Windows Shell Programming

CHAPTER

16

IN THIS CHAPTER

First introduced in Windows 95, the Windows shell is also supported on all subsequent Windows versions (NT 3.51 and higher, 98, 2000, Me, and XP). A far cry from the old Program Manger, the Windows shell includes some great features for extending the shell to meet your needs. The problem is, many of these nifty extensible features are some of the most poorly documented subjects of Win32 development. This chapter is intended to give you the information and examples you need to tap into shell features such as tray-notification icons, application desktop toolbars, shell links, and shell extensions.

A Tray-Notification Icon Component

This section illustrates a technique for encapsulating the Windows shell tray-notification icon cleanly into a Delphi component. As you build the component—called `TTrayNotifyIcon`—`you'll` learn about the API requirements for creating a tray-notification icon as well as how to tackle some of the hairy problems you'll come across as you work to embed all the icon's functionality within the component. If you're unfamiliar with what a tray-notification icon is, it's one of those little icons that appear in the bottom-right corner of the Windows system taskbar (assuming that your taskbar is aligned to the bottom of your screen), as shown in Figure 16.1.

FIGURE 16.1
Tray-notification icons.

The API

Believe it or not, only one API call is involved in creating, modifying, and removing tray-notification icons from the notification tray. The function is called `Shell_NotifyIcon()`.This and other functions dealing with the Windows shell are contained in the `ShellAPI` unit. `Shell_NotifyIcon()` is defined as follows:

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData:
    PNotifyIconData): BOOL; stdcall;
```

The `dwMessage` parameter describes the action to be taken for the icon. This can be any one of the values shown in Table 16.1.

TABLE 16.1 Values for the `dwMessage` Parameter

Constant	*Value*	*Meaning*
`NIM_ADD`	`0`	Adds an icon to the notification tray
`NIM_MODIFY`	`1`	Modifies the properties of an existing icon
`NIM_DELETE`	`2`	Removes an icon from the notification tray

The `lpData` parameter is a pointer to a `TNotifyIconData` record. This record is defined as follows:

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

The `cbSize` field holds the size of the record, and it should be initialized to `SizeOf(TNotifyIconData)`.

`Wnd` is the handle of the window to which tray-notification "callback" messages should be sent. (*Callback* is in quotes here because it's not really a callback in the strict sense; however, the Win32 documentation uses this terminology for messages sent to a window on behalf of a tray-notification icon.)

`uID` is a programmer-defined unique ID number. If you have an application with several icons, you'll need to identify each one by a placing a different number in this field.

`uFlags` describes which of the fields of the `TNotifyIconData` record should be considered live by the `Shell_NotifyIcon()` function, and, therefore, which of the icon properties are to be affected by the action specified by the `dwMessage` parameter. This parameter can be any combination of the flags (using `or` to join them) shown in Table 16.2.

TABLE 16.2 Possible Flags to Be Included in `uFlags`

Constant	*Value*	*Meaning*
`NIF_MESSAGE`	`0`	The `uCallbackMessage` field is live.
`NIF_ICON`	`2`	The `hIcon` field is live.
`NIF_TIP`	`4`	The `szTip` field is live.

uCallbackMessage contains the value of the Windows message to be sent to the window identified by the Wnd field. Generally, the value of this field is obtained by calling RegisterWindowMessage() or by using an offset from WM_USER. The lParam of this message will be the same value as the uID field, and the wParam will hold the mouse message generated over the notification icon.

hIcon identifies the handle to the icon that will be placed in the notification tray.

szTip holds a null-terminated string that will appear in the hint window displayed when the mouse pointer is held above the notification icon.

The TTrayNotifyIcon component encapsulates the Shell_NotifyIcon() into a method called SendTrayMessage(), which is shown here:

```
procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ This method wraps up the call to the API's Shell_NotifyIcon }
begin
  { Fill up record with appropriate values }
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := Tray_Callback;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;
```

In this method, szTip is copied from a private string field called FHint.

uID is used to hold a reference to Self. Because this data will be included in subsequent notification tray messages, correlating notification tray messages for multiple icons to individual components will be easy.

Wnd is assigned the value of IconMgr.HWindow. IconMgr is a global variable of type TIconMgr. You'll see the implementation of this object in a moment, but for now you only need know that it's through this component that all notification tray messages will be sent.

uCallbackMessage is assigned from DDGM_TRAYICON. DDGM_TRAYICON obtains its value from the RegisterWindowMessage() API function. This ensures that DDGM_TRAYICON is a systemwide unique message ID. The following code accomplishes this task:

```
const
  { String to identify registered window message }
```

```
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { Get a unique windows message ID for tray callback }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
```

`hIcon` takes on the return value provided by the `ActiveIconHandle()` method. This method returns the handle for the icon currently selected in the component's `Icon` property.

Handling Messages

We mentioned earlier that all notification tray messages are sent to a window maintained by the global `IconMgr` object. This object is constructed and freed in the `initialization` and `finalization` sections of the component's unit, as shown here:

```
initialization
  { Get a unique windows message ID for tray callback }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
```

This object is fairly small. Here's its definition:

```
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;
```

The window to which notification tray messages will be sent is created in the constructor for this object using the `AllocateHWnd()` function:

```
constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;
```

The `TrayWndProc()` method serves as the window procedure for the window created in the constructor. More about this method will be discussed in a moment.

Icons and Hints

The most straightforward way to surface icons and hints for the component's end user is through properties. Additionally, creating an `Icon` property of type `TIcon` means that it can automatically take advantage of Delphi's property editor for icons, which is a nice touch. Because the tray icon is visible even at design time, you need to ensure that the icon and tip can change dynamically. Doing this really isn't a lot of extra work; it's just a matter of making sure that the `SendTrayMessage()` method is called (using the `NIM_MODIFY` message) in the `write` method of the `Hint` and `Icon` properties.

Here are the `write` methods for those properties:

```
procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Write method for Icon property. }
begin
  FIcon.Assign(Value);  // set new icon
  if FIconVisible then
    { Change icon on notification tray }
    SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetHint(Value: String);
{ Set method for Hint property }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { Change hint on icon on notification tray }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;
```

Mouse Clicks

One of the most challenging parts of this component is ensuring that the mouse clicks are handled properly. You might have noticed that many tray-notification icons perform three different actions because of mouse clicks:

- Brings up a window on a single-click
- Brings up a different window (usually a properties sheet) on a double-click
- Invokes a local menu with a right-click

The challenge comes in creating an event that represents the double-click without also firing the single-click event.

In Windows message terms, when the user double-clicks with the left mouse button, the window with focus will receive both the `WM_LBUTTONDOWN` message and the `WM_LBUTTONDBLCLK` message. In order to allow a double-click message to be processed independently of a single-click, some mechanism is required to delay the handling of the single-click message long enough to ensure that a double-click message isn't forthcoming.

The amount of time to wait before you can be sure that a `WM_LBUTTONDBLCLK` message isn't following a `WM_LBUTTONDOWN` message is actually pretty easy to determine. The API function `GetDoubleClickTime()`, which takes no parameters, returns the maximum amount of time (in milliseconds) that the Control Panel will allow between the two clicks of a double-click. The obvious choice for a mechanism to allow you to wait the number of milliseconds specified by `GetDoubleClickTime()` to ensure that a double-click isn't following a click is the `TTimer` component. Therefore, a `TTimer` component is created and initialized in the `TTrayNotifyIcon` component's constructor with the following code:

```
FTimer := TTimer.Create(Self);
with FTimer do
begin
  Enabled := False;
  Interval := GetDoubleClickTime;
  OnTimer := OnButtonTimer;
end;
```

`OnButtonTimer()` is a method that will be called when the timer interval expires. We'll show you this method in just a moment.

Earlier, we mentioned that notification tray messages are filtered through the `TrayWndProc()` method of the `IconMgr`. Now it's time to spring this method on you, so here it is:

```
procedure TIconManager.TrayWndProc(var Message: TMessage);
{ This allows us to handle all tray callback messages }
{ from within the context of the component. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do
  begin
    { if it's the tray callback message }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { enable timer on first mouse down. }
        { OnClick will be fired by OnTimer method, provided }
```

```
        { double click has not occurred. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { Set no click flag on double click.  This will suppress }
        { the single click. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { Call to SetForegroundWindow is required by API }
              SetForegroundWindow(IconMgr.HWindow);
              { Popup local menu at the cursor position. }
              GetCursorPos(Pt);
              TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
              { Message post required by API to force task switch }
              PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
            end;
          end;
      end;
    end
    else
      { If it isn't a tray callback message, then call DefWindowProc }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;
```

What makes this all work is that the single-click message merely enables the timer, whereas the double-click message sets a flag to indicate that the double-click has occurred before firing its `OnDblClick` event. The right-click, incidentally, invokes the pop-up menu given by the component's `PopupMenu` property. Now take a look at the `OnButtonTimer` method:

```
procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
begin
  { Disable timer because we only want it to fire once. }
  FTimer.Enabled := False;
  { if double click has not occurred, then fire single click. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;   // reset flag
end;
```

This method first disables the timer to ensure that the event fires only once per mouse click. The method then checks the status of the `FNoShowClick` flag. Remember that this flag will be set by the double-click message in the `OwnerWndProc()` method. Therefore, the `OnClick` event will be fired only when `OnDblClk` isn't.

Hiding the Application

Another aspect of tray-notification applications is that they don't appear as buttons in the system taskbar. To provide this functionality, the `TTrayNotifyIcon` component surfaces a `HideTask` property that allows the user to decide whether the application should be visible in the taskbar. The `write` method for this property is shown in the following code. The line of code that does the work is the call to the `ShowWindow()` API procedure, which passes the `Handle` property of `Application` and a constant to indicate whether the application is to be shown normally or hidden. Here's the code:

```
procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Write method for HideTask property }
const
  { Flags to show application normally or hide it }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then begin
    FHideTask := Value;
    { Don't do anything in design mode }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;
```

Listing 16.1 shows the `TrayIcon.pas` unit, which contains the complete source code for the `TTrayNotifyIcon` component.

LISTING 16.1 `TrayIcon.pas`—Source Code for the `TTrayNotifyIcon` Component

```
unit TrayIcon;

interface

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

type
  ENotifyIconError = class(Exception);

  TTrayNotifyIcon = class(TComponent)
```

LISTING 16.1 Continued

```
  private
    FDefaultIcon: THandle;
    FIcon: TIcon;
    FHideTask: Boolean;
    FHint: string;
    FIconVisible: Boolean;
    FPopupMenu: TPopupMenu;
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FNoShowClick: Boolean;
    FTimer: TTimer;
    Tnd: TNotifyIconData;
    procedure SetIcon(Value: TIcon);
    procedure SetHideTask(Value: Boolean);
    procedure SetHint(Value: string);
    procedure SetIconVisible(Value: Boolean);
    procedure SetPopupMenu(Value: TPopupMenu);
    procedure SendTrayMessage(Msg: DWORD; Flags: UINT);
    function ActiveIconHandle: THandle;
    procedure OnButtonTimer(Sender: TObject);
  protected
    procedure Loaded; override;
    procedure LoadDefaultIcon; virtual;
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Icon: TIcon read FIcon write SetIcon;
    property HideTask: Boolean read FHideTask write SetHideTask default False;
    property Hint: String read FHint write SetHint;
    property IconVisible: Boolean read FIconVisible write SetIconVisible
      default False;
    property PopupMenu: TPopupMenu read FPopupMenu write SetPopupMenu;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
  end;

implementation

{ TIconManager }
{ This class creates a hidden window which handles and routes }
{ tray icon messages }
```

LISTING 16.1 Continued

```
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;

var
  IconMgr: TIconManager;
  DDGM_TRAYICON: Integer;

constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;

destructor TIconManager.Destroy;
begin
  if FHWindow <> 0 then DeallocateHWnd(FHWindow);
  inherited Destroy;
end;

procedure TIconManager.TrayWndProc(var Message: TMessage);
{ This allows us to handle all tray callback messages }
{ from within the context of the component. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do
  begin
    { if it's the tray callback message }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { enable timer on first mouse down. }
        { OnClick will be fired by OnTimer method, provided }
        { double click has not occurred. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
```

LISTING 16.1 Continued

```
        { Set no click flag on double click.  This will suppress }
        { the single click. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { Call to SetForegroundWindow is required by API }
              SetForegroundWindow(IconMgr.HWindow);
              { Popup local menu at the cursor position. }
              GetCursorPos(Pt);
              TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
              { Message post required by API to force task switch }
              PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
            end;
          end;
      end;
    end
    else
      { If it isn't a tray callback message, then call DefWindowProc }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;

{ TTrayNotifyIcon }

constructor TTrayNotifyIcon.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FIcon := TIcon.Create;
  FTimer := TTimer.Create(Self);
  with FTimer do
  begin
    Enabled := False;
    Interval := GetDoubleClickTime;
    OnTimer := OnButtonTimer;
  end;
  { Keep default windows icon handy... }
  LoadDefaultIcon;
end;
```

LISTING 16.1 Continued

```
destructor TTrayNotifyIcon.Destroy;
begin
  if FIconVisible then SetIconVisible(False);    // destroy icon
  FIcon.Free;                                    // free stuff
  FTimer.Free;
  inherited Destroy;
end;

function TTrayNotifyIcon.ActiveIconHandle: THandle;
{ Returns handle of active icon }
begin
  { If no icon is loaded, then return default icon }
  if (FIcon.Handle <> 0) then
    Result := FIcon.Handle
  else
    Result := FDefaultIcon;
end;

procedure TTrayNotifyIcon.LoadDefaultIcon;
{ Loads default window icon to keep it handy. }
{ This will allow the component to use the windows logo }
{ icon as the default when no icon is selected in the }
{ Icon property. }
begin
  FDefaultIcon := LoadIcon(0, IDI_WINLOGO);
end;

procedure TTrayNotifyIcon.Loaded;
{ Called after component is loaded from stream }
begin
  inherited Loaded;
  { if icon is supposed to be visible, create it. }
  if FIconVisible then
    SendTrayMessage(NIM_ADD, NIF_MESSAGE or NIF_ICON or NIF_TIP);
end;

procedure TTrayNotifyIcon.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = PopupMenu) then
    PopupMenu := nil;
end;
```

LISTING 16.1 Continued

```
procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
{ Timer used to keep track of time between two clicks of a }
{ double click. This delays the first click long enough to }
{ ensure that a double click hasn't occurred.  The whole   }
{ point of these gymnastics is to allow the component to   }
{ receive OnClicks and OnDblClicks independently. }
begin
  { Disable timer because we only want it to fire once. }
  FTimer.Enabled := False;
  { if double click has not occurred, then fire single click. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;   // reset flag
end;

procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ This method wraps up the call to the API's Shell_NotifyIcon }
begin
  { Fill up record with appropriate values }
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := DDGM_TRAYICON;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;

procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Write method for HideTask property }
const
  { Flags to show application normally or hide it }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then
  begin
    FHideTask := Value;
    { Don't do anything in design mode }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;
```

LISTING 16.1 Continued

```
procedure TTrayNotifyIcon.SetHint(Value: string);
{ Set method for Hint property }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { Change hint on icon on notification tray }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Write method for Icon property. }
begin
  FIcon.Assign(Value);  // set new icon
  { Change icon on notification tray }
  if FIconVisible then SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetIconVisible(Value: Boolean);
{ Write method for IconVisible property }
const
  { Flags to add or delete a tray-notification icon }
  MsgArray: array[Boolean] of DWORD = (NIM_DELETE, NIM_ADD);
begin
  if FIconVisible <> Value then
  begin
    FIconVisible := Value;
    { Set icon as appropriate }
    SendTrayMessage(MsgArray[Value], NIF_MESSAGE or NIF_ICON or NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetPopupMenu(Value: TPopupMenu);
{ Write method for PopupMenu property }
begin
  FPopupMenu := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

const
  { String to identify registered window message }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';
```

LISTING 16.1 Continued

```
initialization
  { Get a unique windows message ID for tray callback }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
end.
```

Figure 16.2 shows a picture of the icon generated by `TTrayNotifyIcon` in the notification tray.

FIGURE 16.2
The `TTrayNotifyIcon` *component in action.*

By the way, because the tray icon is initialized inside the component's constructor and because constructors are executed at design time, this component displays the tray-notification icon even at design time!

Sample Tray Application

In order to provide you with a better overall feel for how the `TTrayNotifyIcon` component works within the context of an application, Figure 16.3 shows the main window of this application, and Listing 16.2 shows the fairly minimal code for the main unit for this application.

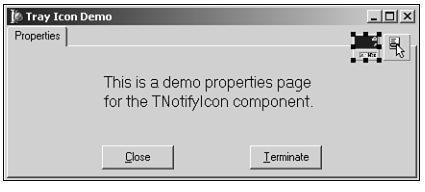


FIGURE 16.3
Notification icon application.

LISTING 16.2 `Main.pas`—the Main Unit for the Notification Icon Demo Application

```
unit main;

interface

uses
```

LISTING 16.2 Continued

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

type
  TMainForm = class(TForm)
    pmiPopup: TPopupMenu;
    pgclPageCtl: TPageControl;
    TabSheet1: TTabSheet;
    btnClose: TButton;
    btnTerm: TButton;
    Terminate1: TMenuItem;
    Label1: TLabel;
    N1: TMenuItem;
    Propeties1: TMenuItem;
    TrayNotifyIcon1: TTrayNotifyIcon;
    procedure NotifyIcon1Click(Sender: TObject);
    procedure NotifyIcon1DblClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnTermClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.NotifyIcon1Click(Sender: TObject);
begin
  ShowMessage('Single click');
end;

procedure TMainForm.NotifyIcon1DblClick(Sender: TObject);
begin
  Show;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caNone;
  Hide;
end;
```

LISTING 16.2 Continued

```
procedure TMainForm.btnTermClick(Sender: TObject);
begin
  Application.Terminate;
end;

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

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TrayNotifyIcon1.IconVisible := True;
end;

end.
```

Application Desktop Toolbars

Application desktop toolbars, also known as *AppBars*, are windows that can dock to one of the edges of your screen. You're already familiar with AppBars, even though you might not know it; the shell's taskbar, which you probably work with every day, is an example of an AppBar. As shown in Figure 16.4, the taskbar is really little more than an AppBar window containing a Start button, notification tray, and other controls.

FIGURE 16.4
The shell's taskbar.

Apart from docking to screen edges, AppBars can, optionally, employ taskbar-like features, such as auto-hide and drag-and-drop functionality. What you might find surprising, however, is how small the API is (just one function). As its small size might imply, the API doesn't provide a whole lot. The role of the API is more advisory than functional. That is, rather than controlling the AppBar with "do this, do that" command types, you interrogate the AppBar with "can I do this, can I do that?" command types.

The API

Just like tray-notification icons, AppBars have only one API function that you'll work with—`SHAppBarMessage()`, in this case. Here's how `SHAppBarMessage()` is defined in the `ShellAPI` unit:

```
function SHAppBarMessage(dwMessage: DWORD; var pData: TAppBarData): UINT;
  stdcall;
```

The first parameter to this function, `dwMessage`, can contain any one of the values described in Table 16.3.

TABLE 16.3 AppBar Messages

Constant	*Value*	*Meaning*
`ABM_NEW`	`$0`	Registers a new AppBar and specifies a new callback message
`ABM_REMOVE`	`$1`	Unregisters an existing AppBar
`ABM_QUERYPOS`	`$2`	Requests a new position and size for an AppBar
`ABM_SETPOS`	`$3`	Sets a new position and size of an AppBar
`ABM_GETSTATE`	`$4`	Gets the auto-hide and always-on-top states of the shell taskbar
`ABM_GETTASKBARPOS`	`$5`	Gets the position of the shell taskbar
`ABM_ACTIVATE`	`$6`	Notifies the shell that a new AppBar has been created
`ABM_GETAUTOHIDEBAR`	`$7`	Gets the handle of an auto-hide AppBar docked to a particular edge of the screen
`ABM_SETAUTOHIDEBAR`	`$8`	Registers an auto-hide AppBar for a particular screen edge
`ABM_WINDOWPOSCHANGED`	`$9`	Notifies the shell that the position of an AppBar has changed

The `pData` parameter of `SHAppBarMessage()` is a record of type `TAppBarData`, which is defined in `ShellAPI` as follows:

```
type
  PAppBarData = ^TAppBarData;
  TAppBarData = record
    cbSize: DWORD;
    hWnd: HWND;
    uCallbackMessage: UINT;
    uEdge: UINT;
    rc: TRect;
    lParam: LPARAM; { message specific }
  end;
```

In this record, the `cbSize` field holds the size of the record, the `hWnd` field holds the window handle of the specified AppBar, `uCallbackMessage` holds the message value that will be sent to the AppBar window along with notification messages, `rc` holds the bounding rectangle of the AppBar in question, and `lParam` holds some additional message-specific information.

TIP

You'll find more information on the `SHAppBarMessage()` API function and the `TAppBarData` type in the Win32 online help.

`TAppBar`: The AppBar Form

Given this fairly small API, it's not terribly difficult to encapsulate an AppBar in a VCL form. This section explains the techniques used to wrap the AppBar API into a control descending from `TCustomForm`. Because `TCustomForm` is a form, you'll interact with the control as a top-level form in the Form Designer rather than as a component on a form.

Most of the work in an AppBar is done by sending a `TAppBarData` record to the shell using the `SHAppBarMessage()` API function. The `TAppBar` component maintains an internal `TAppBarData` record called `FABD`. `FABD` is set up for the call to `SendAppBarMsg()` in the constructor and the `CreateWnd()` methods in order to create the AppBar. In particular, the `cbSize` field is initialized, the `uCallbackMessage` field is set to a value obtained from the `RegisterWindowMessage()` API function, and the `hWnd` field is set to the current window handle of the form. `SendAppBarMessage()` is a simple wrapper for `SHAppBarMessage()` and is defined as follows:

```
function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  Result := SHAppBarMessage(Msg, FABD);
end;
```

If the AppBar is created successfully, the `SetAppBarEdge()` method is called to set the AppBar to its initial position. This method, in turn, calls the `SetAppBarPos()` method, passing the appropriate API-defined flag that indicates the requested screen edge. As you would expect, the `ABE_TOP`, `ABE_BOTTOM`, `ABE_LEFT`, and `ABE_RIGHT` flags represent each of the screen edges. This is shown in the following code snippet:

```
procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;       // set edge
  with FABD.rc do
```

```
  begin
    // set coordinates to full-screen
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    // Send ABM_QUERYPOS to obtain proper rect on edge
    SendAppBarMsg(ABM_QUERYPOS);
    // re-adjust rect based on that modified by ABM_QUERYPOS
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Set the app bar position.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Set the BoundsRect property so that it conforms to the
  // bounding rectangle passed to the system.
  BoundsRect := FABD.rc;
end;
```

This method first sets the `uEdge` field of `FABD` to the value passed via the `Edge` parameter. It then sets the `rc` field to the full-screen coordinates and sends the `ABM_QUERYPOS` message. This message resets the `rc` field so that it contains the correct bounding rectangle for the edge indicated by `uEdge`. Once the proper bounding rectangle has been obtained, `rc` is again adjusted so that it's a reasonable height or width. At this point, `rc` holds the final bounding rectangle for the AppBar. The `ABM_SETPOS` message is then sent to inform the shell of the new rectangle, and the rectangle is set using the control's `BoundsRect` property.

We mentioned earlier that AppBar notification messages will be sent to the window indicated by `FABD.hWnd` using the message identifier held in `FABD.uCallbackMessage`. These notification messages are handled in the `WndProc()` method shown here:

```
procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
      // Sent when always on top or auto-hide state has changed.
      ABN_STATECHANGE:
```

```
        begin
          // Check to see whether the access bar is still ABS_ALWAYSONTOP.
          State := SendAppBarMsg(ABM_GETSTATE);
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end;
      // A full screen application has started, or the last
      // full-screen application has closed.
      ABN_FULLSCREENAPP:
        begin
          // Set the access bar's z-order appropriately.
          State := SendAppBarMsg(ABM_GETSTATE);
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost(False)
            else
              SetTopMost(True);
          end
          else
            if State and ABS_ALWAYSONTOP <> 0 then
              SetTopMost(True);
        end;
      // Sent when something happened which may effect the AppBar position.
      ABN_POSCHANGED:
        begin
          // The taskbar or another access bar
          // has changed its size or position.
          SetAppBarPos(FABD.uEdge);
        end;
    end;
  end
  else
    inherited WndProc(M);
end;
```

This method handles some notification messages that permit the AppBar to respond to changes that might occur in the shell while the application is running. The remainder of the AppBar component code is shown in Listing 16.3.

LISTING 16.3 `AppBars.pas`—Unit Containing Base Class for AppBar Support

```
unit AppBars;

interface
```

LISTING 16.3 Continued

```
uses Windows, Messages, SysUtils, Forms, ShellAPI, Classes, Controls;

type
  TAppBarEdge = (abeTop, abeBottom, abeLeft, abeRight);

  EAppBarError = class(Exception);

  TAppBar = class(TCustomForm)
  private
    FABD: TAppBarData;
    FDockedHeight: Integer;
    FDockedWidth: Integer;
    FEdge: TAppBarEdge;
    FOnEdgeChanged: TNotifyEvent;
    FTopMost: Boolean;
    procedure WMActivate(var M: TMessage); message WM_ACTIVATE;
    procedure WMWindowPosChanged(var M: TMessage); message WM_WINDOWPOSCHANGED;
    function SendAppBarMsg(Msg: DWORD): UINT;
    procedure SetAppBarEdge(Value: TAppBarEdge);
    procedure SetAppBarPos(Edge: UINT);
    procedure SetTopMost(Value: Boolean);
    procedure SetDockedHeight(const Value: Integer);
    procedure SetDockedWidth(const Value: Integer);
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure CreateWnd; override;
    procedure DestroyWnd; override;
    procedure WndProc(var M: TMessage); override;
  public
    constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0); override;
    property DockManager;
  published
    property Action;
    property ActiveControl;
    property AutoScroll;
    property AutoSize;
    property BiDiMode;
    property BorderWidth;
    property Color;
    property Ctl3D;
    property DockedHeight: Integer read FDockedHeight write SetDockedHeight
      default 35;
    property DockedWidth: Integer read FDockedWidth write SetDockedWidth
      default 40;
```

LISTING 16.3 Continued

```
    property UseDockManager;
    property DockSite;
    property DragKind;
    property DragMode;
    property Edge: TAppBarEdge read FEdge write SetAppBarEdge default abeTop;
    property Enabled;
    property ParentFont default False;
    property Font;
    property HelpFile;
    property HorzScrollBar;
    property Icon;
    property KeyPreview;
    property ObjectMenuItem;
    property ParentBiDiMode;
    property PixelsPerInch;
    property PopupMenu;
    property PrintScale;
    property Scaled;
    property ShowHint;
    property TopMost: Boolean read FTopMost write SetTopMost default False;
    property VertScrollBar;
    property Visible;
    property OnActivate;
    property OnCanResize;
    property OnClick;
    property OnClose;
    property OnCloseQuery;
    property OnConstrainedResize;
    property OnCreate;
    property OnDblClick;
    property OnDestroy;
    property OnDeactivate;
    property OnDockDrop;
    property OnDockOver;
    property OnDragDrop;
    property OnDragOver;
    property OnEdgeChanged: TNotifyEvent read FOnEdgeChanged
      write FOnEdgeChanged;
    property OnEndDock;
    property OnGetSiteInfo;
    property OnHide;
    property OnHelp;
    property OnKeyDown;
    property OnKeyPress;
```

LISTING 16.3 Continued

```
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnMouseWheel;
    property OnMouseWheelDown;
    property OnMouseWheelUp;
    property OnPaint;
    property OnResize;
    property OnShortCut;
    property OnShow;
    property OnStartDock;
    property OnUnDock;
  end;

implementation

var
  AppBarMsg: UINT;

constructor TAppBar.CreateNew(AOwner: TComponent; Dummy: Integer);
begin
  FDockedHeight := 35;
  FDockedWidth := 40;
  inherited CreateNew(AOwner, Dummy);
  ClientHeight := 35;
  Width := 100;
  BorderStyle := bsNone;
  BorderIcons := [];
  // set up the TAppBarData record
  FABD.cbSize := SizeOf(FABD);
  FABD.uCallbackMessage := AppBarMsg;
end;

procedure TAppBar.WMWindowPosChanged(var M: TMessage);
begin
  inherited;
  // Must inform shell that the AppBar position has changed
  SendAppBarMsg(ABM_WINDOWPOSCHANGED);
end;

procedure TAppBar.WMActivate(var M: TMessage);
begin
  inherited;
```

LISTING 16.3 Continued

```
  // Must inform shell that the AppBar window was activated
  SendAppBarMsg(ABM_ACTIVATE);
end;

procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
      // Sent when always on top or auto-hide state has changed.
      ABN_STATECHANGE:
        begin
          // Check to see whether the access bar is still ABS_ALWAYSONTOP.
          State := SendAppBarMsg(ABM_GETSTATE);
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end;
      // A full screen application has started, or the last
      // full-screen application has closed.
      ABN_FULLSCREENAPP:
        begin
          // Set the access bar's z-order appropriately.
          State := SendAppBarMsg(ABM_GETSTATE);
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost(False)
            else
              SetTopMost(True);
          end
          else
            if State and ABS_ALWAYSONTOP <> 0 then
              SetTopMost(True);
        end;
      // Sent when something happened which may effect the AppBar position.
      ABN_POSCHANGED:
        // The taskbar or another access bar
        // has changed its size or position.
        SetAppBarPos(FABD.uEdge);
    end;
  end
```

LISTING 16.3 Continued

```
  else
    inherited WndProc(M);
end;

function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  // Don't do AppBar stuff at design time... too funky
  if csDesigning in ComponentState then Result := 0
  else Result := SHAppBarMessage(Msg, FABD);
end;

procedure TAppBar.SetAppBarPos(Edge:  UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;       // set edge
  with FABD.rc do
  begin
    // set coordinates to full-screen
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    // Send ABM_QUERYPOS to obtain proper rect on edge
    SendAppBarMsg(ABM_QUERYPOS);
    // re-adjust rect based on that modified by ABM_QUERYPOS
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Set the app bar position.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Set the BoundsRect property so that it conforms to the
  // bounding rectangle passed to the system.
  BoundsRect := FABD.rc;
end;

procedure TAppBar.SetTopMost(Value: Boolean);
const
  WndPosArray: array[Boolean] of HWND = (HWND_BOTTOM, HWND_TOPMOST);
begin
  if FTopMost <> Value then
```

LISTING 16.3 Continued

```
  begin
    FTopMost := Value;
    if not (csDesigning in ComponentState) then
      SetWindowPos(Handle, WndPosArray[Value], 0, 0, 0, 0, SWP_NOMOVE or
        SWP_NOSIZE or SWP_NOACTIVATE);
  end;
end;

procedure TAppBar.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  if not (csDesigning in ComponentState) then
  begin
    Params.ExStyle := Params.ExStyle or WS_EX_TOPMOST or WS_EX_WINDOWEDGE;
    Params.Style := Params.Style or WS_DLGFRAME;
  end;
end;

procedure TAppBar.CreateWnd;
begin
  inherited CreateWnd;
  FABD.hWnd := Handle;
  if not (csDesigning in ComponentState) then
  begin
    if SendAppBarMsg(ABM_NEW) = 0 then
      raise EAppBarError.Create('Failed to create AppBar');
    // Initialize the position
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.DestroyWnd;
begin
  // Must inform shell that the AppBar is going away
  SendAppBarMsg(ABM_REMOVE);
  inherited DestroyWnd;
end;

procedure TAppBar.SetAppBarEdge(Value: TAppBarEdge);
const
  EdgeArray: array[TAppBarEdge] of UINT =
    (ABE_TOP, ABE_BOTTOM, ABE_LEFT, ABE_RIGHT);
begin
  SetAppBarPos(EdgeArray[Value]);
```

LISTING 16.3 Continued

```
  FEdge := Value;
  if Assigned(FOnEdgeChanged) then FOnEdgeChanged(Self);
end;

procedure TAppBar.SetDockedHeight(const Value: Integer);
begin
  if FDockedHeight <> Value then
  begin
    FDockedHeight := Value;
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.SetDockedWidth(const Value: Integer);
begin
  if FDockedWidth <> Value then
  begin
    FDockedWidth := Value;
    SetAppBarEdge(FEdge);
  end;
end;

initialization
  AppBarMsg := RegisterWindowMessage('DDG AppBar Message');
end.
```

Using TAppBar

If you installed the software found on the CD-ROM accompanying this book, using a `TAppBar` should be a snap: just select the AppBar option from the DDG page of the File, New dialog box. This invokes a wizard that will generate a unit containing a `TAppBar` component.

NOTE

Chapter 17, "Using the Open Tools API," demonstrates how to create a wizard that automatically generates a `TAppBar`. For the purposes of this chapter, you can ignore the wizard implementation for the time being. Just understand that some work is being done behind the scenes to generate the AppBar's form and unit for you.

In this small sample application, `TAppBar`is used to create an application toolbar that contains buttons for various editing commands: Open, Save, Cut, Copy, and Paste. The buttons will manipulate a `TMemo` component found on the main form. The source code for this unit is shown in Listing 16.4, and Figure 16.5 shows the application in action with the AppBar control docked at the bottom of the screen.

LISTING 16.4 `ApBarFrm.pas`—Main Unit for AppBar Demo Application

```
unit ApBarFrm;

interface

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

type
  TAppBarForm = class(TAppBar)
    sbOpen: TSpeedButton;
    sbSave: TSpeedButton;
    sbCut: TSpeedButton;
    sbCopy: TSpeedButton;
    sbPaste: TSpeedButton;
    OpenDialog: TOpenDialog;
    pmPopup: TPopupMenu;
    Top1: TMenuItem;
    Bottom1: TMenuItem;
    Left1: TMenuItem;
    Right1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure Right1Click(Sender: TObject);
    procedure sbOpenClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbCutClick(Sender: TObject);
    procedure sbCopyClick(Sender: TObject);
    procedure sbPasteClick(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormEdgeChanged(Sender: TObject);
  private
    FLastChecked: TMenuItem;
    procedure MoveButtons;
  end;
```

LISTING 16.4 Continued

```
var
  AppBarForm: TAppBarForm;

implementation

uses Main;

{$R *.DFM}

{ TAppBarForm }

procedure TAppBarForm.MoveButtons;
// This method looks complicated, but it really just arranges the buttons
// properly depending on what side the AppBar is docked.
var
  DeltaCenter, NewPos: Integer;
begin
  if Edge in [abeTop, abeBottom] then
  begin
    DeltaCenter := (ClientHeight - sbOpen.Height) div 2;
    sbOpen.SetBounds(10, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Width + 20;
    sbSave.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCut.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCopy.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbPaste.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
  end
  else
  begin
    DeltaCenter := (ClientWidth - sbOpen.Width) div 2;
    sbOpen.SetBounds(DeltaCenter, 10, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Height + 20;
    sbSave.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCut.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCopy.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbPaste.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
  end;
end;
```

LISTING 16.4 Continued

```
procedure TAppBarForm.Right1Click(Sender: TObject);
begin
  FLastChecked.Checked := False;
  (Sender as TMenuItem).Checked := True;
  case TMenuItem(Sender).Caption[2] of
    'T': Edge := abeTop;
    'B': Edge := abeBottom;
    'L': Edge := abeLeft;
    'R': Edge := abeRight;
  end;
  FLastChecked := TMenuItem(Sender);
end;

procedure TAppBarForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    MainForm.FileName := OpenDialog.FileName;
end;

procedure TAppBarForm.sbSaveClick(Sender: TObject);
begin
  MainForm.memEditor.Lines.SaveToFile(MainForm.FileName);
end;

procedure TAppBarForm.sbCutClick(Sender: TObject);
begin
  MainForm.memEditor.CutToClipboard;
end;

procedure TAppBarForm.sbCopyClick(Sender: TObject);
begin
  MainForm.memEditor.CopyToClipboard;
end;

procedure TAppBarForm.sbPasteClick(Sender: TObject);
begin
  MainForm.memEditor.PasteFromClipboard;
end;

procedure TAppBarForm.Exit1Click(Sender: TObject);
begin
  Application.Terminate;
end;
```

LISTING 16.4 Continued

```
procedure TAppBarForm.FormCreate(Sender: TObject);
begin
  FLastChecked := Top1;
end;

procedure TAppBarForm.FormEdgeChanged(Sender: TObject);
begin
  MoveButtons;
end;

end.
```

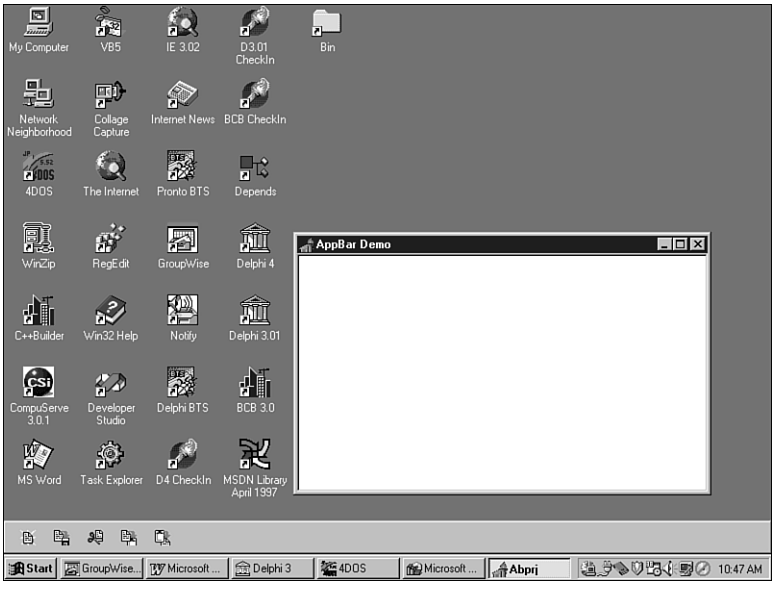


FIGURE 16.5
`TAppBar` *in action.*

Shell Links

The Windows shell exposes a series of interfaces that can be employed to manipulate different aspects of the shell. These interfaces are defined in the `ShlObj` unit. Discussing in depth all the objects in that unit could take a book in its own right, so for now we'll focus on one of the most useful (and most used) interfaces: `IShellLink`.

`IShellLink` is an interface that permits the creating and manipulating of shell links in your applications. In case you're unsure, most of the icons on your desktop are probably shell links. Additionally, each item in the shell's local Send To menu or the Documents menu (off of the Start menu) are all shell links. The `IShellLink` interface is defined as follows:

```
const

type
  IShellLink = interface(IUnknown)
    ['{000214EE-0000-0000-C000-000000000046}']
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
      var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar; cchMaxName: Integer): HResult;
      stdcall;
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar; cchMaxPath: Integer):
      HResult;
      stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult; stdcall;
    function GetArguments(pszArgs: PAnsiChar; cchMaxPath: Integer): HResult;
      stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath: Integer;
      out piIcon: Integer): HResult; stdcall;
    function SetIconLocation(pszIconPath: PAnsiChar; iIcon: Integer): HResult;
      stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar; dwReserved: DWORD):
      HResult;
      stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
  end;
```

NOTE

`IShellLink` and all its methods are described in detail in the Win32 online help, so we won't cover them here.

Obtaining an `IShellLink` Instance

Unlike working with shell extensions, which you'll learn about later in this chapter, you don't implement the `IShellLink` interface. Instead, this interface is implemented by the Windows shell, and you use the `CoCreateInstance()` COM function to create an instance. Here's an example:

```
var
  SL: IShellLink;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  // use SL here
end;
```

NOTE

Don't forget that before you can use any OLE functions, you must initialize the COM library using the `CoInitialize()` function. When you're through using COM, you must clean up by calling `CoUninitialize()`. These functions will be called for you by Delphi in an application that uses `ComObj` and contains a call to `Application.Initialize()`. Otherwise, you'll have to call these functions yourself.

Using `IShellLink`

Shell links seem kind of magical: you right-click on the desktop, create a new shortcut, and *something* happens that causes an icon to appear on the desktop. That *something* is actually a pretty mundane occurrence once you know what's going on. A *shell link* is actually just a file with an `.LNK` extension that lives in some particular directory. When Windows starts up, it looks in certain directories for LNK files, which represent links residing in different *shell folders*. These shell folders, or *special folders*, include items such as Network Neighborhood, Send To, Startup, the Desktop, and so on. The shell stores the link/folder correspondence in the System Registry—they're found mostly under the following key if you're interested in looking:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer
➥\Shell Folders
```

Creating a shell link in a special folder, then, is just a matter of placing a link file in a particular directory. Rather than spelunking through the Registry, you can use the `SHGetSpecialFolderPath()` to obtain the directory path for the various special folders. This method is defined as follows:

```
function SHGetSpecialFolderPath(hwndOwner: HWND; lpszPath: PChar;
  nFolder: Integer; fCreate: BOOL): BOOL; stdcall;
```

`hwndOwner` contains the handle of a window that will serve as the owner to any dialogs the function might invoke.

`lpszPath` is a pointer to a buffer to receive the path. This buffer must be at least `MAX_PATH` characters in length.

`nFolder` identifies the special folder for which you want to obtain the path. Table 16.4 shows the possible values for this parameter and a description for each.

`fCreate` indicates whether a folder should be created if it doesn't exist.

TABLE 16.4 Possible Values for `nFolder`

Flag	*Description*
`CSIDL_ALTSTARTUP`	The directory that corresponds to the user's non-localized Startup program group.
`CSIDL_APPDATA`	The directory that serves as a common repository for application-specific data.
`CSIDL_BITBUCKET`	The directory containing file objects in the user's Recycle Bin. The location of this directory isn't in the Registry; it's marked with the hidden and system attributes to prevent the user from moving or deleting it.
`CSIDL_COMMON_ALTSTARTUP`	The directory that corresponds to the nonlocalized Startup program group for all users.
`CSIDL_COMMON_DESKTOPDIRECTORY`	The directory that contains files and folders that appear on the desktop for all users.
`CSIDL_COMMON_FAVORITES`	The directory that serves as a common repository for all users' favorite items.
`CSIDL_COMMON_PROGRAMS`	The directory that contains the directories for the common program groups that appear on the Start menu for all users.
`CSIDL_COMMON_STARTMENU`	The directory that contains the programs and folders that appear on the Start menu for all users.
`CSIDL_COMMON_STARTUP`	The directory that contains the programs that appear in the Startup folder for all users.
`CSIDL_CONTROLS`	A virtual folder containing icons for the Control Panel applications.
`CSIDL_COOKIES`	The directory that serves as a common repository for Internet cookies.

TABLE 16.4 Continued

Flag	*Description*
CSIDL_DESKTOP	The Windows Desktop virtual folder at the root of the namespace.
CSIDL_DESKTOPDIRECTORY	The directory used to physically store file objects on the desktop (not to be confused with the Desktop folder, itself).
CSIDL_DRIVES	The My Computer virtual folder containing everything on the local computer: storage devices, printers, and the Control Panel. The folder might also contain mapped network drives.
CSIDL_FAVORITES	The directory that serves as a common repository for the user's favorite items.
CSIDL_FONTS	A virtual folder containing fonts.
CSIDL_HISTORY	The directory that serves as a common repository for Internet history items.
CSIDL_INTERNET	A virtual folder representing the Internet.
CSIDL_INTERNET_CACHE	The directory that serves as a common repository for temporary Internet files.
CSIDL_NETHOOD	The directory that contains objects that appear in the Network Neighborhood.
CSIDL_NETWORK	The Network Neighborhood virtual folder representing the top level of the network hierarchy.
CSIDL_PERSONAL	The directory that serves as a common repository for documents.
CSIDL_PRINTERS	A virtual folder containing installed printers.
CSIDL_PRINTHOOD	The directory that serves as a common repository for printer links.
CSIDL_PROGRAMS	The directory that contains the user's program groups (which are also directories).
CSIDL_RECENT	The directory that contains the user's most recently used documents.
CSIDL_SENDTO	The directory that contains Send To menu items.
CSIDL_STARTMENU	The directory that contains Start menu items.

TABLE 16.4 Continued

Flag	*Description*
CSIDL_STARTUP	The directory that corresponds to the user's Startup program group. The system starts these programs whenever any user logs onto Windows NT or starts Windows 95 or 98.
CSIDL_TEMPLATES	The directory that serves as a common repository for document templates.

Creating a Shell Link

The IShellLink interface is an encapsulation of a shell link object, but it has no concept of how to read or write itself to a file on disk. However, implementers of the IShellLink interface are also required to support the IPersistFile interface in order to provide file access. IPersistFile is an interface that provides methods for reading and writing to and from disk, and it's defined as follows:

```
type
  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult;
      stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult;
      stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
      stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
      stdcall;
  end;
```

NOTE

You'll find a complete description of IPersistFile and its methods in the Win32 online help.

Because the class that implements IShellLink is also required to implement IPeristFile, you can QueryInterface the IShellLink instance for an IPersistFile instance using the as operator, as shown here:

```
var
  SL: IShellLink;
  PF: IPersistFile;
```

```
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  PF := SL as IPersistFile;
  // use PF and SL
end;
```

As mentioned earlier, using COM interface objects works the same as using normal Object Pascal objects. The following code, for example, creates a desktop shell link to the Notepad application:

```
procedure MakeNotepad;
const
  // NOTE: Assumed location for Notepad:
  AppName = 'c:\windows\notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { IShellLink implementers are required to implement IPersistFile }
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName)));   // set link path to proper file
  { create a path location and filename for link file }
  LnkName := GetFolderLocation('Desktop') + '\' +
    ChangeFileExt(ExtractFileName(AppName), '.lnk');
  PF.Save(PWideChar(LnkName), True);      // save link file
end;
```

In this procedure, the `SetPath()` method of `IShellLink` is used to point the link to an executable file or document (Notepad in this case). Then, a path and filename for the link is created using the path returned by `GetFolderLocation('Desktop')` (described earlier in this section) and by using the `ChangeFileExt()` function to change the extension of Notepad from `.EXE` to `.LNK`. This new filename is stored in `LnkName`. After that, the `Save()` method saves the link to a disk file. As you've learned, when the procedure terminates and the `SL` and `PF` interface instances fall out of scope, their respective references will be released.

Getting and Setting Link Information

As you can see from the definition of the `IShellLink` interface, it contains a number of `Get`*`XXX`*`()` and `Set`*`XXX`*`()` methods that allow you to get and set different aspects of the shell link. Consider the following record declaration, which contains fields for each of the possible values that can be set or retrieved:

```
type
  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: Integer;
    ShowCmd: Integer;
    HotKey: Word;
  end;
```

Given this record, you can create functions that retrieve the settings of a given shell link to the record or that set a link's values to those indicated by the record's contents. Such functions are shown in Listing 16.5; `WinShell.pas` is a unit that contains the complete source for these functions.

LISTING 16.5 `WinShell.pas`—Unit Containing Functions That Operate on Shell Links

```
unit WinShell;

interface

uses SysUtils, Windows, Registry, ActiveX, ShlObj;

type
  EShellOleError = class(Exception);

  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: integer;
    ShowCmd: integer;
    HotKey: word;
  end;

  TSpecialFolderInfo = record
    Name: string;
    ID: Integer;
  end;

const
  SpecialFolders: array[0..29] of TSpecialFolderInfo = (
```

LISTING 16.5 Continued

```
    (Name: 'Alt Startup'; ID: CSIDL_ALTSTARTUP),
    (Name: 'Application Data'; ID: CSIDL_APPDATA),
    (Name: 'Recycle Bin'; ID: CSIDL_BITBUCKET),
    (Name: 'Common Alt Startup'; ID: CSIDL_COMMON_ALTSTARTUP),
    (Name: 'Common Desktop'; ID: CSIDL_COMMON_DESKTOPDIRECTORY),
    (Name: 'Common Favorites'; ID: CSIDL_COMMON_FAVORITES),
    (Name: 'Common Programs'; ID: CSIDL_COMMON_PROGRAMS),
    (Name: 'Common Start Menu'; ID: CSIDL_COMMON_STARTMENU),
    (Name: 'Common Startup'; ID: CSIDL_COMMON_STARTUP),
    (Name: 'Controls'; ID: CSIDL_CONTROLS),
    (Name: 'Cookies'; ID: CSIDL_COOKIES),
    (Name: 'Desktop'; ID: CSIDL_DESKTOP),
    (Name: 'Desktop Directory'; ID: CSIDL_DESKTOPDIRECTORY),
    (Name: 'Drives'; ID: CSIDL_DRIVES),
    (Name: 'Favorites'; ID: CSIDL_FAVORITES),
    (Name: 'Fonts'; ID: CSIDL_FONTS),
    (Name: 'History'; ID: CSIDL_HISTORY),
    (Name: 'Internet'; ID: CSIDL_INTERNET),
    (Name: 'Internet Cache'; ID: CSIDL_INTERNET_CACHE),
    (Name: 'Network Neighborhood'; ID: CSIDL_NETHOOD),
    (Name: 'Network Top'; ID: CSIDL_NETWORK),
    (Name: 'Personal'; ID: CSIDL_PERSONAL),
    (Name: 'Printers'; ID: CSIDL_PRINTERS),
    (Name: 'Printer Links'; ID: CSIDL_PRINTHOOD),
    (Name: 'Programs'; ID: CSIDL_PROGRAMS),
    (Name: 'Recent Documents'; ID: CSIDL_RECENT),
    (Name: 'Send To'; ID: CSIDL_SENDTO),
    (Name: 'Start Menu'; ID: CSIDL_STARTMENU),
    (Name: 'Startup'; ID: CSIDL_STARTUP),
    (Name: 'Templates'; ID: CSIDL_TEMPLATES));

function CreateShellLink(const AppName, Desc: string; Dest: Integer): string;
function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean): string;
procedure GetShellLinkInfo(const LinkFile: WideString;
  var SLI: TShellLinkInfo);
procedure SetShellLinkInfo(const LinkFile: WideString;
  const SLI: TShellLinkInfo);

implementation

uses ComObj;

function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean): string;
var
  FilePath: array[0..MAX_PATH] of char;
```

LISTING 16.5 Continued

```
begin
  { Get path of selected location }
  SHGetSpecialFolderPathW(0, FilePath, Folder, CanCreate);
  Result := FilePath;
end;

function CreateShellLink(const AppName, Desc: string; Dest: Integer): string;
{ Creates a shell link for application or document specified in  }
{ AppName with description Desc.  Link will be located in folder }
{ specified by Dest, which is one of the string constants shown  }
{ at the top of this unit.  Returns the full path name of the    }
{ link file. }
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { The IShellLink implementer must also support the IPersistFile }
  { interface. Get an interface pointer to it. }
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName)));  // set link path to proper file
  if Desc <> '' then
    OleCheck(SL.SetDescription(PChar(Desc))); // set description
  { create a path location and filename for link file }
  LnkName := GetSpecialFolderPath(Dest, True) + '\' +
             ChangeFileExt(AppName, 'lnk');
  PF.Save(PWideChar(LnkName), True);          // save link file
  Result := LnkName;
end;

procedure GetShellLinkInfo(const LinkFile: WideString;
  var SLI: TShellLinkInfo);
{ Retrieves information on an existing shell link }
var
  SL: IShellLink;
  PF: IPersistFile;
  FindData: TWin32FindData;
  AStr: array[0..MAX_PATH] of char;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { The IShellLink implementer must also support the IPersistFile }
```

LISTING 16.5 Continued

```
  { interface. Get an interface pointer to it. }
  PF := SL as IPersistFile;
  { Load file into IPersistFile object }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_READ));
  { Resolve the link by calling the Resolve interface function. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_NO_UI));
  { Get all the info! }
  with SLI do
  begin
    OleCheck(SL.GetPath(AStr, MAX_PATH, FindData, SLGP_SHORTPATH));
    PathName := AStr;
    OleCheck(SL.GetArguments(AStr, MAX_PATH));
    Arguments := AStr;
    OleCheck(SL.GetDescription(AStr, MAX_PATH));
    Description := AStr;
    OleCheck(SL.GetWorkingDirectory(AStr, MAX_PATH));
    WorkingDirectory := AStr;
    OleCheck(SL.GetIconLocation(AStr, MAX_PATH, IconIndex));
    IconLocation := AStr;
    OleCheck(SL.GetShowCmd(ShowCmd));
    OleCheck(SL.GetHotKey(HotKey));
  end;
end;

procedure SetShellLinkInfo(const LinkFile: WideString;
  const SLI: TShellLinkInfo);
{ Sets information for an existing shell link }
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { The IShellLink implementer must also support the IPersistFile }
  { interface. Get an interface pointer to it. }
  PF := SL as IPersistFile;
  { Load file into IPersistFile object }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_SHARE_DENY_WRITE));
  { Resolve the link by calling the Resolve interface function. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_UPDATE or SLR_NO_UI));
  { Set all the info! }
  with SLI, SL do
  begin
    OleCheck(SetPath(PChar(PathName)));
```

LISTING 16.5 Continued

```
    OleCheck(SetArguments(PChar(Arguments)));
    OleCheck(SetDescription(PChar(Description)));
    OleCheck(SetWorkingDirectory(PChar(WorkingDirectory)));
    OleCheck(SetIconLocation(PChar(IconLocation), IconIndex));
    OleCheck(SetShowCmd(ShowCmd));
    OleCheck(SetHotKey(HotKey));
  end;
  PF.Save(PWideChar(LinkFile), True);   // save file
end;

end.
```

One method of `IShellLink` that has yet to be explained is the `Resolve()` method. `Resolve()` should be called after the `IPersistFile` interface of `IShellLink` is used to load a link file. This searches the specified link file and fills the `IShellLink` object with values specified in the file.

TIP

In the `GetShellLinkInfo()` function shown in Listing 16.5, notice the use of the `AStr` local array into which values are retrieved. This technique is used rather than using the `SetLength()` to allocate space for the strings—using `SetLength()` on so many strings would cause fragmentation of the application's heap. Using `AStr` as an intermediate prevents this from occurring. Additionally, because the length of the strings needs to be set only once, using `AStr` ends up being slightly faster.

A Sample Application

These functions and interfaces might be fun and all, but they're nothing without a nifty application in which to show them off. The Shell Link project allows you to do just that. The main form of this project is shown in Figure 16.6.

Listing 16.6 shows the main unit for this project, `Main.pas`. Listings 16.7 and 16.8 show `NewLinkU.pas` and `PickU.pas`, two supporting units for the project.

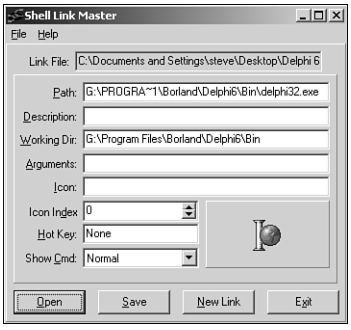


FIGURE 16.6
The Shell Link main form, showing one of the desktop links.

LISTING 16.6 `Main.pas`—Main for Shell Link Project

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    btnOpen: TButton;
    edLink: TEdit;
    btnNew: TButton;
    btnSave: TButton;
    Label3: TLabel;
    Panel2: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    edIcon: TEdit;
    edDesc: TEdit;
    edWorkDir: TEdit;
    edArg: TEdit;
```

LISTING 16.6 Continued

```
    cbShowCmd: TComboBox;
    hkHotKey: THotKey;
    speIcnIdx: TSpinEdit;
    pnlIconPanel: TPanel;
    imgIconImage: TImage;
    btnExit: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    NewLInk1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    edPath: TEdit;
    procedure btnOpenClick(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure edIconChange(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure GetControls(var SLI: TShellLinkInfo);
    procedure SetControls(const SLI: TShellLinkInfo);
    procedure ShowIcon;
    procedure OpenLinkFile(const LinkFileName: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses PickU, NewLinkU, AboutU, CommCtrl, ShellAPI;

type
  THotKeyRec = record
    Char, ModCode: Byte;
  end;

procedure TMainForm.SetControls(const SLI: TShellLinkInfo);
```

LISTING 16.6 Continued

```
{ Sets values of UI controls based on contents of SLI }
var
  Mods: THKModifiers;
begin
  with SLI do
  begin
    edPath.Text := PathName;
    edIcon.Text := IconLocation;
    { if icon name is blank and link is to exe, use exe name for icon }
    { path.  This is done because the icon index is ignored if the    }
    { icon path is blank, but an exe may contain more than one icon.  }
    if (IconLocation = '') and
      (CompareText(ExtractFileExt(PathName), 'EXE') = 0) then
      edIcon.Text := PathName;
    edWorkDir.Text := WorkingDirectory;
    edArg.Text := Arguments;
    speIcnIdx.Value := IconIndex;
    edDesc.Text := Description;
    { SW_* constants start at 1 }
    cbShowCmd.ItemIndex := ShowCmd - 1;
    { Hot key char in low byte }
    hkHotKey.HotKey := Lo(HotKey);
    { Figure out which modifier flags are in high byte }
    Mods := [];
    if (HOTKEYF_ALT and Hi(HotKey)) <> 0 then include(Mods, hkAlt);
    if (HOTKEYF_CONTROL and Hi(HotKey)) <> 0 then include(Mods, hkCtrl);
    if (HOTKEYF_EXT and Hi(HotKey)) <> 0 then include(Mods, hkExt);
    if (HOTKEYF_SHIFT and Hi(HotKey)) <> 0 then include(Mods, hkShift);
    { Set modifiers set }
    hkHotKey.Modifiers := Mods;
  end;
  ShowIcon;
end;

procedure TMainForm.GetControls(var SLI: TShellLinkInfo);
{ Gets values of UI controls and uses them to set values of SLI }
var
  CtlMods: THKModifiers;
  HR: THotKeyRec;
begin
  with SLI do
  begin
    PathName := edPath.Text;
    IconLocation := edIcon.Text;
```

LISTING 16.6 Continued

```
    WorkingDirectory := edWorkDir.Text;
    Arguments := edArg.Text;
    IconIndex := speIcnIdx.Value;
    Description := edDesc.Text;
    { SW_* constants start at 1 }
    ShowCmd := cbShowCmd.ItemIndex + 1;
    { Get hot key character }
    word(HR) := hkHotKey.HotKey;
    { Figure out which modifier keys are being used }
    CtlMods := hkHotKey.Modifiers;
    with HR do begin
      ModCode := 0;
      if (hkAlt in CtlMods) then ModCode := ModCode or HOTKEYF_ALT;
      if (hkCtrl in CtlMods) then ModCode := ModCode or HOTKEYF_CONTROL;
      if (hkExt in CtlMods) then ModCode := ModCode or HOTKEYF_EXT;
      if (hkShift in CtlMods) then ModCode := ModCode or HOTKEYF_SHIFT;
    end;
    HotKey := word(HR);
  end;
end;

procedure TMainForm.ShowIcon;
{ Retrieves icon from appropriate file and shows in IconImage }
var
  HI: THandle;
  IcnFile: string;
  IconIndex: word;
begin
  { Get name of icon file }
  IcnFile := edIcon.Text;
  { If blank, use the exe name }
  if IcnFile = '' then
    IcnFile := edPath.Text;
  { Make sure file exists }
  if FileExists(IcnFile) then
  begin
    IconIndex := speIcnIdx.Value;
    { Extract icon from file }
    HI := ExtractAssociatedIcon(hInstance, PChar(IcnFile), IconIndex);
    { Assign icon handle to IconImage }
    imgIconImage.Picture.Icon.Handle := HI;
  end;
end;
```

LISTING 16.6 Continued

```
procedure TMainForm.OpenLinkFile(const LinkFileName: string);
{ Opens a link file, get info, and displays info in UI }
var
  SLI: TShellLinkInfo;
begin
  edLink.Text := LinkFileName;
  try
    GetShellLinkInfo(LinkFileName, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while opening link', mtError, [mbOk], 0);
  end;
  SetControls(SLI);
end;

procedure TMainForm.btnOpenClick(Sender: TObject);
{ OnClick handler for OpenBtn }
var
  LinkFile: String;
begin
  if GetLinkFile(LinkFile) then
    OpenLinkFile(LinkFile);
end;

procedure TMainForm.btnNewClick(Sender: TObject);
{ OnClick handler for NewBtn }
var
  FileName: string;
  Dest: Integer;
begin
  if GetNewLinkName(FileName, Dest) then
    OpenLinkFile(CreateShellLink(FileName, '', Dest));
end;

procedure TMainForm.edIconChange(Sender: TObject);
{ OnChange handler for IconEd and IcnIdxEd }
begin
  ShowIcon;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
{ OnClick handler for SaveBtn }
var
  SLI: TShellLinkInfo;
```

LISTING 16.6 Continued

```
begin
  GetControls(SLI);
  try
    SetShellLinkInfo(edLink.Text, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while setting info', mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.btnExitClick(Sender: TObject);
{ OnClick handler for ExitBtn }
begin
  Close;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ OnClick handler for Help|About menu item }
begin
  AboutBox;
end;

end.
```

LISTING 16.7 `NewLinkU.pas`—Unit with Form That Helps Create New Link

```
unit NewLinkU;

interface

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

type
  TNewLinkForm = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edLinkTo: TEdit;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    sbOpen: TSpeedButton;
    OpenDialog: TOpenDialog;
```

LISTING 16.7 Continued

```
    procedure sbOpenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;

implementation

uses WinShell;

{$R *.DFM}

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;
{ Gets file name and destination folder for a new shell link. }
{ Only modifies params if Result = True. }
begin
  with TNewLinkForm.Create(Application) do
  try
    cbLocation.ItemIndex := 0;
    Result := ShowModal = mrOk;
    if Result then
    begin
      LinkTo := edLinkTo.Text;
      Dest := cbLocation.ItemIndex;
    end;
  finally
    Free;
  end;
end;

procedure TNewLinkForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    edLinkTo.Text := OpenDialog.FileName;
end;

procedure TNewLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

LISTING 16.8 PickU.pas—Unit with Form that Enables User to Choose Link Location

```
unit PickU;

interface

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

type
  TLinkForm = class(TForm)
    lbLinkFiles: TFileListBox;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    Label1: TLabel;
    procedure lbLinkFilesDblClick(Sender: TObject);
    procedure cbLocationChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetLinkFile(var S: String): Boolean;

implementation

{$R *.DFM}

uses WinShell, ShlObj;

function GetLinkFile(var S: String): Boolean;
{ Returns link file name in S. }
{ Only modifies S when Result is True. }
begin
  with TLinkForm.Create(Application) do
    try
      { Make sure location is selected }
      cbLocation.ItemIndex := 0;
      { Get path of selected location }
      cbLocationChange(nil);
      Result := ShowModal = mrOk;
      { Return full pathname for link file }
      if Result then
        S := lbLinkFiles.Directory + '\' +
          lbLinkFiles.Items[lbLinkFiles.ItemIndex];
    finally
```

LISTING 16.8 Continued

```
      Free;
    end;
end;

procedure TLinkForm.lbLinkFilesDblClick(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TLinkForm.cbLocationChange(Sender: TObject);
var
  Folder: Integer;
begin
  { Get path of selected location }
  Folder := SpecialFolders[cbLocation.ItemIndex].ID;
  lbLinkFiles.Directory := GetSpecialFolderPath(Folder, False);
end;

procedure TLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

Shell Extensions

For the ultimate in extensibility, the Windows shell provides a means for you to develop code that executes from within the shell's own process and namespace. *Shell extensions* are implemented as in-process COM servers that are created and used by the shell.

NOTE

Because shell extensions are COM servers at heart, understanding them requires a basic understand of COM. If your COM knowledge needs brushing up, Chapter 15, "COM Development," provides this foundation.

Several types of shell extensions are available to deal with a variety of the shell's aspects. Also known as a *handler*, a shell extension must implement one or more COM interfaces. The shell supports the following types of shell extensions:

- *Copy hook handlers* implement the `ICopyHook` interface. These shell extensions allow you to receive notifications whenever a folder is copied, deleted, moved, or renamed and to optionally prevent the operation from occurring.
- *Context menu handlers* implement the `IContextMenu` and `IShellExtInit` interfaces. These shell extensions enable you to add items to the context menu of a particular file object in the shell.
- *Drag-and-drop handlers* also implement the `IContextMenu` and `IShellExtInit` interfaces. These shell extensions are almost identical in implementation to context menu handlers, except that they're invoked when a user drags an object and drops it to a new location.
- *Icon handlers* implement the `IExtractIcon` and `IPersistFile` interfaces. Icon handlers allow you to provide different icons for multiple instances of the same type of file object.
- *Property sheet handlers* implement the `IShellPropSheetExt` and `IShellExtInit` interfaces, and they allow you to add pages to the properties dialog associated with a file type.
- *Drop target handlers* implement the `IDropTarget` and `IPersistFile` interfaces. These shell extensions allow you to control what happens when you drop one shell object on another.
- *Data object handlers* implement the `IDataObject` and `IPersistFile` interfaces, and they supply the data object used when files are being dragged and dropped or copied and pasted.

Debugging Shell Extensions

Before we get into the subject of actually writing shell extensions, consider the question of debugging shell extensions. Because shell extensions execute from within the shell's own process, how is it possible to "hook into" the shell in order to debug your shell extension?

The solution to the problem is based on the fact that the shell is an executable (not very different from any other application) called `explorer.exe`. `Explorer.exe` has a property, however, that is kind of unique: The first instance of `explorer.exe` will invoke the shell. Subsequent instances will simply invoke additional "Explorer" windows in the shell.

Using a little-known trick in the shell, it's possible to close the shell without closing Windows. Follow these steps to debug your shell extensions in Delphi:

1. Make `explorer.exe` the host application for your shell extension in the Run, Parameters dialog box. Be sure to include the full path (that is, `c:\windows\explorer.exe`).
2. From the shell's Start menu, select Shut Down. This will invoke the Shut Down Windows dialog box.
3. In the Shut Down Windows dialog box, hold down Ctrl+Alt+Shift and click the No button. This will close the shell without closing Windows.
4. Using Alt+Tab, switch back to Delphi and run the shell extension. This will invoke a new copy of the shell running under the Delphi debugger. You can now set breakpoints in your code and debug as usual.
5. When you're ready to close Windows, you can still do so properly without the use of the shell: Use Ctrl+Esc to invoke the Tasks window and then select Windows, Shutdown Windows to close Windows.

The remainder of this chapter is dedicated to showing a cross section of the shell extensions just described. You'll learn about copy hook handlers, context menu handlers, and icon handlers.

The COM Object Wizard

Before discussing each of the shell extension DLLs, we should first mention a bit about how they're created. Because shell extensions are in-process COM servers, you can let the Delphi IDE do most of the grunt work in creating the source code for you. Work begins for all the shell extensions with the same two steps:

1. Select ActiveX Library from the ActiveX page of the New Items dialog box. This will create a new COM server DLL into which you can insert COM objects.
2. Select COM Object from the ActiveX page of the New Items dialog boxes. This will invoke the COM Server Wizard. In the wizard's dialog box, enter a name and description for your shell extension and select the Apartment threading model. Click OK, and a new unit containing the code for your COM object will be generated.

Copy Hook Handlers

As mentioned earlier, copy hook shell extensions allow you to install a handler that receives notifications whenever a folder is copied, deleted, moved, or renamed. After receiving this notification, the handler can optionally prevent the operation from occurring. Note that the handler is only called for folder and printer objects; it's not called for files and other objects.

The first step in creating a copy hook handler is to create an object that descends from `TComObject` and implements the `ICopyHook` interface. This interface is defined in the `ShlObj` unit as follows:

```
type
  ICopyHook = interface(IUnknown)
    ['{000214EF-0000-0000-C000-000000000046}']
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
      dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

The `CopyCallback()` Method

As you can see, `ICopyHook` is a pretty simple interface, and it implements only one function: `CopyCallback()`. This function will be called whenever a shell folder is manipulated. The following paragraphs describe the parameters for this function.

`Wnd` is the handle of the window the copy hook handler should use as the parent for any windows it displays. `wFunc` indicates the operation being performed. This can be any one of the values shown in Table 16.5.

TABLE 16.5 The `wFunc` Values for `CopyCallback()`

Constant	*Value*	*Meaning*
`FO_COPY`	`$2`	Copies the file specified by `pszSrcFile` to the location specified by `pszDestFile`.
`FO_DELETE`	`$3`	Deletes the file specified by `pszSrcFile`.
`FO_MOVE`	`$1`	Moves the file specified by `pszSrcFile` to the location specified by `pszDestFile`.
`FO_RENAME`	`$4`	Renames the file specified by `pszSrcFile`.
`PO_DELETE`	`$13`	Deletes the printer specified by `pszSrcFile`.
`PO_PORTCHANGE`	`$20`	Changes the printer port. The `pszSrcFile` and `pszDestFile` parameters contain double null-terminated lists of strings. Each list contains the printer name followed by the port name. The port name in `pszSrcFile` is the current printer port, and the port name in `pszDestFile` is the new printer port.
`PO_RENAME`	`$14`	Renames the printer specified by `pszSrcFile`.
`PO_REN_PORT`	`$34`	A combination of `PO_RENAME` and `PO_PORTCHANGE`.

wFlags holds the flags that control the operation. This parameter can be a combination of the values shown in Table 16.6.

TABLE 16.6 The wFlags Values for CopyCallback()

Constant	*Value*	*Meaning*
FOF_ALLOWUNDO	$40	Preserves undo information (when possible).
FOF_MULTIDESTFILES	$1	The SHFileOperation() function specifies multiple destination files (one for each source file) rather than one directory where all the source files are to be deposited. A copy hook handler typically ignores this value.
FOF_NOCONFIRMATION	$10	Responds with "Yes to All" for any dialog box that's displayed.
FOF_NOCONFIRMMKDIR	$200	Does not confirm the creation of any needed directories if the operation requires a new directory to be created.
FOF_RENAMEONCOLLISION	$8	Gives the file being operated on a new name (such as "Copy #1 of. . .") in a copy, move, or rename operation when a file with the target name already exists.
FOF_SILENT	$4	Does not display a progress dialog box.
FOF_SIMPLEPROGRESS	$100	Displays a progress dialog box, but the dialog box doesn't show the names of the files.

pszSourceFile is the name of the source folder, dwSrcAttribs holds the attributes of the source folder, pszDestFile is the name of the destination folder, and dwDestAttribs holds the attributes of the destination folder.

Unlike most methods, this interface doesn't return an OLE result code. Instead, it must return one of the values listed in Table 16.7, as defined in the Windows unit.

TABLE 16.7 The wFlags Values for CopyCallback()

Constant	*Value*	*Meaning*
IDYES	6	Allows the operation
IDNO	7	Prevents the operation on this file but continues with any other operations (for example, a batch copy operation)

TABLE 16.7 Continued

Constant	*Value*	*Meaning*
IDCANCEL	2	Prevents the current operation and cancels any pending operations

TCopyHook Implementation

Being an object that implements one interface with one method, there isn't much to TCopyHook:

```
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar;
      dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT;
      stdcall;
  end;
```

The implementation of the CopyCallback() method is also small. The MessageBox() API function is called to confirm whatever operation is being attempted. Conveniently, the return value for MessageBox() will be the same as the return value for this method:

```
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
begin
  // confirm operation
  Result := MessageBox(Wnd,  PChar(Format(MyMessage, [pszSrcFile])),
    'DDG Shell Extension', MB_YESNO);
end;
```

TIP

You might wonder why the MessageBox() API function is used to display a message rather than using a Delphi function such as MessageDlg() or ShowMessage(). The reason is simple: size and efficiency. Calling any function out of the Dialogs or Forms unit would cause a great deal of VCL to be linked into the DLL. By keeping these units out of the uses clause, the shell extension DLL weighs in at a svelte 70KB.

Believe it or not, that's all there is to the `TCopyHook` object itself. However, there's still one major detail to work through before calling it a day: The shell extension must be registered with the System Registry before it will function.

Registration

In addition to the normal registration required of any COM server, a copy hook handler must have an additional Registry entry under

```
HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers
```

Furthermore, Windows NT requires that all shell extensions be registered as approved shell extensions under

```
HKEY_LOCAL_MACHINE\ SOFTWARE\Microsoft\Windows\CurrentVersion
➥\Shell Extensions\Approved
```

You can take several approaches to registering shell extensions: They can be registered via a REG file or through an installation program. The shell extension DLL, itself, can be self-registering. Although it might be just a bit more work, the best solution is to make each shell extension DLL self-registering. This is cleaner because it makes your shell extension a one-file, self-contained package.

As you learned in Chapter 15, COM objects are always created from class factories. Within the VCL framework, class factory objects are also responsible for registering the COM object they will create. If a COM object requires custom Registry entries (as is the case with a shell extension), setting up these entries is just a matter of overriding the class factory's `UpdateRegistry()` method. Listing 16.9 shows the completed `CopyMain` unit, which includes a specialized class factory used to perform custom registration.

LISTING 16.9 `CopyMain.pas`—Main Unit for Copy Hook Implementation

```
unit CopyMain;

interface

uses Windows, ComObj, ShlObj;

type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
      pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

LISTING 16.9 Continued

```
  TCopyHookFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, Registry;

{ TCopyHook }

// This is the method which is called by the shell for folder operations
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
begin
  // confirm operation
  Result := MessageBox(Wnd,  PChar(Format(MyMessage, [pszSrcFile])),
    'DDG Shell Extension', MB_YESNO);
end;

{ TCopyHookFactory }

function TCopyHookFactory.GetProgID: string;
begin
  // ProgID not needed for shell extension
  Result := '';
end;

procedure TCopyHookFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
    // add shell extension clsid to CopyHookHandlers Reg entry
    CreateRegKey('directory\shellex\CopyHookHandlers\' + ClassName, '',
      ClsID)
```

LISTING 16.9 Continued

```
  else
    DeleteRegKey('directory\shellex\CopyHookHandlers\' + ClassName);
end;

procedure TCopyHookFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// This registry entry is required in order for the extension to
// operate correctly under Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
➥Extensions\Approved';

begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID = '{66CD5F60-A044-11D0-A9BF-00A016E3867F}';

initialization
  TCopyHookFactory.Create(ComServer, TCopyHook, CLSID_CopyHook,
    'DDG_CopyHook', 'DDG Copy Hook Shell Extension Example',
    ciMultiInstance, tmApartment);
end.
```

What makes the `TCopyHookFactory` class factory work is the fact that an instance of it, rather than the usual `TComObjectFactory`, is being created in the `initialization` part of the unit. Figure 16.7 shows what happens when you try to rename a folder in the shell after the copy hook shell extension DLL is installed.

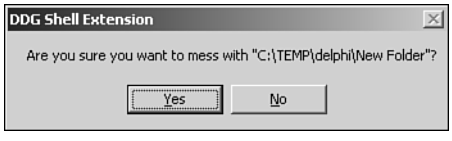


FIGURE 16.7
The copy hook handler in action.

Context Menu Handlers

Context menu handlers enable you to add items to the local menu that are associated with file objects in the shell. A sample local menu for an EXE file is shown in Figure 16.8.

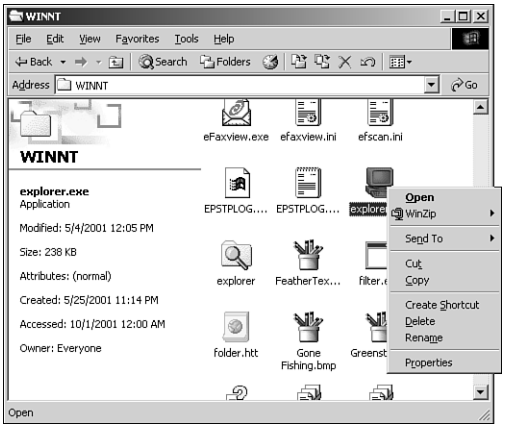


FIGURE 16.8
The shell local menu for an EXE file.

Context menu shell extensions work by implementing the `IShellExtInit` and `IContextMenu` interfaces. In this case, we'll implement these interfaces to create a context menu handler for Borland Package Library (BPL) files; the local menu for package files in the shell will provide an option for obtaining package information. This context menu handler object will be called `TContextMenu`, and, like the copy hook handler, `TContextMenu` will descend from `TComObject`.

IShellExtInit

The `IShellExtInit` interface is used to initialize a shell extension. This interface is defined in the `ShlObj` unit as follows:

```
type
  IShellExtInit = interface(IUnknown)
    ['{000214E8-0000-0000-C000-000000000046}']
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; stdcall;
  end;
```

`Initialize()`, being the only method of this interface, is called to initialize the context menu handler. The following paragraphs describe the parameters for this method.

`pidlFolder` is a pointer to a `PItemIDList` (item identifier list) structure for the folder that contains the item whose context menu is being displayed. `lpdobj` holds the `IDataObject` interface

object used to retrieve the objects being acted upon. `hkeyProgID` contains the Registry key for the file object or folder type.

The implementation for this method is shown in the following code. Upon first glance, the code might look complex, but it really boils down to three things: a call to `lpobj.GetData()` to obtain data from `IDataObject` and two calls to `DragQueryFile()` (one call to obtain the number of files and the other to obtain the filename). The filename is stored in the object's `FFileName` field. Here's the code:

```
function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
  hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // Fail the call if lpdobj is nil.
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    // Render the data referenced by the IDataObject pointer to an HGLOBAL
    // storage medium in CF_HDROP format.
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
      // If only one file is selected, retrieve the file name and store it in
      // szFile. Otherwise fail the call.
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
```

```
    finally
      ReleaseStgMedium(medium);
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

IContextMenu

The `IContextMenu` interface is used to manipulate the pop-up menu associated with a file in the shell. This interface is defined in the `ShlObj` unit as follows:

```
type
  IContextMenu = interface(IUnknown)
    ['{000214E4-0000-0000-C000-000000000046}']
    function QueryContextMenu(Menu: HMENU;
      indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  end;
```

After the handler has been initialized through the `IShellExtInit` interface, the next method to be called is `IContextMenu.QueryContextMenu()`. The parameters passed to this method include a menu handle, the index at which to insert the first menu item, the minimum and maximum values for menu item IDs, and flags that indicate menu attributes. The following `TContextMenu` implementation of this method adds a menu item with the text "Package Info. . ." to the menu handle passed in the `Menu` parameter (note that the return value for `QueryContextMenu()` is the index of the last menu item inserted plus one):

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
  idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // Add one menu item to context menu
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION, idCmdFirst,
    'Package Info...');
  // Return index of last inserted item + 1
  Result := FMenuIdx + 1;
end;
```

The next method called by the shell is `GetCommandString()`. This method is intended to retrieve the language-independent command string or help string for a particular menu item. The parameters for this method include the menu item offset, flags indicating the type of information to receive, a reserved parameter, and a string buffer and buffer size. The following `TContextMenu` implementation of this method only needs to deal with providing the help string for the menu item:

```
function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    // make sure menu index is correct, and shell is asking for help string
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      // return help string for menu item
      StrLCopy(pszName, 'Get information for the selected package.', cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

When you click the new item in the context menu, the shell will call the `InvokeCommand()` method. The method accepts a `TCMInvokeCommandInfo` record as a parameter. This record is defined in the `ShlObj` unit as follows:

```
type
  PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
  TCMInvokeCommandInfo = packed record
    cbSize: DWORD;        { must be SizeOf(TCMInvokeCommandInfo) }
    fMask: DWORD;         { any combination of CMIC_MASK_* }
    hwnd: HWND;           { might be NULL (indicating no owner window) }
    lpVerb: LPCSTR;       { either a string of MAKEINTRESOURCE(idOffset) }
    lpParameters: LPCSTR; { might be NULL (indicating no parameter) }
    lpDirectory: LPCSTR;  { might be NULL (indicating no specific directory) }
    nShow: Integer;       { one of SW_ values for ShowWindow() API }
    dwHotKey: DWORD;
    hIcon: THandle;
  end;
```

The low word or the `lpVerb` field will contain the index of the menu item selected. Here's the implementation of this method:

```
function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
begin
  Result := S_OK;
  try
    // Make sure we are not being called by an application
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    // Execute the command specified by lpici.lpVerb.
    // Return E_INVALIDARG if we are passed an invalid argument number.
```

```
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Error obtaining package information.', 'Error',
      MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;
```

If all goes well, the `ExecutePackInfoApp()` function is called to invoke the `PackInfo.exe` application, which displays various information about a package. We won't go into the particulars of that application right now; however, it's discussed in detail on the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book in Chapter 13, "Hard-Core Techniques."

Registration

Context menu handlers must be registered under

`HKEY_CLASSES_ROOT\<file type>\shellex\ContextMenuHandlers`

in the System Registry. Following the model of the copy hook extension, registration capability is added to the DLL by creating a specialized `TComObject` descendant. The object is shown in Listing 16.10 along with the complete source code for the unit containing `TContextMenu`. Figure 16.9 shows the local menu for the BPL file with the new item, and Figure 16.10 shows the `PackInfo.exe` window as invoked by the context menu handler.

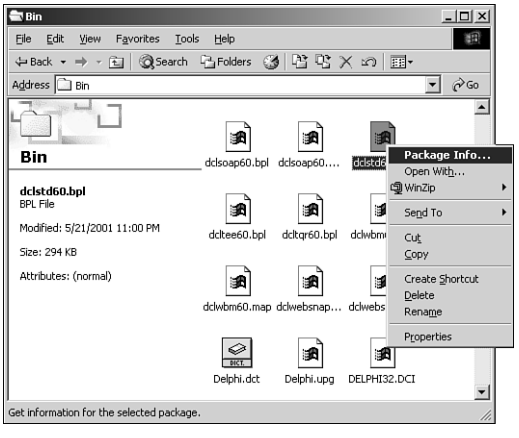


FIGURE 16.9

The context menu handler in action.

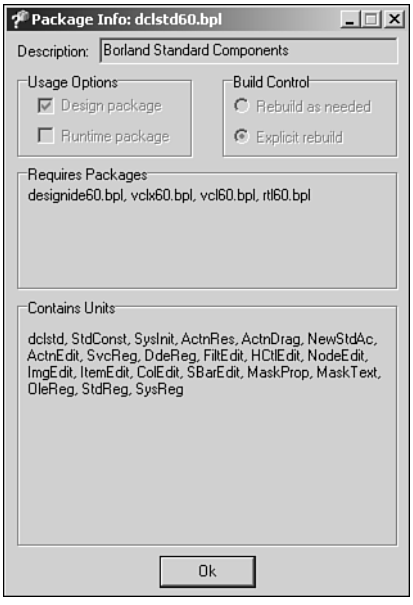


FIGURE 16.10
Obtaining package information from the context menu handler.

LISTING 16.10 `ContMain.pas`—Main Unit for Context Menu Handler Implementation

```
unit ContMain;

interface

uses Windows, ComObj, ShlObj, ActiveX;

type
  TContextMenu = class(TComObject, IContextMenu, IShellExtInit)
  private
    FFileName: array[0..MAX_PATH] of char;
    FMenuIdx: UINT;
  protected
    // IContextMenu methods
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst, idCmdLast,
      uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
    // IShellExtInit method
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
```

LISTING 16.10 Continued

```
      hKeyProgID: HKEY): HResult; reintroduce; stdcall;
  end;

  TContextMenuFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, ShellAPI, Registry;

procedure ExecutePackInfoApp(const FileName: string; ParentWnd: HWND);
const
  SPackInfoApp = '%sPackInfo.exe';
  SCmdLine = '"%s" %s';
  SErrorStr = 'Failed to execute PackInfo:'#13#10#13#10;
var
  PI: TProcessInformation;
  SI: TStartupInfo;
  ExeName, ExeCmdLine: string;
  Buffer: array[0..MAX_PATH] of char;
begin
  // Get directory of this DLL.  Assume EXE being executed is in same dir.
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  ExeName := Format(SPackInfoApp, [ExtractFilePath(Buffer)]);
  ExeCmdLine := Format(SCmdLine, [ExeName, FileName]);
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  if not CreateProcess(PChar(ExeName), PChar(ExeCmdLine), nil, nil, False,
    0, nil, nil, SI, PI) then
    MessageBox(ParentWnd, PChar(SErrorStr + SysErrorMessage(GetLastError)),
      'Error', MB_OK or MB_ICONERROR);
end;

{ TContextMenu }

{ TContextMenu.IContextMenu }

function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
  idCmdLast, uFlags: UINT): HResult;
```

LISTING 16.10 Continued

```
begin
  FMenuIdx := indexMenu;
  // Add one menu item to context menu
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION, idCmdFirst,
    'Package Info...');
  // Return index of last inserted item + 1
  Result := FMenuIdx + 1;
end;

function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
begin
  Result := S_OK;
  try
    // Make sure we are not being called by an application
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    // Execute the command specified by lpici.lpVerb.
    // Return E_INVALIDARG if we are passed an invalid argument number.
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Error obtaining package information.', 'Error',
      MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;

function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    // make sure menu index is correct, and shell is asking for help string
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      // return help string for menu item
      StrLCopy(pszName, 'Get information for the selected package.', cchMax)
    else
      Result := E_INVALIDARG;
  except
```

LISTING 16.10 Continued

```
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenu.IShellExtInit }

function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
  hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // Fail the call if lpdobj is nil.
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    // Render the data referenced by the IDataObject pointer to an HGLOBAL
    // storage medium in CF_HDROP format.
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
      // If only one file is selected, retrieve the file name and store it in
      // szFile. Otherwise fail the call.
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
```

LISTING 16.10 Continued

```
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenuFactory }

function TContextMenuFactory.GetProgID: string;
begin
  // ProgID not required for context menu shell extension
  Result := '';
end;

procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // must register .bpl as a file type
    CreateRegKey('.bpl', '', 'DelphiPackageLibrary');
    // register this DLL as a context menu handler for .bpl files
    CreateRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName, '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName);
  end;
end;

procedure TContextMenuFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// This registry entry is required in order for the extension to
// operate correctly under Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
➥Shell Extensions\Approved';

begin
  with TRegistry.Create do
```

LISTING 16.10 Continued

```
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID = '{7C5E74A0-D5E0-11D0-A9BF-E886A83B9BE5}';

initialization
  TContextMenuFactory.Create(ComServer, TContextMenu, CLSID_CopyHook,
    'DDG_ContextMenu', 'DDG Context Menu Shell Extension Example',
    ciMultiInstance, tmApartment);
end.
```

Icon Handlers

Icon handlers enable you to cause different icons to be used for multiple instances of the same type of file. In this example, the `TIconHandler` icon handler object provides different icons for different types of Borland Package (BPL) files. Depending on whether a package is runtime, design time, both, or none, a different icon will be displayed in a shell folder.

Package Flags

Before getting into the implementations of the interfaces necessary for this shell extension, take a moment to examine the method that determines the type of a particular package file. The method returns `TPackType`, which is defined as follows:

```
TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);
```

Now here's the method:

```
function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: 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(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // mask off all but design and run flags, and return result
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;
```

This method works by calling the `GetPackageInfo()` method from the `SysUtils` unit to obtain the package flags. An interesting point to note concerning performance optimization is that the `LoadLibraryEx()` API function is called rather than Delphi's `LoadPackage()` procedure to load the package library. Internally, the `LoadPackage()` procedure calls the `LoadLibrary()` API to load the BPL and then calls `InitializePackage()` to execute the initialization code for each of the units in the package. Because all we want to do is get the package flags and they reside in a resource linked to the BPL, we can safely load the package with `LoadLibraryEx()` using the `LOAD_LIBRARY_AS_DATAFILE` flag.

Icon Handler Interfaces

As mentioned earlier, icon handlers must support both the `IExtractIcon` (defined in `ShlObj`) and `IPersistFile` (defined in the `ActiveX` unit) interfaces. These interfaces are shown here:

```
type
  IExtractIcon = interface(IUnknown)
    ['{000214EB-0000-0000-C000-000000000046}']
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax: UINT;
      out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
      out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult; stdcall;
  end;

  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
```

```
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;
```

Although this might look like a lot of work, it's really not; only two of these methods actually have to be implemented. The first file that must be implemented is `IPersistFile.Load()`. This is the method that's called to initialize the shell extension, and in it, you must save the filename passed via the `pszFileName` parameter. Here's the `TExtractIcon` implementation of this method:

```
function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  // this method is called to initialized the icon handler shell
  // extension.  We must save the file name which is passed in pszFileName
  FFileName := pszFileName;
  Result := S_OK;
end;
```

The other method that must be implemented is `IExtractIcon.GetIconLocation()`. The parameters for this method are discussed in the following paragraphs.

`uFlags` indicates the type of icon to be displayed. This parameter can be `0`, `GIL_FORSHELL`, or `GIL_OPENICON`. `GIL_FORSHELL` means that the icon is to be displayed in a shell folder. `GIL_OPENICON` means that the icon should be in the "open" state if images for both the open and closed states are available. If this flag isn't specified, the icon should be in the normal, or "closed," state. This flag is typically used for folder objects.

`szIconFile` is the buffer to receive the icon location, and `cchMax` is the size of the buffer. `piIndex` is an integer that receives the icon index, which further describes the icon location. `pwFlags` receives zero or more of the values shown in Table 16.8.

TABLE 16.8 The `pwFlags` Values for `GetIconLocation()`

Flag	*Meaning*
`GIL_DONTCACHE`	The physical image bits for this icon shouldn't be cached by the caller. This distinction is important to consider because a `GIL_DONTCACHELOCATION` flag might be introduced in future versions of the shell.
`GIL_NOTFILENAME`	The location isn't a filename/index pair. Callers that decide to extract the icon from the location must call this object's `IExtractIcon.Extract()` method to obtain the desired icon images.

TABLE 16.8 Continued

Flag	*Meaning*
`GIL_PERCLASS`	All objects of this class have the same icon. This flag is used internally by the shell. Typical implementations of `IExtractIcon` don't require this flag because it implies that an icon handler is not required to resolve the icon on a per-object basis. The recommended method for implementing per-class icons is to register a default icon for the class.
`GIL_PERINSTANCE`	Each object of this class has its own icon. This flag is used internally by the shell to handle cases such as `setup.exe`, where more than one object with identical names might be known to the shell and use different icons. Typical implementations of `IExtractIcon` don't require this flag.
`GIL_SIMULATEDOC`	The caller should create a document icon using the specified icon.

The `TIconHandler` implementation of `GetIconLocation()` is shown here:

```
function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    // return this DLL for name of module to find icon
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // tell shell not to cache image bits, in case icon changes
    // and that each instance may have its own icon
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    // icon index coincides with TPackType
    piIndex := Ord(GetPackageType);
  except
    // if there's an error, use the default package icon
    piIndex := Ord(ptNone);
  end;
end;
```

The icons are linked into the shell extension DLL as a resource file, so the name of the current file, as returned by `GetModuleFileName()`, is written to the `szIconFile` buffer. Also, the icons are arranged in such a way that the index of an icon for a package type corresponds to the package type's index into the `TPackType` enumeration, so the return value of `GetPackageType()` is assigned to `piIndex`.

Registration

Icon handlers must be registered under the

```
HKEY_CLASSES_ROOT\<file type>\shellex\IconHandler
```

key in the Registry. Again, a descendant of `TComObjectFactory` is created to deal with the registration of this shell extension. This is shown in Listing 16.11 along with the rest of the source code for the icon handler.

Figure 16.11 shows a shell folder containing packages of different types. Notice the different icons for different types of packages.

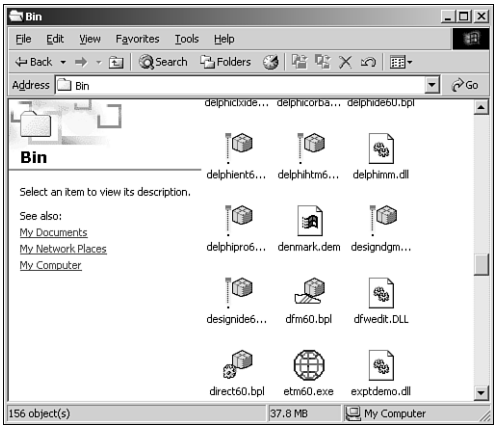


FIGURE 16.11
The result of using the icon handler.

LISTING 16.11 `IconMain.pas`—Main Unit for Icon Handler Implementation

```
unit IconMain;

interface

uses Windows, ActiveX, ComObj, ShlObj;

type
  TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);

  TIconHandler = class(TComObject, IExtractIcon, IPersistFile)
  private
    FFileName: string;
    function GetPackageType: TPackType;
```

LISTING 16.11 Continued

```
  protected
    // IExtractIcon methods
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax: UINT;
      out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
      out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult; stdcall;
    // IPersist method
    function GetClassID(out classID: TCLSID): HResult; stdcall;
    // IPersistFile methods
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;

  TIconHandlerFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses SysUtils, ComServ, Registry;

{ TIconHandler }

procedure PackInfoProc(const Name: string; NameType: TNameType; Flags: Byte;
  Param: Pointer);
begin
  // we don't need to implement this procedure because we are only
  // interested in package flags, not contained units and required pkgs.
end;

function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  // Since we only need to get into the package's resources,
```

LISTING 16.11 Continued

```
  // LoadLibraryEx with LOAD_LIBRARY_AS_DATAFILE provides a speed-
  // efficient means for loading the package.
  PackMod := LoadLibraryEx(PChar(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // mask off all but design and run flags, and return result
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;

{ TIconHandler.IExtractIcon }

function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    // return this DLL for name of module to find icon
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // tell shell not to cache image bits, in case icon changes
    // and that each instance may have its own icon
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    // icon index coincides with TPackType
    piIndex := Ord(GetPackageType);
  except
    // if there's an error, use the default package icon
    piIndex := Ord(ptNone);
  end;
end;

function TIconHandler.Extract(pszFile: PAnsiChar; nIconIndex: UINT;
  out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult;
```

LISTING 16.11 Continued

```
begin
  // This method only needs to be implemented if the icon is stored in
  // some type of user-defined data format.  Since our icon is in a
  // plain old DLL, we just return S_FALSE.
  Result := S_FALSE;
end;

{ TIconHandler.IPersist }

function TIconHandler.GetClassID(out classID: TCLSID): HResult;
begin
  // this method is not called for icon handlers
  Result := E_NOTIMPL;
end;

{ TIconHandler.IPersistFile }

function TIconHandler.IsDirty: HResult;
begin
  // this method is not called for icon handlers
  Result := S_FALSE;
end;

function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  // this method is called to initialized the icon handler shell
  // extension.  We must save the file name which is passed in pszFileName
  FFileName := pszFileName;
  Result := S_OK;
end;

function TIconHandler.Save(pszFileName: POleStr; fRemember: BOOL): HResult;
begin
  // this method is not called for icon handlers
  Result := E_NOTIMPL;
end;

function TIconHandler.SaveCompleted(pszFileName: POleStr): HResult;
begin
  // this method is not called for icon handlers
  Result := E_NOTIMPL;
end;

function TIconHandler.GetCurFile(out pszFileName: POleStr): HResult;
```

LISTING 16.11 Continued

```
begin
  // this method is not called for icon handlers
  Result := E_NOTIMPL;
end;

{ TIconHandlerFactory }

function TIconHandlerFactory.GetProgID: string;
begin
  // ProgID not required for context menu shell extension
  Result := '';
end;

procedure TIconHandlerFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // must register .bpl as a file type
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    // register this DLL as an icon handler for .bpl files
    CreateRegKey('BorlandPackageLibrary\shellex\IconHandler', '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\IconHandler');
  end;
end;

procedure TIconHandlerFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// This registry entry is required in order for the extension to
// operate correctly under Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\
➥Shell Extensions\Approved';

begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
```

LISTING 16.11 Continued

```
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_IconHandler: TGUID = '{ED6D2F60-DA7C-11D0-A9BF-90D146FC32B3}';

initialization
  TIconHandlerFactory.Create(ComServer, TIconHandler, CLSID_IconHandler,
    'DDG_IconHandler', 'DDG Icon Handler Shell Extension Example',
    ciMultiInstance, tmApartment);
end.
```

InfoTip Handlers

Introduced in the Windows 2000 shell, InfoTip handlers provide the ability to create custom pop-up *InfoTips* (also called *ToolTips* in Delphi) when the mouse is placed over the icon representing a file in the shell. The default InfoTip displayed by the shell contains the name of the file, the type of file (as determined based on its extension), and the file size. InfoTip handlers are handy when you want to display more than this rather limited and generic bit of file information to the user at a glance.

For Delphi developers, a great case in point is package files. Although we all know that package files are composed of one or more units, it's impossible to know at a glance exactly which units are contained within. Earlier in this chapter, you saw a context menu handler that provides this information by choosing an option from a local menu, causing an external application to be launched. Now you'll see how to get this information even more easily, without the use of an external program.

InfoTip Handler Interfaces

InfoTip handlers must implement the `IQueryInfo` and `IPersistFile` interfaces. You already learned about `IPersistFile` in the discussions on shell links and icon handlers earlier in this chapter, and it is used in this case to obtain the name of the file in question. `IQueryInfo` is a relatively simple interface containing two methods, and it is defined in the `ShlObj` unit as shown here:

```
type
  IQueryInfo = interface(IUnknown)
```

```
    [SID_IQueryInfo]
    function GetInfoTip(dwFlags: DWORD; var ppwszTip: PWideChar): HResult;
      stdcall;
    function GetInfoFlags(out pdwFlags: DWORD): HResult; stdcall;
  end;
```

The `GetInfoTip()` method is called by the shell to retrieve the InfoTip for a given file. The `dwFlags` parameter is currently unused. The InfoTip string is returned in the `ppwszTip` parameter.

NOTE

The `ppwszTip` parameter points to a wide character string. Memory for this string must be allocated within the InfoTip handler using the shell's memory allocator. The shell is responsible for freeing this memory.

Implementation

Like the other shell extensions, the InfoTip handler is implemented as a simple COM server DLL. The COM object contained within implements the `IQueryInfo` and `IPersistFile` methods. Listing 16.12 shows the contents of `InfoMain.pas`, the main unit for the `DDGInfoTip` project, which contains the Delphi implementation of an InfoTip handler.

LISTING 16.12 `InfoMain.pas`—Main Unit for InfoTip Handler

```
unit InfoMain;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, ShlObj;

type
  TInfoTipHandler = class(TComObject, IQueryInfo, IPersistFile)
  private
    FFileName: string;
    FMalloc: IMalloc;
  protected
    { IQUeryInfo }
    function GetInfoTip(dwFlags: DWORD; var ppwszTip: PWideChar): HResult;
      stdcall;
    function GetInfoFlags(out pdwFlags: DWORD): HResult; stdcall;
    {IPersist}
    function GetClassID(out classID: TCLSID): HResult; stdcall;
```

LISTING 16.12 Continued

```
    { IPersistFile }
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  public
    procedure Initialize; override;
  end;

  TInfoTipFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

const
  Class_InfoTipHandler: TGUID = '{5E08F28D-A5B1-4996-BDF1-5D32108DB5E5}';

implementation

uses ComServ, SysUtils, Registry;

const
  TipBufLen = 1024;

procedure PackageInfoCallback(const Name: string; NameType: TNameType;
  Flags: Byte; Param: Pointer);
var
  S: string;
begin
  // if we are being passed the name of a contained unit, then
  // concatenate it to the list of units, which is passed in Param.
  if NameType = ntContainsUnit then
  begin
    S := Name;
    if PChar(Param)^ <> #0 then
      S := ', ' + S;
    StrLCat(PChar(Param), PChar(S), TipBufLen);
  end;
end;
```

LISTING 16.12 Continued

```
function TInfoTipHandler.GetClassID(out classID: TCLSID): HResult;
begin
  classID := Class_InfoTipHandler;
  Result := S_OK;
end;

function TInfoTipHandler.GetCurFile(out pszFileName: POleStr): HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.GetInfoFlags(out pdwFlags: DWORD): HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.GetInfoTip(dwFlags: DWORD;
  var ppwszTip: PWideChar): HResult;
var
  PackMod: HModule;
  TipStr: PChar;
  Size, Flags, TipStrLen: Integer;
begin
  Result := S_OK;
  if (CompareText(ExtractFileExt(FFileName), '.bpl') = 0) and
    Assigned(FMalloc) then
  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(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
    if PackMod <> 0 then
      try
        TipStr := StrAlloc(TipBufLen);
        try
          FillChar(TipStr^, TipBufLen, 0);  // zero out string memory
          // Fill up TipStr with contained units
          GetPackageInfo(PackMod, TipStr, Flags, PackageInfoCallback);
          TipStrLen := StrLen(TipStr);
          Size := (TipStrLen + 1) * SizeOf(WideChar);
          ppwszTip := FMalloc.Alloc(Size); // use shell's allocator
          // copy PAnsiChar to PWideChar
          MultiByteToWideChar(0, 0, TipStr, TipStrLen, ppwszTip, Size);
        finally
```

LISTING 16.12 Continued

```
          StrDispose(TipStr);
        end;
      finally
        FreeLibrary(PackMod);
      end;
  end;
end;

procedure TInfoTipHandler.Initialize;
begin
  inherited;
  // shells shell's memory allocator and save it away
  SHGetMalloc(FMalloc);
end;

function TInfoTipHandler.IsDirty: HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.Load(pszFileName: POleStr;
  dwMode: Integer): HResult;
begin
  // This is the only important IPersistFile method -- we need to save
  // away the file name
  FFileName := pszFileName;
  Result := S_OK;
end;

function TInfoTipHandler.Save(pszFileName: POleStr;
  fRemember: BOOL): HResult;
begin
  Result := E_NOTIMPL;
end;

function TInfoTipHandler.SaveCompleted(pszFileName: POleStr): HResult;
begin
  Result := E_NOTIMPL;
end;

{ TInfoTipFactory }

function TInfoTipFactory.GetProgID: string;
begin
  // ProgID not required for IntoTip handler shell extension
```

LISTING 16.12 Continued

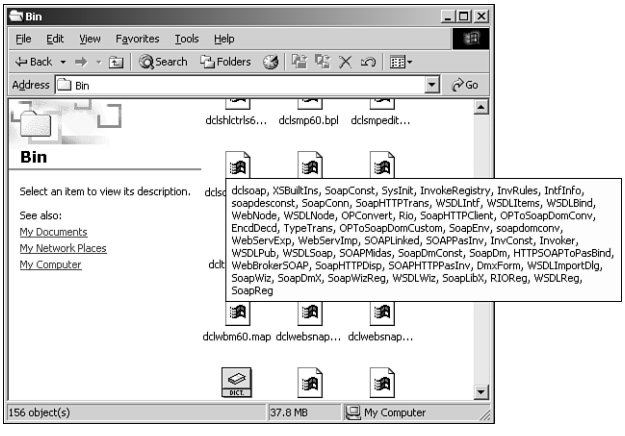
```
  Result := '';
end;

procedure TInfoTipFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // register this DLL as the InfoTip handler for .bpl files
    CreateRegKey('.bpl\shellex\{00021500-0000-0000-C000-000000000046}',
      '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl\shellex\{00021500-0000-0000-C000-000000000046}');
  end;
end;

procedure TInfoTipFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// This registry entry is required in order for the extension to
// operate correctly under Windows NT.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\' +
    'Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

initialization
  TInfoTipFactory.Create(ComServer, TInfoTipHandler, Class_InfoTipHandler,
    'InfoTipHandler', 'DDG sample InfoTip handler', ciMultiInstance,
    tmApartment);
end.
```

There are a couple of interesting points in this implementation. Notice that the shell's memory allocator is retrieved and stored in the `Initialize()` method. The allocator is later used to allocate memory for the InfoTip string in the `GetInfoTip()` method. The name of the file in question is passed to the handler in the `Load()` method. The work is done in the `GetInfoTip()` method, which gets package information using the `GetPackageInfo()` function that you learned about earlier in this chapter. As the `PackageInfoCallback()` callback function is called repeatedly from within `GetPackageInfo()`, the IntoTip string is concatenated together file-by-file.

Registration

The technique used for registration of the COM server DLL is almost identical to that of the other shell extensions in this chapter, as you can see in Listing 16.12. The key difference is the key under which InfoTip handlers are registered; these are always registered under `HKEY_CLASSES_ROOT\<file extension>\shellex\{00021500-0000-0000-C000-000000000046}`, where *`<file extension>`* is the file extension name, including the preceding dot.

Figure 16.12 shows this InfoTip handler in action.



The Delphi InfoTip handler shell extension.

Summary

This chapter covers all the different aspects of extending the Windows shell: tray-notification icons, AppBars, shell links, and a variety of shell extensions. It builds upon some of the knowledge you obtained in the last chapter when working with COM. In Chapter 17, you'll learn more about component-based development using interfaces.