHookEx()` call, `Play` is the name of the hook function. The declaration for `Play` is as follows:

```
function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
```

Table 13.5 shows its parameters.

TABLE 13.5 Parameters for `Play()`, the Windows Hook Function

Value	*Meaning*
`Code`	A value of `hc_GetNext` indicates that you should prepare the next message in the list for processing. You do this by copying the next message from the list into your global buffer. A value of `hc_Skip` means that a pointer to the next message should be placed into the `lParam` parameter for processing. Any other value means that you should call `CallNextHookEx()` and pass on the parameters to the next hook in the chain.
`wParam`	Unused.
`lParam`	If `Code` is `hc_Skip`, you should place a pointer to the next `TEventMsg` record in the `lParam` parameter.
`Return value`	Returns zero if `Code` is `hc_GetNext`. If `Code` is `hc_Skip`, returns the amount of time (in ticks) before this message should be processed. If zero is returned, the message is processed. Otherwise, the return value should be the return value of `CallNextHookEx()`.

Listing 13.7 shows the complete source code to the SendKey.pas unit.

LISTING 13.7 The SendKey.pas Unit

```
unit SendKey;

interface

uses
 SysUtils, Windows, Messages, Classes, KeyDefs;

type
  { Error codes }
  TSendKeyError = (sk_None, sk_FailSetHook, sk_InvalidToken,
    sk_UnknownError, sk_AlreadyPlaying);
  { first vk code to last vk code }
  TvkKeySet = set of vk_LButton..vk_Scroll;

  { exceptions }
  ESendKeyError = class(Exception);
  ESKSetHookError = class(ESendKeyError);
  ESKInvalidToken = class(ESendKeyError);
  ESKAlreadyPlaying = class(ESendKeyError);

function SendKeys(S: String): TSendKeyError;
procedure WaitForHook;
procedure StopPlayback;

var
  Playing: Boolean;

implementation

uses Forms;

type
  { a TList descendant that know how to dispose of its contents }
  TMessageList = class(TList)
  public
    destructor Destroy; override;
  end;

const
  { valid "sys" keys }
  vkKeySet: TvkKeySet = [Ord('A')..Ord('Z'), vk_Menu, vk_F1..vk_F12];

destructor TMessageList.Destroy;
```

```
var
  i: longint;
begin
  { deallocate all the message records before discarding the list }
  for i := 0 to Count - 1 do
    Dispose(PEventMsg(Items[i]));
  inherited Destroy;
end;

var
  { variables global to the DLL }
  MsgCount: word = 0;
  MessageBuffer: TEventMsg;
  HookHandle: hHook = 0;
  MessageList: TMessageList = Nil;
  AltPressed, ControlPressed, ShiftPressed: Boolean;

procedure StopPlayback;
{ Unhook the hook, and clean up }
begin
  { if Hook is currently active, then unplug it }
  if Playing then
    UnhookWindowsHookEx(HookHandle);
  MessageList.Free;
  Playing := False;
end;

function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
{ This is the JournalPlayback callback function.  It is called by }
{ Windows when Windows polls for hardware events.  The code parameter }
{ indicates what to do. }
begin
  case Code of
    HC_SKIP:
      { HC_SKIP means to pull the next message out of our list. If we }
      { are at the end of the list, it's okay to unhook the }
      { JournalPlayback hook from here. }
      begin
        { increment message counter }
        inc(MsgCount);
        { check to see if all messages have been played }
        if MsgCount >= MessageList.Count then StopPlayback
        { otherwise copy next message from list into buffer }
        else MessageBuffer := TEventMsg(MessageList.Items[MsgCount]^);
        Result := 0;
      end;
```

continues

LISTING 13.7 Continued

```
    HC_GETNEXT:
      { HC_GETNEXT means to fill the wParam and lParam with the proper }
      { values so that the message can be played back.  DO NOT unhook }
      { hook from within here.  Return value indicates how much time }
      { until Windows should playback message.  We'll return 0 so that }
      { it is processed right away. }
      begin
        { move message in buffer to message queue }
        PEventMsg(lParam)^ := MessageBuffer;
        Result := 0  { process immediately }
      end
    else
      { if Code isn't HC_SKIP or HC_GETNEXT, call next hook in chain }
      Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
  end;
end;

procedure StartPlayback;
{ Initializes globals and sets the hook }
begin
  { grab first message from list and place in buffer in case we }
  { get a hc_GetNext before and hc_Skip }
  MessageBuffer := TEventMsg(MessageList.Items[0]^);
  { initialize message count and play indicator }
  MsgCount := 0;
  { initialize Alt, Control, and Shift key flags }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
  { set the hook! }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('Failed to set hook');
  Playing := True;
end;

procedure MakeMessage(vKey: byte; M: Cardinal);
{ procedure builds a TEventMsg record that emulates a keystroke and }
{ adds it to message list }
var
  E: PEventMsg;
begin
  New(E);                                  // allocate a message record
  with E^ do
```

```
  begin
    message := M;                          // set message field
    paramL := vKey;                        // vk code in ParamL
    paramH := MapVirtualKey(vKey, 0);      // scan code in ParamH
    time := GetTickCount;                  // set time
    hwnd := 0;                             // ignored
  end;
  MessageList.Add(E);
end;

procedure KeyDown(vKey: byte);
{ Generates KeyDownMessage }
begin
  { don't generate a "sys" key if the control key is pressed }
  { (This is a Windows quirk) }
  if AltPressed and (not ControlPressed) and  (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyDown)
  else
    MakeMessage(vKey, wm_KeyDown);
end;

procedure KeyUp(vKey: byte);
{ Generates KeyUp message }
begin
  { don't generate a "sys" key if the control key is pressed }
  { (This is a Windows quirk) }
  if AltPressed and (not ControlPressed) and (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyUp)
  else
    MakeMessage(vKey, wm_KeyUp);
end;

procedure SimKeyPresses(VKeyCode: Word);
{ This function simulates keypresses for the given key, taking into }
{ account the current state of Alt, Control, and Shift keys }
begin
  { press Alt key if flag has been set }
  if AltPressed then
    KeyDown(vk_Menu);
  { press Control key if flag has been set }
  if ControlPressed then
    KeyDown(vk_Control);
  { if shift is pressed, or shifted key and control is not pressed... }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or
    ShiftPressed then
    KeyDown(vk_Shift);    { ...press shift }
```

continues

LISTING 13.7 Continued

```
  KeyDown(Lo(VKeyCode));  { press key down }
  KeyUp(Lo(VKeyCode));    { release key }
  { if shift is pressed, or shifted key and control is not pressed... }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or
    ShiftPressed then
    KeyUp(vk_Shift);      { ...release shift }
  { if shift flag is set, reset flag }
  if ShiftPressed then begin
    ShiftPressed := False;
  end;
  { Release Control key if flag has been set, reset flag }
  if ControlPressed then begin
    KeyUp(vk_Control);
    ControlPressed := False;
  end;
  { Release Alt key if flag has been set, reset flag }
  if AltPressed then begin
    KeyUp(vk_Menu);
    AltPressed := False;
  end;
end;

procedure ProcessKey(S: String);
{ This function parses each character in the string to create the }
{ message list }
var
  KeyCode: word;
  Key: byte;
  index: integer;
  Token: TKeyString;
begin
  index := 1;
  repeat
    case S[index] of
      KeyGroupOpen:
        { It's the beginning of a special token! }
        begin
          Token := '';
          inc(index);
          while S[index] <> KeyGroupClose do begin
            { add to Token until the end token symbol is encountered }
            Token := Token + S[index];
            inc(index);
            { check to make sure the token's not too long }
```

```
            if (Length(Token) = 7) and (S[index] <> KeyGroupClose) then
              raise ESKInvalidToken.Create('No closing brace');
          end;
          { look for token in array, Key parameter will }
          { contain vk code if successful }
          if not FindKeyInArray(Token, Key) then
            raise ESKInvalidToken.Create('Invalid token');
          { simulate keypress sequence }
          SimKeyPresses(MakeWord(Key, 0));
        end;
      AltKey: AltPressed := True;           // set Alt flag
      ControlKey: ControlPressed := True;   // set Control flag
      ShiftKey: ShiftPressed := True;       // set Shift flag
      else begin
      { A normal character was pressed }
        { convert character into a word where the high byte contains }
        { the shift state and the low byte contains the vk code }
        KeyCode := vkKeyScan(S[index]);
        { simulate keypress sequence }
        SimKeyPresses(KeyCode);
      end;
    end;
    Inc(index);
  until index > Length(S);
end;

procedure WaitForHook;
begin
  repeat Application.ProcessMessages until not Playing;
end;

function SendKeys(S: String): TSendKeyError;
{ This is the one entry point.  Based on the string passed in the S  }
{ parameter, this function creates a list of keyup/keydown messages, }
{ sets a JournalPlayback hook, and replays the keystroke messages.   }
begin
  Result := sk_None;                    // assume success
  try
    if Playing then raise ESKAlreadyPlaying.Create('');
    MessageList := TMessageList.Create;  // create list of messages
    ProcessKey(S);                       // create messages from string
    StartPlayback;                      // set hook and play back messages
  except
    { if an exception occurs, return an error code, and clean up }
    on E:ESendKeyError do
    begin
```

continues

LISTING 13.7 Continued

```
      MessageList.Free;
      if E is ESKSetHookError then
        Result := sk_FailSetHook
      else if E is ESKInvalidToken then
        Result := sk_InvalidToken
      else if E is ESKAlreadyPlaying then
        Result := sk_AlreadyPlaying;
    end
    else
      Result := sk_UnknownError;  // Catch-all exception handler
  end;
end;

end.
```

Using SendKeys()

In this section, you'll create a small project that demonstrates the `SendKeys()` function. Start with a form that contains two `TEdit` components and several `TButton` components, as shown in Figure 13.4. This project is called `TestSend.dpr`.

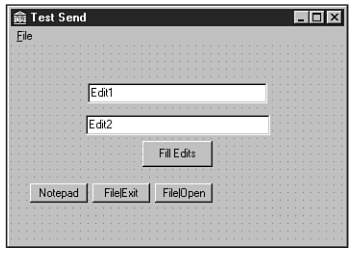

FIGURE 13.4
The TestSend main form.

Listing 13.8 shows the source code for TestSend's main unit, `Main.pas`. This unit includes event handlers for the button-click events.

LISTING 13.8 The Source Code for `Main.pas`

```
unit Main;

interface

uses
```

```
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Menus;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Button2: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Exit1: TMenuItem;
    Button4: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses SendKey, KeyDefs;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;                              // focus Edit1
  SendKeys('^{DELETE}I love...');              // send keys to Edit1
  WaitForHook;                                 // let keys playback
  Perform(WM_NEXTDLGCTL, 0, 0);                // move to Edit2
  SendKeys('~delphi ~developer''s ~guide!');   // send keys to Edit2
end;
```

continues

LISTING 13.8 Continued

```
procedure TForm1.Button2Click(Sender: TObject);
var
  H: hWnd;
  PI: TProcessInformation;
  SI: TStartupInfo;
begin
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  { Invoke notepad }
  if CreateProcess(nil, 'notepad', nil, nil, False, 0, nil, nil, SI,
    PI) then
  begin
    { wait until notepad is ready to receive keystrokes }
    WaitForInputIdle(PI.hProcess, INFINITE);
    { find new notepad window }
    H := FindWindow('Notepad', 'Untitled - Notepad');
    if SetForegroundWindow(H) then            // bring it to front
      SendKeys('Hello from the Delphi Developer''s Guide SendKeys ' +
        'example!{ENTER}');  // send keys!
  end
  else
    MessageDlg(Format('Failed to invoke Notepad.  Error code %d',
      [GetLastError]), mtError, [mbOk], 0);
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
  ShowMessage('Open');
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fx');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WaitForHook;
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fo');
end;

end.
```

After you click `Button1`, `SendKeys()` is called, and the following key presses are sent: Shift+Del deletes the contents of `Edit1`; "`I love...`" then is typed into `Edit1`; a tab character is sent, which moves the focus to `Edit2`, where Shift+D, "`elphi `", Shift+D, "`evelopers `", Shift+G, "`uide!`" is sent.

The `OnClick` handler for `Button2` is also interesting. This method uses the `CreateProcess()` API function to invoke an instance of Notepad. It then uses the `WaitForInputIdle()` API function to pause until Notepad's process is ready for input. Finally, it types a message in the Notepad window.

Using C/C++ OBJ Files

Delphi provides you with the capability for linking object (OBJ) files created using another compiler directly into your Delphi programs. You can link an object file into your Object Pascal code by using the `$L` or `$LINK` directives. The syntax for this is as follows:

```
{$L filename.obj}
```

After the object file is linked, you must define each function you want to call out of the object file in your Object Pascal code. Use the `external` directive to indicate that the Pascal compiler should wait until link time to attempt to resolve the function name. For example, the following line of code defines an external function called `Foo` that neither takes nor returns any parameters:

```
procedure Foo; external;
```

Although this capability might seem powerful on the surface, it comes with a number of limitations that make this feature difficult to implement in many cases:

- Object Pascal can directly access only code, not data, contained in object files (although there is a trick to getting at data in an OBJ, which you'll see later). However, Pascal data can be accessed from object files.
- Object Pascal can't link with LIB (static library) files.
- Object files containing C++ classes will not link due to the implicit references to C++ RTL. Although it might be possible to resolve these references by pulling apart the C++ RTL into OBJs, it's generally more trouble than it's worth.

- Object files must be in the Intel OMF format. This is the output format of the Borland C++ compilers, but not the Microsoft C++ compilers, which produce COFF-format OBJ files.

NOTE

One previously stifling limitation that has recently been addressed by the Delphi compiler is the capability to resolve OBJ-to-OBJ references. In earlier versions of Delphi, object files couldn't contain references to code or data stored in other object files.

Calling a Function

Suppose you had a C++ object file called `ccode.obj` that includes a function with the following prototype:

```
int  __fastcall SAYHELLO(char * hellostr)
```

To call this function from a Delphi application, you must first link the object file into the EXE using either the `$L` or `$LINK` directive:

```
{$L ccode.obj}
```

After that, you must create an Object Pascal definition for the function, as shown here:

```
function SayHello(Text: PChar): integer; external;
```

CAUTION

Notice the use of the `__fastcall` directive in C++, which serves to ensure that the calling conventions used in the C++ and Object Pascal code are the same. Heinous crash errors can occur if you don't correctly match calling conventions between the C++ prototype and the Object Pascal declaration, and calling convention problems are the most common obstacle for developers trying to share code between the two languages. To help clear things up, the following table shows the correspondence between Object Pascal and C++ calling convention directives.

Object Pascal	C++
`register*`	`__fastcall`
`pascal`	`__pascal`
`cdecl`	`__cdecl*`
`stdcall`	`__stdcall`

**Indicates the default calling convention for the language.*

Name Mangling

By default, the C++ compiler will mangle the names of functions not explicitly declared using the `extern "C"` modifier. The Object Pascal compiler, of course, doesn't mangle the names of functions. For example, Delphi's `TDUMP` utility reveals the exported symbol name of the `SAYHELLO` function shown earlier in `ccode.obj` as `@SAYHELLO$qqrpc`, whereas the name of the imported function according to Object Pascal is `SAYHELLO` (Object Pascal forces symbols to uppercase).

On the surface, this would seem to be a problem: How can the Delphi linker resolve the external if the function name isn't even the same? The answer is that the Delphi linker simply ignores the mangled portion (the `@` and everything after the `$`) of the symbol, but this can have some pretty nasty side effects.

The whole reason C++ mangles names is to allow function overloading (functions having the same names and different parameter lists). If you have a function that has several overloaded definitions and Delphi ignores the mangling portion of the symbol, you'll never know for sure whether Delphi is calling the overloaded function you want to call. Because of these complexities, we recommend that you don't attempt to call overloaded functions through object files.

NOTE

Functions in a C++ source file (`.CPP`) will always be mangled unless the prototypes are combined with the `extern "C"` modifier or the proper command-line switch is used on the C++ compiler to suppress name mangling.

Sharing Data

As mentioned earlier, it's possible to access Delphi data from the object file. The first step is to declare a global variable in your Object Pascal source similar to the variable shown here (note the underscore):

```
var
  _GLOBALVAR: PChar = 'This is a Delphi String';
```

Note that although the variable is initialized, this isn't a requirement.

In the C++ module, declare a variable of the same name using the external modifier, as shown here:

```
extern char * GLOBALVAR;
```

CAUTION

The default behavior of the Borland C++ compiler is to prepend external variables with an underscore when generating the external symbol (that is, `GLOBALVAR` becomes `_GLOBALVAR`). You can get around this in one of two ways:

- Use the command-line switch to disable the addition of the underscore (`-u-` with Borland C++ compilers).
- Place an underscore in front of the variable name in the Object Pascal code.

Although it's not possible to directly share data declared in an OBJ file with Object Pascal code, it is possible to trick Object Pascal into accessing OBJ-based data. The first step is to declare the data you want to export in your C++ code using the `__export` directive. For example, you would make a `char` array available for export like this:

```
char __export C_VAR[128];
```

Next (here comes part one of the trick), you declare this data as an external procedure in your Object Pascal code as follows (note, again, the underscore):

```
procedure _C_VAR; external;  // trick to import OBJ data
```

This will allow the linker to resolve references to `_C_VAR` in your Pascal code. Finally (here's the second part of the trick), you can use `_C_VAR` in your Pascal code as a pointer to the data. For example, the following code can be used to get the value of the array:

```
type
  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;

function GetCArray: string;
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  Result := A^;
end;
```

And the following code can be used to set the value of the array:

```
procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
```

```
  A := PCharArray(@_C_VAR);
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;
```

Using the Delphi RTL

It can be difficult to link an object file to your Delphi application if the object file contains references to the C++ RTL. This is because the C++ RTL generally lives in LIB files, and Delphi doesn't have the capability to link with LIB files.

How do you get around this problem? One way is to cut the definitions of the external functions you use out of the C++ RTL source code and place it in your object file. However, unless you're calling only one or two external functions, this type of solution will get mighty complex—not to mention the fact that your object file will become huge.

A more elegant solution to this problem is to create one or more header files that redeclare all the RTL functions you call using the `external` modifier and actually implement these functions inside your Object Pascal code. For example, let's say you want to call the `MessageBox()` API function from your C++ code. Normally, this would require you to use the `#include` preprocessor directive to include `windows.h` and link with the necessary Win32 libraries. However, redefining `MessageBox()` in your C++ code like this

```
extern int __stdcall MessageBox(long, char *, char *, long);
```

will cause the Object Pascal linker to search for a function of its own called `MessageBox` when it builds the executable. Of course, there's a function of that name defined in the Windows unit. Now your application will happily compile and link without a hitch.

Listing 13.9 shows a complete example of everything we've talked about so far. It's a fairly simple C module called `ccode.c`.

LISTING 13.9 A Simple C++ Module: `ccode.c`

```
#include "PasStng.h"

// globals
extern char * GLOBALVAR;

// exported data
char __export C_VAR[128];

#ifdef __cplusplus
extern "C" {
#endif
```

continues

LISTING 13.9 Continued

```
//externals
extern int __stdcall MessageBox(long, char *, char *, long);

//functions
int __export __cdecl SAYHELLO(char * hellostr)
{
  char a[64];
  memset(a, 64, 0);
  strcat(a, hellostr);
  strcat(a, " from Borland C++Builder");
  MessageBox(0, a, GLOBALVAR, 0);
  return 0;
}

#ifdef __cplusplus
}     // end of extern "C"
#endif
```

In addition to `MessageBox()`, notice the calls that this module makes to the `memset()` and `strcat()` C++ RTL functions. These functions are handled similarly in the `PasStng.h` header file, which contains some of the more common functions from the `string.h` header. This file is shown in Listing 13.10.

LISTING 13.10 `PasStng.h`, C++ `string.h` Emulation for Pascal

```
// PasStng.h
// This module externalizes a portion of the string.h C++ RTL header so
// that the Object Pascal RTL can instead handle the calls.

#ifndef PASSTNG_H
#define PASSTNG_H

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned size_t;
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern char * __cdecl strcat(char *dest, const char *src);
extern int __cdecl stricmp(const char *s1, const char *s2);
extern size_t __cdecl strlen(const char *s);
```

```
extern char * __cdecl strlwr(char *s);
extern char * __cdecl strncat(char *dest, const char *src,
  size_t maxlen);
extern void * __cdecl memcpy(void *dest, const void *src, size_t n);
extern int __cdecl strncmp(const char *s1, const char *s2,
  size_t  maxlen);
extern int __cdecl strncmpi(const char *s1, const char *s2, size_t n);
extern void * __cdecl memmove(void *dest, const void *src, size_t n);
extern char * __cdecl strncpy(char *dest, const char *src,
  size_t maxlen);
extern void * __cdecl memset(void *s, int c, size_t n);
extern int __cdecl strnicmp(const char *s1, const char *s2,
  size_t maxlen);
extern void __cdecl movmem(const void *src, void *dest, unsigned length);
extern void __cdecl setmem(void *dest, unsigned length, char value);
extern char * __cdecl stpcpy(char *dest, const char *src);
extern int __cdecl strcmp(const char *s1, const char *s2);
extern char * __cdecl strstr(char *s1, const char *s2);
extern int __cdecl strcmpi(const char *s1, const char *s2);
extern char * __cdecl strupr(char *s);
extern char * __cdecl strcpy(char *dest, const char *src);

#ifdef __cplusplus
}       // end of extern "C"
#endif

#endif  // PASSTNG_H
```


Because these functions don't exist in the Object Pascal RTL, we can work around the problem by creating an Object Pascal unit to include in our project that maps these functions to their Object Pascal counterparts. This unit, `PasStrng.pas`, is shown in Listing 13.11.

LISTING 13.11 `PasStrng.pas`, an Implementation of `string.h` Emulation Functions

```
unit PasStrng;

interface

uses Windows;

function _strcat(Dest, Source: PChar): PChar; cdecl;
procedure _memset(P: Pointer; Count: Integer; value: DWORD); cdecl;
function _stricmp(P1, P2: PChar): Integer; cdecl;
function _strlen(P1: PChar): Integer; cdecl;
function _strlwr(P1: PChar): PChar; cdecl;
```

continues

LISTING 13.11 Continued

```
function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
procedure _movmem(Source, Dest: Pointer; MaxLen: Integer); cdecl;
procedure _setmem(Dest: Pointer; Len: Integer; Value: Char); cdecl;
function _stpcpy(Dest, Source: PChar): PChar; cdecl;
function _strcmp(P1, P2: PChar): Integer; cdecl;
function _strstr(P1, P2: PChar): PChar; cdecl;
function _strcmpi(P1, P2: PChar): Integer; cdecl;
function _strupr(P: PChar): PChar; cdecl;
function _strcpy(Dest, Source: PChar): PChar; cdecl;

implementation

uses SysUtils;

function _strcat(Dest, Source: PChar): PChar;
begin
  Result := SysUtils.StrCat(Dest, Source);
end;

function _stricmp(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strlen(P1: PChar): Integer;
begin
  Result := SysUtils.StrLen(P1);
end;

function _strlwr(P1: PChar): PChar;
begin
  Result := StrLower(P1);
end;

function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCat(Dest, Source, MaxLen);
end;
```

```
function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;

function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLComp(P1, P2, MaxLen);
end;

function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;

function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCopy(Dest, Source, MaxLen);
end;

procedure _memset(P: Pointer; Count: Integer; Value: DWORD);
begin
  FillChar(P^, Count, Value);
end;

function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

procedure _movmem(Source, Dest: Pointer; MaxLen: Integer);
begin
  Move(Source^, Dest^, MaxLen);
end;

procedure _setmem(Dest: Pointer; Len: Integer; Value: Char);
begin
  FillChar(Dest^, Len, Value);
end;
```

continues

LISTING 13.11 Continued

```
function _stpcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;

function _strcmp(P1, P2: PChar): Integer;
begin
  Result := StrComp(P1, P2);
end;

function _strstr(P1, P2: PChar): PChar;
begin
  Result := StrPos(P1, P2);
end;

function _strcmpi(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strupr(P: PChar): PChar;
begin
  Result := StrUpper(P);
end;

function _strcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;

end.
```

TIP

Using the technique shown here, you could externalize more of the C++ RTL and Win32 API into header files that map to Object Pascal units.

Using C++ Classes

Although it's impossible to use C++ classes contained in an object file, it's possible to get some limited use from C++ classes contained in DLLs. By "limited use," we mean that you'll be able to call the virtual functions exposed by the C++ class only from the Delphi side. This

is possible because both Object Pascal and C++ follow the COM standard for virtual interfaces (see Chapter 23, "COM and ActiveX").

Listing 13.12 shows the source code for `cdll.cpp`, a C++ module that contains a class definition. Notice in particular the standalone functions—one of which creates and returns a reference to a new object, and another of which frees a given reference. These functions are the conduits through which we'll share the object between the languages.

LISTING 13.12 `cdll.cpp`: A C++ Module That Contains a Class Definition

```
#include <windows.h>

// objects
class TFoo
{
  virtual int function1(char *);
  virtual int function2(int);
};

//member functions
int TFoo::function1(char * str1)
{
  MessageBox(NULL, str1, "Hello from C++ DLL", MB_OK);
  return 0;
}

int TFoo::function2(int i)
{
  return i * i;
}

#ifdef __cplusplus
extern "C"  {
#endif

//prototypes
TFoo * __declspec(dllexport) ClassFactory(void);
void __declspec(dllexport) ClassKill(TFoo *);

TFoo * __declspec(dllexport) CLASSFACTORY(void)
{
  TFoo * Foo;
  Foo = new TFoo;
  return Foo;
}
```

continues

LISTING 13.12 Continued

```
void __declspec(dllexport) CLASSKILL(TFoo * Foo)
{
  delete Foo;
}

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}

#ifdef __cplusplus
}
#endif
```

To use this object from a Delphi application, you must do two things. First, you must import the functions that create and destroy class instances. Second, you must define a virtual abstract Object Pascal class definition that wraps the C++ class. Here's how to do that:

```
type
  TFoo = class
    function Function1(Str1: PChar): integer; virtual; cdecl; abstract;
    function Function2(i: integer): integer; virtual; cdecl; abstract;
  end;

function ClassFactory: TFoo; cdecl; external 'cdll.dll'
  name '_CLASSFACTORY';
procedure ClassKill(Foo: TFoo); cdecl; external 'cdll.dll' name
  '_CLASSKILL';
```

NOTE

When defining the Object Pascal wrapper for a C++ class, you don't need to worry about the names of the functions because they're unimportant in determining how the function is called internally. Because all calls will be dispatched through the Virtual Method Table, the order in which the functions are declared is key. Make sure that the order of the functions is the same in both the C++ and Object Pascal definitions.

Listing 13.13 shows `Main.pas`, the main unit for the `CallC.dpr` project, which demonstrates all the C++ techniques shown so far in this chapter. The main form for this project is shown in Figure 13.5.

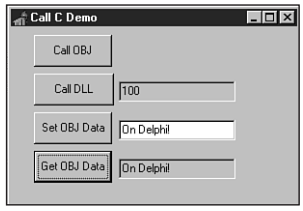

FIGURE 13.5
The main form for the CallC project.

LISTING 13.13 `Main.pas`, the Main Unit for the `CallC` Project

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Button1: TButton;
    Button2: TButton;
    FooData: TEdit;
    Button3: TButton;
    Button4: TButton;
    SetCVarData: TEdit;
    GetCVarData: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;
  _GlobalVar: PChar = 'This is a Delphi String';

implementation
```

continues

LISTING 13.13 Continued

```
uses PasStrng;

{$R *.DFM}

{$L ccode.obj}

type
  TFoo = class
    function Function1(Str1: PChar): integer; virtual; cdecl; abstract;
    function Function2(i: integer): integer; virtual; cdecl; abstract;
  end;

  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;

// import from OBJ file:
function _SAYHELLO(Text: PChar): Integer; cdecl; external;
procedure _C_VAR; external;  // trick to import OBJ data

// imports from DLL file:
function ClassFactory: TFoo; cdecl; external 'cdll.dll'
  name '_CLASSFACTORY';
procedure ClassKill(Foo: TFoo); cdecl; external 'cdll.dll'
  name '_CLASSKILL';

procedure TMainForm.Button1Click(Sender: TObject);
begin
  _SayHello('hello world');
end;

procedure TMainForm.Button2Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := ClassFactory;
  Foo.Function1('huh huh, cool.');
  FooData.Text := IntToStr(Foo.Function2(10));
  ClassKill(Foo);
end;

function GetCArray: string;
var
  A: PCharArray;
begin
```

```
  A := PCharArray(@_C_VAR);
  Result := A^;
end;

procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;

procedure TMainForm.Button3Click(Sender: TObject);
begin
  SetCArray(SetCVarData.Text);
end;

procedure TMainForm.Button4Click(Sender: TObject);
begin
  GetCVarData.Text := GetCArray;
end;

end.
```


TIP

Although the technique demonstrated here does allow a limited means for communicating with C++ classes from Object Pascal, if you want to do this type of thing on a large scale, we recommend you use COM objects to communicate between languages, as described in Chapter 23.

Thunking

At some point in your development of Windows and Win32 applications, you'll need to call 16-bit code from a 32-bit application or even 32-bit code from a 16-bit application. This process is known as *thunking*. Although the different varieties of Win32 provide various facilities to make this possible, it remains one of the more difficult tasks to accomplish when developing Windows applications.

TIP

Aside from thunking, you should know that Automation (described in Chapter 23) provides a reasonable alternative for crossing 16/32-bit boundaries. This capability is built into Automation's `IDispatch` interface.

Win32 provides three different types of thunking: universal, generic, and flat. Each of these techniques has its advantages and drawbacks:

- *Universal thunking* is available only under the Win32s platform (Win32s is the Win32 API subset available under 16-bit Windows). It allows 16-bit applications to load and call Win32 DLLs. Because this variety of thunking is supported only for Win32s, a platform not officially supported by Delphi, we won't devote any more discussion to this topic.
- *Generic thunking* enables 16-bit Windows applications to call Win32 DLLs under Windows 95, 98, NT, and 2000. This is the most flexible type of thunking because it's available on all major Win32 platforms and is API-based. We'll discuss this option in detail shortly.
- *Flat thunking* allows Win32 applications to call 16-bit DLLs and 16-bit applications to call Win32 DLLs. Unfortunately, this type of thunking is available only under Windows 95/98; it also requires the use of a thunk compiler to create object files, which must be linked to both the 32-bit and 16-bit sides. Because of the lack of portability and requirement for additional tools, we won't cover flat thunking here.

In addition, there's a way to share data between 32-bit and 16-bit processes by using the `WM_COPYDATA` Windows message. In particular, `WM_COPYDATA` provides a straightforward means for accessing 16-bit code from Windows NT/2000 (where thunking can be a headache), so we'll also cover that in this section.

Generic Thunking

Generic thunking is facilitated through a set of APIs that sit on both the 16-bit and 32-bit sides. These APIs are known as `WOW16` and `WOW32`, respectively. From 16-bit land, `WOW16` provides functions that allow you to load the Win32 DLL, get the address of functions in the DLL, and call those functions. The source code for the `WOW16.pas` unit is shown in Listing 13.14.

LISTING 13.14 `WOW16.pas`, Functions to Load a 32-bit DLL from a 16-bit Application

```
unit WOW16;
// Unit which provides an interface to the 16-bit Windows on Win32 (WOW)
// API from a 16-bit application running under Win32.
```

```
// These functions allow 16-bit applications to call 32-bit DLLs.
// Copyright (c) 1996, 1999 Steve Teixeira and Xavier Pacheco

interface

uses WinTypes;

type
  THandle32 = Longint;
  DWORD = Longint;


{ Win32 module management.}

{ The following routines accept parameters that correspond directly  }
{ to the respective Win32 API function calls that they invoke. Refer }
{ to the Win32 reference documentation for more detail.              }
function LoadLibraryEx32W(LibFileName: PChar; hFile, dwFlags: DWORD):
  THandle32;
function FreeLibrary32W(LibModule: THandle32): BOOL;
function GetProcAddress32W(Module: THandle32; ProcName: PChar): TFarProc;

{ GetVDMPointer32W converts a 16-bit (16:16) pointer into a       }
{ 32-bit flat (0:32) pointer. The value of FMode should be 1 if   }
{ the 16-bit pointer is a protected mode address (the normal      }
{ situation in Windows 3.x) or 0 if the 16-bit pointer is real    }
{ mode.                                                          }
{ NOTE:  Limit checking is not performed in the retail build      }
{ of Windows NT.  It is performed in the checked (debug) build    }
{ of WOW32.DLL, which will cause 0 to be returned when the        }
{ limit is exceeded by the supplied offset.                       }
function GetVDMPointer32W(Address: Pointer; fProtectedMode: WordBool):
  DWORD;

{ CallProc32W calls a proc whose address was retrieved by         }
{ GetProcAddress32W. The true definition of this function         }
{ actually allows for multiple DWORD parameters to be passed      }
{ prior to the ProcAddress parameter, and the nParams parameter   }
{ should reveal the number of params passed prior to ProcAddress. }
{ The AddressConvert parameter is a bitmask which indicates which }
{ of the params are 16-bit pointers in need of conversion before  }
{ the 32-bit function is called. Since this function doesn't lend }
{ itself to being defined in Object Pascal, you may want to use   }
{ the simplified Call32BitProc function instead. }
function CallProc32W(Params: DWORD; ProcAddress, AddressConvert,
                     nParams: DWORD): DWORD;
```

continues

LISTING 13.14 Continued

```
{ Call32BitProc accepts a constant array of Longints as the parameter }
{ list for the function given by ProcAddress. This procedure is       }
{ responsible for packaging the parameters into the correct format    }
{ and calling the CallProc32W WOW function. }
function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
                       AddressConvert: Longint): DWORD;

{ Converts a 16-bit window handle to 32-bit for use by Windows NT. }
function HWnd16To32(Handle: hWnd): THandle32;

{ Converts a 32-bit window handle to 16-bit. }
function HWnd32To16(Handle: THandle32): hWnd;

implementation

uses WinProcs;

function HWnd16To32(Handle: hWnd): THandle32;
begin
  Result := Handle or $FFFF0000;
end;

function HWnd32To16(Handle: THandle32): hWnd;
begin
  Result := LoWord(Handle);
end;

function BitIsSet(Value: Longint; Bit: Byte): Boolean;
begin
  Result := Value and (1 shl Bit) <> 0;
end;

procedure FixParams(var Params: array of Longint; AddConv: Longint);
var
  i: integer;
begin
  for i := Low(Params) to High(Params) do
    if BitIsSet(AddConv, i) then
      Params[i] := GetVDMPointer32W(Pointer(Params[i]), True);
end;

function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
                       AddressConvert: Longint): DWORD;
```

```
var
  NumParams: word;
begin
  FixParams(Params, AddressConvert);
  NumParams := High(Params) + 1;
  asm
    les di, Params              { es:di -> Params }
    mov cx, NumParams           { loop counter = num params }
  @@1:
    push es:word ptr [di + 2]   { push hiword of param x }
    push es:word ptr [di]       { push loword of param x }
    add di, 4                   { skip to next param }
    loop @@1                    { iterate over all params }
    mov cx, ProcAddress.Word[2] { cx = hiword of ProcAddress }
    mov dx, ProcAddress.Word[0] { dx = loword of ProcAddress }
    push cx                     { push hi ProcAddress }
    push dx                     { push lo ProcAddress }
    mov ax, 0
    push ax                     { push dummy hi AddressConvert }
    push ax                     { push dummy lo AddressConvert }
    push ax                     { push hi NumParams }
    mov cx, NumParams
    push cx                     { push lo Number of Params }
    call CallProc32W            { call function }
    mov Result.Word[0], ax
    mov Result.Word[2], dx      { store return value }
  end
end;

{ 16-bit WOW functions }
function LoadLibraryEx32W;             external 'KERNEL' index 513;
function FreeLibrary32W;               external 'KERNEL' index 514;
function GetProcAddress32W;            external 'KERNEL' index 515;
function GetVDMPointer32W;             external 'KERNEL' index 516;
function CallProc32W;                  external 'KERNEL' index 517;

end.
```

All the functions in this unit are simply exports from the 16-bit kernel except for the `Call32BitProc()` function, which employs some assembly code to allow the user to pass a variable number of parameters in an array of `Longint`.

The `WOW32` functions make up the `WOW32.pas` unit, which is shown in Listing 13.15.

LISTING 13.15 WOW32.pas, Interface for WOW32.dll, Which Provides Access to 16-bit code from Win32 Applications

```
unit WOW32;
// Import of WOW32.DLL, which provides utilities for accessing
// 16-bit code from Win32.
// Copyright (c) 1996, 1999 Steve Teixeira and Xavier Pacheco

interface

uses Windows;

//
// 16:16 -> 0:32 Pointer translation.
//
// WOWGetVDMPointer will convert the passed in 16-bit address
// to the equivalent 32-bit flat pointer.  If fProtectedMode
// is TRUE, the function treats the upper 16 bits as a selector
// in the local descriptor table.  If fProtectedMode is FALSE,
// the upper 16 bits are treated as a real-mode segment value.
// In either case the lower 16 bits are treated as the offset.
//
// The return value is 0 if the selector is invalid.
//
// NOTE:  Limit checking is not performed in the retail build
// of Windows NT.  It is performed in the checked (debug) build
// of WOW32.DLL, which will cause 0 to be returned when the
// limit is exceeded by the supplied offset.
//
function WOWGetVDMPointer(vp, dwBytes: DWORD; fProtectedMode: BOOL):
  Pointer; stdcall;

//
// The following two functions are here for compatibility with
// Windows 95.  On Win95, the global heap can be rearranged,
// invalidating flat pointers returned by WOWGetVDMPointer, while
// a thunk is executing.  On Windows NT, the 16-bit VDM is completely
// halted while a thunk executes, so the only way the heap will
// be rearranged is if a callback is made to Win16 code.
//
// The Win95 versions of these functions call GlobalFix to
// lock down a segment's flat address, and GlobalUnfix to
// release the segment.
//
// The Windows NT implementations of these functions do *not*
// call GlobalFix/GlobalUnfix on the segment, because there
// will not be any heap motion unless a callback occurs.
```

```
// If your thunk does callback to the 16-bit side, be sure
// to discard flat pointers and call WOWGetVDMPointer again
// to be sure the flat address is correct.
//
function WOWGetVDMPointerFix(vp, dwBytes: DWORD; fProtectedMode: BOOL):
  Pointer; stdcall;
procedure WOWGetVDMPointerUnfix(vp: DWORD); stdcall;

//
// Win16 memory management.
//
// These functions can be used to manage memory in the Win16
// heap.  The following four functions are identical to their
// Win16 counterparts, except that they are called from Win32
// code.
//
function WOWGlobalAlloc16(wFlags: word; cb: DWORD): word; stdcall;
function WOWGlobalFree16(hMem: word): word; stdcall;
function WOWGlobalLock16(hMem: word): DWORD; stdcall;
function WOWGlobalUnlock16(hMem: word): BOOL; stdcall;

//
// The following three functions combine two common operations in
// one switch to 16-bit mode.
//
function WOWGlobalAllocLock16(wFlags: word; cb: DWORD; phMem: PWord):
  DWORD; stdcall;
function WOWGlobalLockSize16(hMem: word; pcb: PDWORD): DWORD; stdcall;
function WOWGlobalUnlockFree16(vpMem: DWORD): word; stdcall;

//
// Yielding the Win16 nonpreemptive scheduler
//
// The following two functions are provided for Win32 code called
// via Generic Thunks which needs to yield the Win16 scheduler so
// that tasks in that VDM can execute while the thunk waits for
// something to complete.  These two functions are functionally
// identical to calling back to 16-bit code which calls Yield or
// DirectedYield.
//
procedure WOWYield16;
procedure WOWDirectedYield16(htask16: word);

//
// Generic Callbacks.
//
```

continues

LISTING 13.15 Continued

```
// WOWCallback16 can be used in Win32 code called
// from 16-bit (such as by using Generic Thunks) to call back to
// the 16-bit side.  The function called must be declared similarly
// to the following:
//
// function CallbackRoutine(dwParam: Longint): Longint; export;
//
// If you are passing a pointer, declare the parameter as such:
//
// function CallbackRoutine(vp: Pointer): Longint; export;
//
// NOTE: If you are passing a pointer, you'll need to get the
// pointer using WOWGlobalAlloc16 or WOWGlobalAllocLock16
//
// If the function called returns a word instead of a Longint, the
// upper 16 bits of the return value is undefined.  Similarly, if
// the function called has no return value, the entire return value
// is undefined.
//
// WOWCallback16Ex allows any combination of arguments up to
// WCB16_MAX_CBARGS bytes total to be passed to the 16-bit routine.
// cbArgs is used to properly clean up the 16-bit stack after calling
// the routine.  Regardless of the value of cbArgs, WCB16_MAX_CBARGS
// bytes will always be copied from pArgs to the 16-bit stack.  If
// pArgs is less than WCB16_MAX_CBARGS bytes from the end of a page,
// and the next page is inaccessible, WOWCallback16Ex will incur an
// access violation.
//
// If cbArgs is larger than the WCB16_MAX_ARGS which the running
// system supports, the function returns FALSE and GetLastError
// returns ERROR_INVALID_PARAMETER.  Otherwise the function
// returns TRUE and the DWORD pointed to by pdwRetCode contains
// the return code from the callback routine.  If the callback
// routine returns a WORD, the HIWORD of the return code is
// undefined and should be ignored using LOWORD(dwRetCode).
//
// WOWCallback16Ex can call routines using the PASCAL and CDECL
// calling conventions.  The default is to use the PASCAL
// calling convention.  To use CDECL, pass WCB16_CDECL in the
// dwFlags parameter.
//
// The arguments pointed to by pArgs must be in the correct
// order for the callback routine's calling convention.
// To call the routine SetWindowText,
//
```

```
// SetWindowText(Handle: hWnd; lpsz: PChar): Longint;
//
// pArgs would point to an array of words:
//
// SetWindowTextArgs: array[0..2] of word =
//     (LoWord(Longint(lpsz)), HiWord(Longint(lpsz)), Handle);
//
// In other words, the arguments are placed in the array in reverse
// order with the least significant word first for DWORDs and offset
// first for FAR pointers.  Further, the arguments are placed in the
// array in the order listed in the function prototype with the least
// significant word first for DWORDs and offset first for FAR pointers.
//
function WOWCallback16(vpfn16, dwParam: DWORD): DWORD; stdcall;

const
  WCB16_MAX_CBARGS = 16;
  WCB16_PASCAL     = $0;
  WCB16_CDECL      = $1;

function WOWCallback16Ex(vpfn16, dwFlags, cbArgs: DWORD; pArgs: Pointer;
  pdwRetCode: PDWORD): BOOL; stdcall;

//
// 16 <—> 32 Handle mapping functions.
//
type
  TWOWHandleType = (
    WOW_TYPE_HWND,
    WOW_TYPE_HMENU,
    WOW_TYPE_HDWP,
    WOW_TYPE_HDROP,
    WOW_TYPE_HDC,
    WOW_TYPE_HFONT,
    WOW_TYPE_HMETAFILE,
    WOW_TYPE_HRGN,
    WOW_TYPE_HBITMAP,
    WOW_TYPE_HBRUSH,
    WOW_TYPE_HPALETTE,
    WOW_TYPE_HPEN,
    WOW_TYPE_HACCEL,
    WOW_TYPE_HTASK,
    WOW_TYPE_FULLHWND);

function WOWHandle16(Handle32: THandle; HandType: TWOWHandleType): Word;
  stdcall;
```

continues

LISTING 13.15 Continued

```
function WOWHandle32(Handle16: word; HandleType: TWOWHandleType):
  THandle; stdcall;

implementation

const
  WOW32DLL = 'WOW32.DLL';

function WOWCallback16;
  external WOW32DLL name 'WOWCallback16';
function WOWCallback16Ex;
  external WOW32DLL name 'WOWCallback16Ex';
function WOWGetVDMPointer;
  external WOW32DLL name 'WOWGetVDMPointer';
function WOWGetVDMPointerFix;
  external WOW32DLL name 'WOWGetVDMPointerFix';
procedure WOWGetVDMPointerUnfix;
  external WOW32DLL name 'WOWGetVDMPointerUnfix'
function WOWGlobalAlloc16;
  external WOW32DLL name 'WOWGlobalAlloc16'
function WOWGlobalAllocLock16;
  external WOW32DLL name 'WOWGlobalAllocLock16';
function WOWGlobalFree16;
  external WOW32DLL name 'WOWGlobalFree16';
function WOWGlobalLock16;
  external WOW32DLL name 'WOWGlobalLock16';
function WOWGlobalLockSize16;
  external WOW32DLL name 'WOWGlobalLockSize16';
function WOWGlobalUnlock16;
  external WOW32DLL name 'WOWGlobalUnlock16';
function WOWGlobalUnlockFree16;
  external WOW32DLL name 'WOWGlobalUnlockFree16';
function WOWHandle16;
  external WOW32DLL name 'WOWHandle16';
function WOWHandle32;
  external WOW32DLL name 'WOWHandle32';
procedure WOWYield16;
  external WOW32DLL name 'WOWYield16';
procedure WOWDirectedYield16;
  external WOW32DLL name 'WOWDirectedYield16';

end.
```

To illustrate generic thunking, we'll create a small 32-bit DLL that will be called from a 16-bit executable. The 32-bit DLL project, `TestDLL.dpr`, is shown in Listing 13.16.

LISTING 13.16 TestDLL.dpr, DLL Project for Testing Generic Thunking. -s

```
library TestDLL;

uses
  SysUtils, Dialogs, Windows, WOW32;

const
  DLLStr = 'I am in the 32-bit DLL. The string you sent is: "%s"';

function DLLFunc32(P: PChar; CallBackFunc: DWORD): Integer; stdcall;
const
  MemSize = 256;
var
  Mem16: DWORD;
  Mem32: PChar;
  Hand16: word;
begin
  { Show string P }
  ShowMessage(Format(DLLStr, [P]));
  { Allocate some 16-bit memory }
  Hand16 := WOWGlobalAlloc16(GMem_Share or GMem_Fixed or GMem_ZeroInit,
                             MemSize);
  { Lock the 16-bit memory }
  Mem16 := WOWGlobalLock16(Hand16);
  { Convert 16-bit pointer to 32-bit pointer.  Now they point to the }
  { same place. }
  Mem32 := PChar(WOWGetVDMPointer(Mem16, MemSize, True));
  { Copy string into 32-bit pointer }
  StrPCopy(Mem32, 'I REALLY love DDG!!');
  { Call back into the 16-bit app, passing 16-bit pointer }
  Result := WOWCallback16(CallBackFunc, Mem16);
  { clean up allocated 16-bit memory }
  WOWGlobalUnlockFree16(Mem16);
end;

exports
  DLLFunc32 name 'DLLFunc32' resident;

begin
end.
```

This DLL exports one function that takes a `PChar` and a callback function as parameters. The `PChar` is immediately displayed in a `ShowMessage()`. The callback function allows the function to call back into the 16-bit process, passing some specially allocated 16-bit memory.

The code for the 16-bit application, `Call32.dpr`, is shown in Listing 13.17. The main form is shown in Figure 13.6.

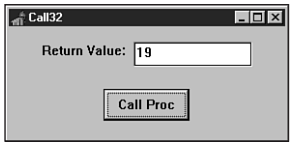

FIGURE 13.6
The Call32 main form.

LISTING 13.17 `Main.pas`, the Main Unit for the 16-bit Portion of the Generic Thunking Test Application

```
unit Main;
{$C FIXED DEMANDLOAD PERMANENT}

interface

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

type
  TMainForm = class(TForm)
    CallBtn: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure CallBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
```

```
uses WOW16;

const
  ExeStr = 'The 32-bit DLL has called back into the 16-bit EXE. ' +
           'The string to the EXE is: "%s"';

function CallBackFunc(P: PChar): Longint; export;
begin
  ShowMessage(Format(ExeStr, [StrPas(P)]));
  Result := StrLen(P);
end;

procedure TMainForm.CallBtnClick(Sender: TObject);
var
  H: THandle32;
  R, P: Longint;
  AStr: PChar;
begin
  { load 32-bit DLL }
  H := LoadLibraryEx32W('TestDLL.dll', 0, 0);
  AStr := StrNew('I love DDG.');
  try
    if H > 0 then
    begin
      { Retrieve address of proc from 32-bit DLL }
      TFarProc(P) := GetProcAddress32W(H, 'DLLFunc32');
      if P > 0 then
      begin
        { Call proc in 32-bit DLL }
        R := Call32BitProc(P, [Longint(AStr), Longint(@CallBackFunc)],
          1);
        Edit1.Text := IntToStr(R);
      end;
    end;
  finally
    StrDispose(AStr);
    if H > 0 then FreeLibrary32W(H);
  end;
end;

end.
```


This application passes a 16-bit `PChar` and function address to the 32-bit DLL. `CallBackFunc()` is eventually called by the 32-bit DLL. In fact, if you look closely, the return value of `DLLFunc32()` is the value returned by `CallBackFunc()`.

WM_COPYDATA

Windows 95/98 supports flat thunks to call 16-bit DLLs from Win32 applications. Windows NT/2000 doesn't provide a means to directly call 16-bit code from a Win32 application. Given this limitation, the question that follows is, what's the best way to communicate data between 32-bit and 16-bit processes on NT? What's more, that leads us to another question: Is there an easy way to share data in such a way that it runs under all the major Win32 platforms, Windows 95, 98, NT, and 2000?

The answer to both questions is `WM_COPYDATA`. The `WM_COPYDATA` Windows message provides a means for transferring binary data between processes—whether 32-bit or 16-bit processes. When a `WM_COPYDATA` message is sent to a window, the `wParam` of this message identifies the window passing the data, and the `lParam` holds a pointer to a `TCopyDataStruct` record. This record is defined as follows:

```
type
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = packed record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
  end;
```

The `dwData` field holds 32 bits of user-defined information. `cbData` contains the size of the buffer pointed to by `lpData`. `lpData` is a pointer to a buffer of information you want to pass between applications. If you send this message between a 32-bit and a 16-bit application, Windows will automatically convert the `lpData` pointer from a 0:32 pointer to a 16:16 pointer, or vice versa. Additionally, Windows will ensure that the data pointed to by `lpData` is mapped into the receiving process's address space.

NOTE

`WM_COPYDATA` works great for relatively small amounts of information, but if you have a lot of information that you must communicate across the 16/32-bit boundary, you may wish to do so using Automation, which has the built-in ability to marshal across process boundaries. Automation is described in Chapter 23.

TIP

It should be clear that, although NT doesn't support direct usage of 16-bit DLLs from Win32 applications, you can create a 16-bit executable that encapsulates the DLL and can communicate with that executable by using `WM_COPYDATA`.

To show how WM_COPYDATA works, we'll create two projects, the first being a 32-bit application. This application will have a memo control into which you can type some text. Additionally, this application will provide a means for communicating with the second project, a 16-bit application, to transfer memo text. To provide a means whereby the two applications can begin communication, take the following steps:

1. Register a window message to obtain a unique message ID for interapplication communication.
2. Broadcast the message system-wide from the Win32 application. In the wParam of this message, store the handle to the main window of the Win32 application.
3. When the 16-bit application receives the broadcast message, it will answer by sending the registered message back to the sending application and pass its own main form's window handle as the wParam.
4. After receiving the response, the 32-bit application now has the handle to the main form of the 16-bit application. The 32-bit application can now send a WM_COPYDATA message to the 16-bit application so that the sharing can begin.

The code for the RegMsg.pas unit, which is shared by the two projects, is shown in Listing 13.18.

LISTING 13.18 RegMsg.pas, the Unit Which Registers the Handshaking Message

```
unit RegMsg;

interface

var
  DDGM_HandshakeMessage: Cardinal;

implementation

uses WinProcs;

const
  HandshakeMessageStr: PChar = 'DDG.CopyData.Handshake';


initialization
  DDGM_HandshakeMessage := RegisterWindowMessage(HandshakeMessageStr);
end.
```

The source code for CopyMain.pas, the main unit of the 32-bit CopyData.dpr project, is shown in Listing 13.19. This is the unit that establishes the conversation and sends the data.

LISTING 13.19 CopyMain.pas, the Main Unit for the 32-bit Portion of the WM_COPYDATA Demonstration

```
unit CopyMain;

interface

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

type
  TMainForm = class(TForm)
    DataMemo: TMemo;
    BottomPnl: TPanel;
    BtnPnl: TPanel;
    CloseBtn: TButton;
    CopyBtn: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    CopyData1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    procedure CloseBtnClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure CopyBtnClick(Sender: TObject);
  private
    { Private declarations }
  protected
    procedure WndProc(var Message: TMessage); override;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses AboutU, RegMsg;

// The following declaration is necessary because of an error in
// the declaration of BroadcastSystemMessage() in the Windows unit
```

```
function BroadcastSystemMessage(Flags: DWORD; Recipients: PDWORD;
  uiMessage: UINT; wParam: WPARAM; lParam: LPARAM): Longint; stdcall;
  external 'user32.dll';

var
  Recipients: DWORD = BSM_APPLICATIONS;

procedure TMainForm.WndProc(var Message: TMessage);
var
  DataBuffer: TCopyDataStruct;
  Buf: PChar;
  BufSize: Integer;
begin
  if Message.Msg = DDGM_HandshakeMessage then begin
    { Allocate buffer }
    BufSize := DataMemo.GetTextLen + (1 * SizeOf(Char));
    Buf := AllocMem(BufSize);
    { Copy memo to buffer }
    DataMemo.GetTextBuf(Buf, BufSize);
    try
      with DataBuffer do begin
        { Fill dwData with registered message as safety check }
        dwData := DDGM_HandshakeMessage;
        cbData := BufSize;
        lpData := Buf;
      end;
      { NOTE: WM_COPYDATA message must be *sent* }
      SendMessage(Message.wParam, WM_COPYDATA, Handle,
        Longint(@DataBuffer));
    finally
      FreeMem(Buf, BufSize);
    end;
  end
  else
    inherited WndProc(Message);
end;

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

procedure TMainForm.FormResize(Sender: TObject);
begin
  BtnPnl.Left := BottomPnl.Width div 2 - BtnPnl.Width div 2;
end;
```

continues

LISTING 13.19 Continued

```
procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;

procedure TMainForm.CopyBtnClick(Sender: TObject);
begin
  { Call for any listening apps }
  BroadcastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @Recipients, DDGM_HandshakeMessage, Handle, 0);
end;

end.
```

The source for `ReadMain.pas`, the main unit for the 16-bit `ReadData.dpr` project, is shown in Listing 13.20. This is the unit that communicates with the CopyData project and receives the data buffer.

LISTING 13.20 `ReadMain.pas`, the Main Unit for the 16-bit Portion of the `WM_COPYDATA` Demonstration

```
unit Readmain;

interface

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

{ The WM_COPYDATA Windows message is not defined in the 16-bit Messages }
{ unit, although it is available to 16-bit applications running under   }
{ Windows 95 or NT.  This message is discussed in the Win32 API online  }
{ help. }
const
  WM_COPYDATA = $004A;

type
  TMainForm = class(TForm)
    ReadMemo: TMemo;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
```

```
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure OnAppMessage(var M: TMsg; var Handled: Boolean);
    procedure WMCopyData(var M: TMessage); message WM_COPYDATA;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses RegMsg, AboutU;

type
  { The TCopyDataStruct record type is not defined in WinTypes unit, }
  { although it is available in the 16-bit Windows API when running  }
  { under Windows 95 and NT. The lParam of the WM_COPYDATA message   }
  { points to one of these. }
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
  end;

procedure TMainForm.OnAppMessage(var M: TMsg; var Handled: Boolean);
{ OnMessage handler for Application object. }
begin
  { The DDGM_HandshakeMessage message is received as a broadcast to     }
  { all applications.  The wParam of this message contains the handle   }
  { of the window which broadcasted the message.  We respond by posting }
  { the same message back to the sender, with our handle in the wParam. }
  if M.Message = DDGM_HandshakeMessage then begin
    PostMessage(M.wParam, DDGM_HandshakeMessage, Handle, 0);
    Handled := True;
  end;
end;

procedure TMainForm.WMCopyData(var M: TMessage);
{ Handler for WM_COPYDATA message }
begin
  { Check wParam to ensure we know WHO sent us the WM_COPYDATA message }
```

continues

LISTING 13.20 Continued

```
  if PCopyDataStruct(M.lParam)^.dwData = DDGM_HandshakeMessage then
    { When WM_COPYDATA message is received, the lParam points to}
    ReadMemo.SetTextBuf(PChar(PCopyDataStruct(M.lParam)^.lpData));
end;

procedure  TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := OnAppMessage;
end;

procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;

end.
```

Figure 13.7 shows the two applications working in harmony.

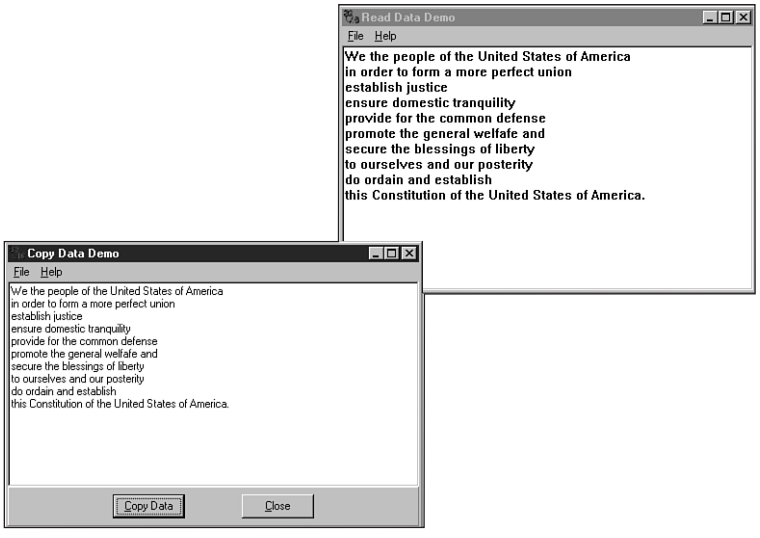

FIGURE 13.7
Communicating with `WM_COPYDATA`.

Obtaining Package Information

Packages are great. They provide a convenient means to logically and physically divide your application into separate modules. Packages are compiled binary modules consisting of one or more units, and they can reference units contained in other compiled packages. Of course, if you have the source code for a particular package, it's very easy to figure out what units are contained in that package and what other packages it requires. But what happens when you need to obtain that information for a package for which you have no source code? Fortunately, this is not terribly difficult if you don't mind writing a few lines of code. In fact, you can obtain this information with a call to only one procedure: `GetPackageInfo()`, which is contained in the `SysUtils` unit. `GetPackageInfo()` is declared as follows:

```
procedure GetPackageInfo(Module: HMODULE; Param: Pointer; var Flags: Integer;
  InfoProc: TPackageInfoProc);
```

`Module` is the Win32 API module handle of the package file, such as the handle returned by the `LoadLibrary()` API function.

`Param` is user-defined data that will be passed to the procedure specified by the `InfoProc` parameter.

Upon return, the `Flags` parameter will hold information about the package. This will become a combination of the flags shown in Table 13.6.

TABLE 13.6 `GetPackageInfo()` Flags

Flag	*Value*	*Meaning*
`pfNeverBuild`	`$00000001`	This is a "never build" package.
`pfDesignOnly`	`$00000002`	This is a design package.
`pfRunOnly`	`$00000004`	This is a run package.
`pfIgnoreDupUnits`	`$00000008`	Ignores multiple instances of the same unit in this package.
`pfModuleTypeMask`	`$C0000000`	The mask used to identify the module type.
`pfExeModule`	`$00000000`	The package module is an EXE (not used).
`pfPackageModule`	`$40000000`	The package module is a package file.
`pfProducerMask`	`$0C000000`	The mask used to identify the product that created this package.
`pfV3Produced`	`$00000000`	The package produced by Delphi 3 or BCB 3.
`pfProducerUndefined`	`$04000000`	The producer of this package is not defined.
`pfBCB4Produced`	`$08000000`	The packages were produced by BCB 4.
`pfDelphi4Produced`	`$0C000000`	The package was produced by Delphi 4.
`pfLibraryModule`	`$80000000`	The package module is a DLL.

The `InfoProc` parameter identifies a callback method that will be called once for each package this package requires and for each unit contained in this package. This parameter is of type `TPackageInfoProc`, which is defined as follows:

```
type
  TNameType = (ntContainsUnit, ntRequiresPackage);
  TPackageInfoProc = procedure (const Name: string; NameType: TNameType;
    Flags: Byte; Param: Pointer);
```

In this method type, `Name` identifies the name of the package or unit, `NameType` indicates whether this file is a package or a unit, `Flags` provides some additional information for the file, and `Param` contains the user-defined data originally passed to `GetPackageInfo()`.

To demonstrate the `GetPackageInfo()` procedure, what follows is a sample application used to obtain information for any package. This project is called PackInfo, and the project file is shown in Listing 13.21.

LISTING 13.21 `PackInfo.dpr`, the Project File for the Application

```
program PkgInfo;

uses
  Forms,
  Dialogs,
  SysUtils,
  PkgMain in 'PkgMain.pas' {PackInfoForm};

{$R *.RES}

var
  OpenDialog: TOpenDialog;
begin
  if (ParamCount > 0) and FileExists(ParamStr(1)) then
    PkgName := ParamStr(1)
  else begin
    OpenDialog := TOpenDialog.Create(Application);
    OpenDialog.DefaultExt := '*.bpl';
    OpenDialog.Filter := 'Packages (*.bpl)|*.bpl|Delphi 3 Packages ' +
      '(*.dpl)|*.dpl';
    if OpenDialog.Execute then PkgName := OpenDialog.FileName;
  end;
  if PkgName <> '' then
  begin
    Application.Initialize;
    Application.CreateForm(TPackInfoForm, PackInfoForm);
    Application.Run;
  end;
end.
```

If no command-line parameters are passed to this application, it immediately presents the user with a File Open dialog box in which the user can select a package file. If a package file is passed on the command line or if a file is selected in the dialog box, that filename is assigned to `PkgName`, and the application then runs normally.

The main unit for this application is shown in Listing 13.22. This is the unit that performs the call to `GetPackageInfo()`.

LISTING 13.22 `PkgMain.pas`, Obtaining Package Information

```
unit PkgMain;

interface

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

type
  TPackInfoForm = class(TForm)
    GroupBox1: TGroupBox;
    DsgnPkg: TCheckBox;
    RunPkg: TCheckBox;
    BuildCtl: TRadioGroup;
    GroupBox2: TGroupBox;
    GroupBox3: TGroupBox;
    Button1: TButton;
    Label1: TLabel;
    DescEd: TEdit;
    memContains: TMemo;
    memRequires: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  end;

var
  PackInfoForm: TPackInfoForm;
  PkgName: string;  // This is assigned in project file

implementation

{$R *.DFM}

procedure PackageInfoCallback(const Name: string; NameType: TNameType;
  Flags: Byte; Param: Pointer);
var
```

continues

LISTING 13.22 Continued

```
  AddName: string;
  Memo: TMemo;
begin
  Assert(Param <> nil);
  AddName := Name;
  case NameType of
    ntContainsUnit: Memo := TPackInfoForm(Param).memContains;
    ntRequiresPackage: Memo := TPackInfoForm(Param).memRequires;
  else
    Memo := nil;
  end;
  if Memo <> nil then
  begin
    if Memo.Text <> '' then AddName := ', ' + AddName;
    Memo.Text := Memo.Text + AddName;
  end;
end;

procedure TPackInfoForm.FormCreate(Sender: TObject);
var
  PackMod: HMODULE;
  Flags: Integer;
begin
  // Since we only need to get into the package's resources,
  // LoadLibraryEx with LOAD_LIBRARY_AS_DATAFILE provides a speed-
  // efficient means for loading the package.
  PackMod := LoadLibraryEx(PChar(PkgName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then Exit;
  try
    GetPackageInfo(PackMod, Pointer(Self), Flags, PackageInfoCallback);
  finally
    FreeLibrary(PackMod);
  end;
  Caption := 'Package Info: ' + ExtractFileName(PkgName);
  DsgnPkg.Checked := Flags and pfDesignOnly <> 0;
  RunPkg.Checked := Flags and pfRunOnly <> 0;
  if Flags and pfNeverBuild <> 0 then
    BuildCtl.ItemIndex := 1;
  DescEd.Text := GetPackageDescription(PChar(PkgName));
end;

procedure TPackInfoForm.Button1Click(Sender: TObject);
```

```
begin
  Close;
end;

end.
```

It seems as though there's a disproportionately small amount of code for this unit, considering the low-level information it obtains. When the form is created, the package is loaded, `GetPackageInfo()` is called, and some UI is updated. The `PackageInfoCallback()` method is passed in the `InfoProc` parameter of `GetPackageInfo()`. `PackageInfoCallback()` adds the package or unit name to the appropriate `TMemo` control. Figure 13.8 shows the PackInfo application displaying information for one of the Delphi packages.

FIGURE 13.8
Viewing package information with `PackInfo`.

Summary

Whew, this was an in-depth chapter! Step back for a moment and take a look at all you learned: subclassing window procedures, preventing multiple instances, windows hooks, BASM programming, using C++ object files, using C++ classes, thunking, `WM_COPYDATA`, and getting information for compiled packages. I don't know about you, but we've covered so much hacker stuff in this chapter that I'm hungry for Cheetos and Jolt Cola! Since we're on a roll with low-level programming, the next chapter, "Snooping System Information," details how to get inside the OS to obtain information about processes, threads, and modules.