Adventures in Messaging

CHAPTER

3

IN THIS CHAPTER

Although Visual Component Library (VCL) components expose many Win32 messages via Object Pascal events, it's still essential that you, the Win32 programmer, understand how the Windows message system works.

As a Delphi applications programmer, you'll find that the events surfaced by VCL will suit most of your needs; only occasionally will you have to delve into the world of Win32 message handling. As a Delphi component developer, however, you and messages will become very good friends because you have to directly handle many Windows messages and then invoke events corresponding to those messages.

NOTE

The messaging capabilities covered in this chapter are specific to the VCL and aren't supported under the CLX environment. For more on the CLX architectures, see Chapters 10, "Component Architecture: VCL and CLX," and 13, "CLX Component Development."

What Is a Message?

A *message* is a notification of some occurrence sent by Windows to an application. Clicking a mouse button, resizing a window, or pressing a key on the keyboard, for example, causes Windows to send a message to an application notifying it of what occurred.

A message manifests itself as a *record* passed to an application by Windows. That record contains information such as what type of event occurred and additional information specific to the message. The message record for a mouse button click message, for example, contains the mouse coordinates at the time the button was pressed. The record type passed from Windows to the application is called a `TMsg`, which is defined in the `Windows` unit as shown in the following code:

```
type
  TMsg = packed record
    hwnd: HWND;     // the handle of the Window for which the message
                    // is intended
    message: UINT;  // the message constant identifier
    wParam: WPARAM; // 32 bits of additional message-specific information
    lParam: LPARAM; // 32 bits of additional message-specific information
    time: DWORD;    // the time that the message was created
    pt: TPoint;     // Mouse cursor position when the message was created
  end;
```

What's in a Message?

Does the information in a message record look like Greek to you? If so, here's a little insight into what's what:

`hwnd`	The 32-bit window handle of the window for which the message is intended. The window can be almost any type of screen object because Win32 maintains window handles for most visual objects (windows, dialog boxes, buttons, edits, and so on).
`message`	A constant value that represents some message. These constants can be defined by Windows in the `Windows` unit or by you through user-defined messages.
`wParam`	This field often contains a constant value associated with the message; it can also contain a window handle or the identification number of some window or control associated with the message.
`lParam`	This field often holds an index or pointer to some data in memory. Because `wParam`, `lParam`, and `Pointer` are all 32 bits in size, you can typecast interchangeably between them.

Now that you have an idea what makes up a message, it's time to take a look at some different types of Windows messages.

Types of Messages

The Win32 API predefines a constant for each Windows message. These constants are the values kept in the message field of the `TMsg` record. All these constants are defined in Delphi's `Messages` unit; most are also described in the online help. Notice that each of these constants begins with the letters *WM*, which stand for *Windows Message*. Table 3.1 lists some of the common Windows messages, along with their meanings and values.

TABLE 3.1 Common Windows Messages

Message Identifier	*Value*	*Tells a Window That. . .*
`wm_Activate`	`$0016`	It's being activated or deactivated.
`wm_Char`	`$0102`	`wm_KeyDown` and `wm_KeyUp` messages have been sent for one key.
`wm_Close`	`$0010`	It should terminate.
`wm_KeyDown`	`$0100`	A keyboard key is being pressed.
`wm_KeyUp`	`$0101`	A keyboard key has been released.
`wm_LButtonDown`	`$0201`	The user is pressing the left mouse button.
`wm_MouseMove`	`$0200`	The mouse is being moved.
`WM_PAINT`	`$000F`	It must repaint its client area.
`wm_Timer`	`$0113`	A timer event has occurred.
`wm_Quit`	`$0012`	A request has been made to shut down the program.

How the Windows Message System Works

A Windows application's message system has three key components:

- Message queue—Windows maintains a message queue for each application. A Windows application must get messages from this queue and dispatch them to the proper windows.
- Message loop—This is the loop mechanism in a Windows program that fetches a message from the application queue and dispatches it to the appropriate window, fetches the next message, dispatches it to the appropriate window, and so on.
- Window procedure—Each window in your application has a window procedure that receives each of the messages passed to it by the message loop. The window procedure's job is to take each window message and respond to it accordingly. A window procedure is a callback function; a window procedure usually returns a value to Windows after processing a message.

NOTE

A *callback function* is a function in your program that's called by Windows or some other external module.

Getting a message from point A (some event occurs, creating a message) to point B (a window in your application responds to the message) is a five-step process:

1. Some event occurs in the system.
2. Windows translates this event into a message and places it into the message queue for your application.
3. Your application retrieves the message from the queue and places it in a `TMsg` record.
4. Your application passes on the message to the window procedure of the appropriate window in your application.
5. The window procedure performs some action in response to the message.

Steps 3 and 4 make up the application's *message loop*. The message loop is often considered the heart of a Windows program because it's the facility that enables your program to respond to external events. The message loop spends its whole life fetching messages from the application queue and passing them to the appropriate windows in your application. If there are no messages in your application's queue, Windows allows other applications to process their messages. Figure 3.1 shows these steps.

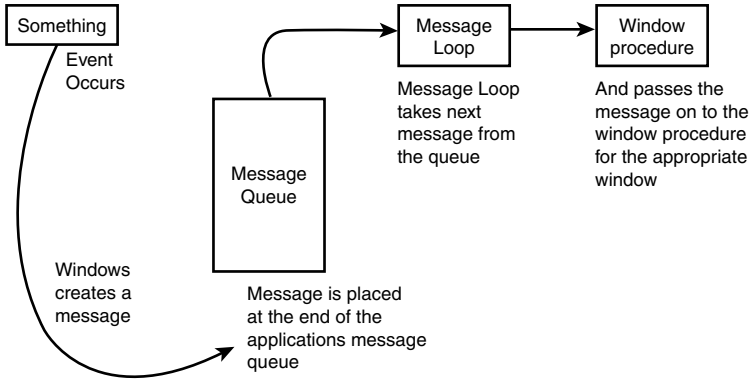

FIGURE 3.1
The Windows Message system.

Delphi's Message System

VCL handles many of the details of the Windows message system for you. The message loop is built into VCL's `Forms` unit, for example, so you don't have to worry about fetching

messages from the queue or dispatching them to a window procedure. Delphi also places the information located in the Windows `TMsg` record into a generic `TMessage` record:

```
type
  TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam: Longint;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

Notice that `TMessage` record has a little less information than does `TMsg`. That's because Delphi internalizes the other `TMsg` fields; `TMessage` contains just the essential information you need to handle a message.

It's important to note that the `TMessage` record also contains a `Result` field. As mentioned earlier, some messages require the window procedure to return some value after processing a message. With Delphi, you accomplish this process in a straightforward fashion by placing the return value in the `Result` field of `TMessage`. This process is explained in detail later in the section "Assigning Message Result Values."

Message-Specific Records

In addition to the generic `TMessage` record, Delphi defines a message-specific record for every Windows message. The purpose of these message-specific records is to give you all the information the message offers without having to decipher the `wParam` and `lParam` fields of a record. All the message-specific records can be found in the `Messages` unit. As an example, here's the message record used to hold most mouse messages:

```
type
  TWMMouse = packed record
    Msg: Cardinal;
    Keys: Longint;
    case Integer of
      0: (
        XPos: Smallint;
```

```
      YPos: Smallint);
    1: (
      Pos: TSmallPoint;
      Result: Longint);
end;
```

All the record types for specific mouse messages (`WM_LBUTTONDOWN` and `WM_RBUTTONUP`, for example) are simply defined as equal to `TWMMouse`, as in the following example:

```
TWMRButtonUp = TWMMouse;
TWMLButtonDown = TWMMouse;
```

NOTE

A message record is defined for nearly every standard Windows message. The naming convention dictates that the name of the record must be the same as the name of the message with a *T* prepended, using camel capitalization and without the underscore. For example, the name of the message record type for a `WM_SETFONT` message is `TWMSetFont`.

By the way, `TMessage` works with all messages in all situations but isn't as convenient as message-specific records.

Handling Messages

Handling or *processing* a message means that your application responds in some manner to a Windows message. In a standard Windows application, message handling is performed in each window procedure. By internalizing the window procedure, however, Delphi makes it much easier to handle individual messages; instead of having one procedure that handles all messages, each message has its own procedure. Three requirements must be met for a procedure to be a message-handling procedure:

- The procedure must be a method of an object.
- The procedure must take one `var` parameter of a `TMessage` or other message-specific record type.
- The procedure must use the `message` directive followed by the constant value of the message you want to process.

Here's an example of a procedure that handles `WM_PAINT` messages:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

NOTE

When naming message-handling procedures, the convention is to give them the same name as the message itself, using camel capitalization and without the underscore.

As another example, let's write a simple message-handling procedure for `WM_PAINT` that processes the message simply by beeping.

Start by creating a new, blank project. Then access the Code Editor window for this project and add the header for the `WMPaint` function to the `private` section of the `TForm1` object:

```
procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
```

Now add the function definition to the `implementation` part of this unit. Remember to use the dot operator to scope this procedure as a method of `TForm1`. Don't use the `message` directive as part of the function implementation:

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  Beep;
  inherited;
end;
```

Notice the use of the `inherited` keyword here. Call `inherited` when you want to pass the message to the ancestor object's handler. By calling `inherited` in this example, you pass on the message to `TForm`'s `WM_PAINT` handler.

NOTE

Unlike normal calls to inherited methods, here you don't give the name of the inherited method because the name of the method is unimportant when it's dispatched. Delphi knows what method to call based on the message value used with the `message` directive in the class interface.

The main unit in Listing 3.1 provides a simple example of a form that processes the `WM_PAINT` message. Creating this project is easy: Just create a new project and add the code for the `WMPaint` procedure to the `TForm` object.

LISTING 3.1 GetMess—A Message-Handling Example

```
unit GMMain;

interface

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

type
  TForm1 = class(TForm)
  private
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
  inherited;
end;

end.
```


Whenever a `WM_PAINT` message comes down the pike, it's passed to the `WMPaint` procedure. The `WMPaint` procedure simply informs you of the `WM_PAINT` message by making some noise with the `MessageBeep()` procedure and then passes the message to the inherited handler.

`MessageBeep()`: The Poor Man's Debugger

While we're on the topic of beeping, now is a good time for a slight digression. The `MessageBeep()` procedure is one of the most straightforward and useful elements in the Win32 API. Its use is simple: Call `MessageBeep()`, pass a predefined constant, and Windows beeps the PC's speaker. (If you have a sound card, it plays a WAV file.) Big

continues

deal, you say? On the surface it might not seem like much, but `MessageBeep()` really shines as an aid in debugging your programs.

If you're looking for a quick-and-dirty way to tell whether your program is reaching a certain place in your code—without having to bother with the debugger and breakpoints—`MessageBeep()` is for you. Because it doesn't require a handle or some other Windows resource, you can use it practically anywhere in your code, and as a wise man once said, "`MessageBeep()` is for the itch you can't scratch with the debugger." If you have a sound card, you can pass `MessageBeep()` one of several predefined constants to have it play a wider variety of sounds—these constants are defined under `MessageBeep()` in the Win32 API help file.

If you're like the authors and are too lazy to type out that whole big, long function name and parameter, you can use the `Beep()` procedure found in the `SysUtils` unit. The implementation of `Beep()` is simply a call to `MessageBeep()` with the parameter `0`.

Message Handling: Not Contract Free

Unlike responding to Delphi events, handling Windows messages is not "contract free." Often, when you decide to handle a message yourself, Windows expects you to perform some action when processing the message. Most of the time, VCL has much of this basic message processing built in—all you have to do is call `inherited` to get to it. Think of it this way: You write a message handler so that your application will do the things you expect, and you call `inherited` so that your application will do the additional things Windows expects.

NOTE

The contractual nature of message handling can be more than just calling the inherited handler. In message handlers, you're sometimes restricted in what you can do. For example, in a `WM_KILLFOCUS` message, you cannot set focus to another control without causing a crash.

To demonstrate the `inherited` elements, consider the program in Listing 3.1 without calling `inherited` in the `WMPaint()` method. the procedure would look like this:

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
begin
  MessageBeep(0);
end;
```

This procedure never gives Windows a chance to perform basic handling of the `WM_PAINT` message, and the form will never paint itself. In fact, you might end up with several WM_PAINT

messages stacking up in the message queue, causing the beep to continue until the queue is cleared.

Sometimes there are circumstances in which you don't want to call the inherited message handler. An example is handling the `WM_SYSCOMMAND` messages to prevent a window from being minimized or maximized.

Assigning Message Result Values

When you handle some Windows messages, Windows expects you to return a result value. The classic example is the `WM_CTLCOLOR` message. When you handle this message, Windows expects you to return a handle to a brush with which you want Windows to paint a dialog box or control. (Delphi provides a `Color` property for components that does this for you, so the example is just for illustration purposes.) You can return this brush handle easily with a message-handling procedure by assigning a value to the `Result` field of `TMessage` (or another message record) after calling `inherited`. For example, if you were handling `WM_CTLCOLOR`, you could return a brush handle value to Windows with the following code:

```
procedure TForm1.WMCtlColor(var Msg: TWMCtlColor);
var
  BrushHand: hBrush;
begin
  inherited;
  { Create a brush handle and place into BrushHand variable }
  Msg.Result := BrushHand;
end;
```


The `TApplication` Type's `OnMessage` Event

Another technique for handling messages is to use `TApplication`'s `OnMessage` event. When you assign a procedure to `OnMessage`, that procedure is called whenever a message is pulled from the queue and about to be processed. This event handler is called before Windows itself has a chance to process the message. The `Application.OnMessage` event handler is of `TMessageEvent` type and must be defined with a parameter list, as shown here:

```
procedure SomeObject.AppMessageHandler(var Msg: TMsg;
  var Handled: Boolean);
```

All the message parameters are passed to the `OnMessage` event handler in the `Msg` parameter. (Note that this parameter is of the Windows `TMsg` record type described earlier in this chapter.) The `Handled` field requires you to assign a Boolean value indicating whether you have handled the message.

You can create an `OnMessage` event handler by using a `TApplicationEvents` component from the Additional page of the Component Palette. Here is an example of such an event handler:

```
var
  NumMessages: Integer;

procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
  var Handled: Boolean);
begin
  Inc(NumMessages);
  Handled := False;
end;
```

One limitation of `OnMessage` is that it's executed only for messages pulled out of the queue and not for messages sent directly to the window procedures of windows in your application. Chapter 13, "Hard-Core Techniques," of *Delphi 5 Developers Guide,* which is on this book's CD-ROM, shows techniques for working around this limitation by hooking into the application window procedure.

TIP

`OnMessage` sees all messages posted to all window handles in your application. This is the busiest event in your application (thousands of messages per second), so don't do anything in an `OnMessage` handler that takes a lot of time because you'll slow your whole application to a crawl. Clearly, this is one place where a breakpoint would be a very bad idea.

Sending Your Own Messages

Just as Windows sends messages to your application's windows, you will occasionally need to send messages between windows and controls within your application. Delphi provides several ways to send messages within your application, such as the `Perform()` method (which works independently of the Windows API) and the `SendMessage()` and `PostMessage()` API functions.

The `Perform()` Method

VCL provides the `Perform()` method for all `TControl` descendants; `Perform()` enables you to send a message to any form or control object given an instance of that object. The `Perform()` method takes three parameters—a message and its corresponding `lParam` and `wParam`—and is defined as follows:

```
function TControl.Perform(Msg: Cardinal; WParam, LParam: Longint):
  Longint;
```

To send a message to a form or control, use the following syntax:

```
RetVal := ControlName.Perform(MessageID, wParam, lParam);
```

`Perform()` is synchronous in that it doesn't return until the message has been handled. The `Perform()` method packages its parameters into a `TMessage` record and then calls the object's `Dispatch()` method to send the message—bypassing the Windows API messaging system. The `Dispatch()` method is described later in this chapter.

The `SendMessage()` and `PostMessage()` API Functions

Sometimes you need to send a message to a window for which you don't have a Delphi object instance. For example, you might want to send a message to a non-Delphi window, but you have only a handle to that window. Fortunately, the Windows API offers two functions that fit this bill: `SendMessage()` and `PostMessage()`. These two functions are essentially identical, except for one key difference: `SendMessage()`, similar to `Perform()`, sends a message directly to the window procedure of the intended window and waits until the message is processed before returning; `PostMessage()` posts a message to the Windows message queue and returns immediately.

`SendMessage()` and `PostMessage()` are declared as follows:

```
function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
function PostMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM;
  lParam: LPARAM): BOOL; stdcall;
```

- `hWnd` is the window handle for which the message is intended.
- `Msg` is the message identifier.
- `wParam` is 32 bits of additional message-specific information.
- `lParam` is 32 bits of additional message-specific information.

NOTE

Although `SendMessage()` and `PostMessage()` are used similarly, their respective return values are different. `SendMessage()` returns the result value of the message being processed, but `PostMessage()` returns only a `BOOL` that indicates whether the message was placed in the target window's queue. Another way to think of this is that `SendMessage()` is a synchronous operation, whereas `PostMessage()` is asynchronous.

Nonstandard Messages

Until now, the discussion has centered on regular Windows messages (those that begin with `WM_XXX`). However, two other major categories of messages merit some discussion: notification messages and user-defined messages.

Notification Messages

Notification messages are messages sent to a parent window when something happens in one of its child controls that might require the parent's attention. Notification messages occur only with the standard Windows controls (button, list box, combo box, and edit control) and with the Windows Common Controls (tree view, list view, and so on). For example, clicking or double-clicking a control, selecting some text in a control, and moving the scrollbar in a control all generate notification messages.

You can handle notification messages by writing message-handling procedures in the form that contains a particular control. Table 3.2 lists the Win32 notification messages for standard Windows controls.

TABLE 3.2 Standard Control Notification Messages

Notification	*Meaning*
Button Notification	
`BN_CLICKED`	The user clicked a button.
`BN_DISABLE`	A button is disabled.
`BN_DOUBLECLICKED`	The user double-clicked a button.
`BN_HILITE`	The user highlighted a button.
`BN_PAINT`	The button should be painted.
`BN_UNHILITE`	The highlight should be removed.
Combo Box Notification	
`CBN_CLOSEUP`	The list box of a combo box has closed.
`CBN_DBLCLK`	The user double-clicked a string.
`CBN_DROPDOWN`	The list box of a combo box is dropping down.
`CBN_EDITCHANGE`	The user has changed text in the edit control.
`CBN_EDITUPDATE`	Altered text is about to be displayed.
`CBN_ERRSPACE`	The combo box is out of memory.
`CBN_KILLFOCUS`	The combo box is losing the input focus.
`CBN_SELCHANGE`	A new combo box list item is selected.

TABLE 3.2 Continued

Notification	*Meaning*
CBN_SELENDCANCEL	The user's selection should be canceled.
CBN_SELENDOK	The user's selection is valid.
CBN_SETFOCUS	The combo box is receiving the input focus.
Edit Notification	
EN_CHANGE	The display is updated after text changes.
EN_ERRSPACE	The edit control is out of memory.
EN_HSCROLL	The user clicked the horizontal scrollbar.
EN_KILLFOCUS	The edit control is losing the input focus.
EN_MAXTEXT	The insertion is truncated.
EN_SETFOCUS	The edit control is receiving the input focus.
EN_UPDATE	The edit control is about to display altered text.
EN_VSCROLL	The user clicked the vertical scrollbar.
List Box Notification	
LBN_DBLCLK	The user double-clicked a string.
LBN_ERRSPACE	The list box is out of memory.
LBN_KILLFOCUS	The list box is losing the input focus.
LBN_SELCANCEL	The selection is canceled.
LBN_SELCHANGE	The selection is about to change.
LBN_SETFOCUS	The list box is receiving the input focus.

Internal VCL Messages

VCL has an extensive collection of its own internal and notification messages. Although you don't commonly use these messages in your Delphi applications, Delphi component writers will find them useful. These messages begin with CM_ (for *component message*) or CN_ (for *component notification*), and they are used to manage VCL internals such as focus, color, visibility, window re-creation, dragging, and so on. You can find a complete list of these messages in the "Creating Custom Components" portion of the Delphi online help.

A common inquiry is how to detect that the mouse is entered or left a controls space. This can be handled by processing the custom messages CM_MOUSEENTER and CM_MOUSELEAVE. Consider the following component:

```
TSpecialPanel = class(TPanel)
protected
```

```
    procedure CMMouseEnter(var Msg: TMessage); message CM_MOUSEENTER;
    procedure CMMouseLeave(var Msg: TMessage); message CM_MOUSELEAVE;
  end;
…
procedure TSpecialPanel.CMMouseEnter(var Msg: TMessage);
begin
  inherited;
  Color := clWhite;
end;

procedure TSpecialPanel.CMMouseLeave(var Msg: TMessage);
begin
  inherited;
  Color := clBtnFace;
end;
```

This component handles the custom messages by turning the panel white when the mouse has entered the component's surface area and then turns the color back to `clBtnFace` when the mouse leaves. You'll find an example of this code on the CD under the directory `CustMessage`.

User-Defined Messages

At some point, you'll come across a situation in which one of your own applications must send a message to itself, or you have to send messages between two of your own applications. At this point, one question that might come to mind is, "Why would I send myself a message instead of simply calling a procedure?" It's a good question, and there are actually several answers. First, messages give you polymorphism without requiring knowledge of the recipient's type. Messages are therefore as powerful as virtual methods but more flexible. Also, messages allow for optional handling: If the recipient doesn't do anything with the message, no harm is done. Finally, messages allow for broadcast notifications to multiple recipients and "parasitic" eavesdropping, which isn't easily done with procedures alone.

Messages Within Your Application

Having an application send a message to itself is easy. Just use the `Perform()`, `SendMessage()`, or `PostMessage()` function and use a message value in the range of `WM_USER + 100` through `$7FFF` (the value Windows reserves for user-defined messages):

```
const
 SX_MYMESSAGE = WM_USER + 100;

begin
  SomeForm.Perform(SX_MYMESSAGE, 0, 0);
  { or }
  SendMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
```

```
  { or }
  PostMessage(SomeForm.Handle, SX_MYMESSAGE, 0, 0);
  .
  .
  .
end;
```

Then create a normal message-handling procedure for this message in the form in which you want to handle the message:

```
TForm1 = class(TForm)
  .
  .
  .
private
  procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
end;

procedure TForm1.SXMyMessage(var Msg: TMessage);
begin
  MessageDlg('She turned me into a newt!', mtInformation, [mbOk], 0);
end;
```

As you can see, there's little difference between using a user-defined message in your application and handling any standard Windows message. The real key here is to start at `WM_USER + 100` for interapplication messages and to give each message a name that has something to do with its purpose.

CAUTION

Never send messages with values of `WM_USER` through `$7FFF` unless you're sure that the intended recipient is equipped to handle the message. Because each window can define these values independently, the potential for bad things to happen is great unless you keep careful tabs on which recipients you send `WM_USER` through `$7FFF` messages to.

Messaging Between Applications

When you want to send messages between two or more applications, it's usually best to use the `RegisterWindowMessage()` API function in each application. This method ensures that every application uses the same message number for a given message.

`RegisterWindowMessage()` accepts a null-terminated string as a parameter and returns a new message constant in the range of `$C000` through `$FFFF`. This means that all you have to do is

call `RegisterWindowMessage()` with the same string in each application between which you want to send messages; Windows returns the same message value for each application. The true benefit of `RegisterWindowMessage()` is that because a message value for any given string is guaranteed to be unique throughout the system, you can safely broadcast such messages to all windows with fewer harmful side effects. It can be a bit more work to handle this kind of message, though; because the message identifier isn't known until runtime, you can't use a standard message handler procedure, and you must override a control's `WndProc()` or `DefaultHandler()` method or subclass an existing window procedure. A technique for handling registered messages is demonstrated in Chapter 13, "Hard-Core Techniques," of *Delphi 5 Developer's Guide*, found on this book's CD-ROM. This useful demo shows how to prevent multiple copies of your application from being launched.

NOTE

The number returned by `RegisterWindowMessage()` varies between Windows sessions and can't be determined until runtime.

Broadcasting Messages

`TWinControl` descendants can broadcast a message record to each of their owned controls—thanks to the `Broadcast()` method. This technique is useful when you need to send the same message to a group of components. For example, to send a user-defined message called `um_Foo` to all of `Panel1`'s owned controls, use the following code:

```
var
  M: TMessage;
begin
  with M do
  begin
    Message := UM_FOO;
    wParam := 0;
    lParam := 0;
    Result := 0;
  end;
  Panel1.Broadcast(M);
end;
```

Anatomy of a Message System: VCL

There's much more to VCL's message system than handling messages with the `message` directive. After a message is issued by Windows, it makes a couple of stops before reaching your message-handling procedure (and it might make a few more stops afterward). All along the way, you have the power to act on the message.

For posted messages, the first stop for a Windows message in VCL is the `Application.ProcessMessage()` method, which houses the VCL main message loop. The next stop for a message is the handler for the `Application.OnMessage` event. `OnMessage` is called as messages are fetched from the application queue in the `ProcessMessage()` method. Because sent messages aren't queued, `OnMessage` won't be called for sent messages.

For posted messages, the `DispatchMessage()` API is then called internally to dispatch the message to the `StdWndProc()` function. For sent messages, `StdWndProc()` will be called directly by Win32. `StdWndProc()` is an assembler function that accepts the message from Windows and routes it to the object for which the message is intended.

The object method that receives the message is called `MainWndProc()`. Beginning with `MainWndProc()`, you can perform any special handling of the message your program might require. Generally, you handle a message at this point only if you don't want a message to go through VCL's normal dispatching.

After leaving the `MainWndProc()` method, the message is routed to the object's `WndProc()` method and then on to the dispatch mechanism. The dispatch mechanism, found in the object's `Dispatch()` method, routes the message to any specific message-handling procedure that you've defined or that already exists within VCL.

Then the message finally reaches your message-specific handling procedure. After flowing through your handler and the inherited handlers you might have invoked using the `inherited` keyword, the message goes to the object's `DefaultHandler()` method. `DefaultHandler()` performs any final message processing and then passes the message to the Windows `DefWindowProc()` function or other default window procedure (such as `DefMDIProc`) for any Windows default processing. Figure 3.2 shows VCL's message-processing mechanism.

NOTE

You should always call `inherited` when handling messages unless you're absolutely certain you want to prevent normal message processing.

TIP

Because all unhandled messages flow to `DefaultHandler()`, that's usually the best place to handle interapplication messages in which the values were obtained by way of the `RegisterWindowMessage()` procedure.

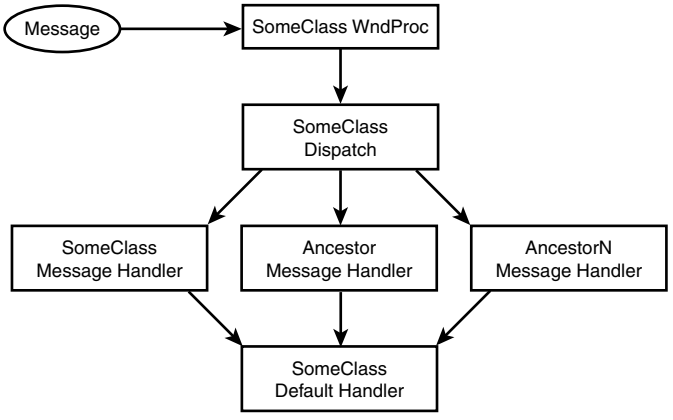

FIGURE 3.2
VCL's message system.

To better understand VCL's message system, create a small program that can handle a message at the `Application.OnMessage`, `WndProc()`, message procedure, or `DefaultHandler()` stage. This project is called `CatchIt`; its main form is shown in Figure 3.3.

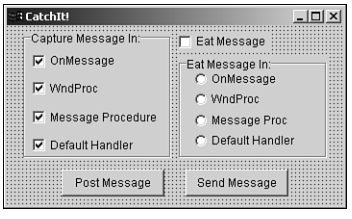

FIGURE 3.3
The main form of the `CatchIt` *message example.*

The `OnClick` event handlers for `PostMessButton` and `SendMessButton` are shown in the following code. The former uses `PostMessage()` to post a user-defined message to the form; the latter uses `SendMessage()` to send a user-defined message to the form. To differentiate between post and send, note that the value `1` is passed in the `wParam` of `PostMessage()` and that the value `0` (zero) is passed for `SendMessage()`. Here's the code:

```
procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ posts message to form }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;
```

```
procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ sends message to form }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0); // send message to form
end;
```

This application provides the user with the opportunity to "eat" the message in the `OnMessage` handler, `WndProc()` method, message-handling method, or `DefaultHandler()` method (that is, to not trigger the inherited behavior and to therefore stop the message from fully circulating through VCL's message-handling system). Listing 3.2 shows the completed source code for the main unit of this project, thus demonstrating the flow of messages in a Delphi application.

LISTING 3.2 The Source Code for `CIMain.PAS`

```
unit CIMain;

interface

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

const
  SX_MYMESSAGE = WM_USER;                // User-defined message value
  MessString = '%s message now in %s.';  // String to alert user

type
  TMainForm = class(TForm)
    GroupBox1: TGroupBox;
    PostMessButton: TButton;
    WndProcCB: TCheckBox;
    MessProcCB: TCheckBox;
    DefHandCB: TCheckBox;
    SendMessButton: TButton;
    AppMsgCB: TCheckBox;
    EatMsgCB: TCheckBox;
    EatMsgGB: TGroupBox;
    OnMsgRB: TRadioButton;
    WndProcRB: TRadioButton;
    MsgProcRB: TRadioButton;
    DefHandlerRB: TRadioButton;
    procedure PostMessButtonClick(Sender: TObject);
    procedure SendMessButtonClick(Sender: TObject);
    procedure EatMsgCBClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
```

LISTING 3.2 Continued

```
    procedure AppMsgCBClick(Sender: TObject);
  private
    { Handles messages at Application level }
    procedure OnAppMessage(var Msg: TMsg; var Handled: Boolean);
    { Handles messages at WndProc level }
    procedure WndProc(var Msg: TMessage); override;
    { Handles message after dispatch }
    procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
    { Default message handler }
    procedure DefaultHandler(var Msg); override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  // strings which will indicate whether a message is sent or posted
  SendPostStrings: array[0..1] of String = ('Sent', 'Posted');

procedure TMainForm.FormCreate(Sender: TObject);
{ OnCreate handler for main form }
begin
  // set OnMessage to my OnAppMessage method
  Application.OnMessage := OnAppMessage;
  // use the Tag property of checkboxes to store a reference to their
  // associated radio buttons
  AppMsgCB.Tag := Longint(OnMsgRB);
  WndProcCB.Tag := Longint(WndProcRB);
  MessProcCB.Tag := Longint(MsgProcRB);
  DefHandCB.Tag := Longint(DefHandlerRB);
  // use the Tag property of radio buttons to store a reference to their
  // associated checkbox
  OnMsgRB.Tag := Longint(AppMsgCB);
  WndProcRB.Tag := Longint(WndProcCB);
  MsgProcRB.Tag := Longint(MessProcCB);
  DefHandlerRB.Tag := Longint(DefHandCB);
end;

procedure TMainForm.OnAppMessage(var Msg: TMsg; var Handled: Boolean);
{ OnMessage handler for Application }
```

LISTING 3.2 Continued

```
begin
  // check to see if message is my user-defined message
  if Msg.Message = SX_MYMESSAGE then
  begin
    if AppMsgCB.Checked then
    begin
      // Let user know about the message.  Set Handled flag appropriately
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'Application.OnMessage']));
      Handled := OnMsgRB.Checked;
    end;
  end;
end;

procedure TMainForm.WndProc(var Msg: TMessage);
{ WndProc procedure of form }
var
  CallInherited: Boolean;
begin
  CallInherited := True;           // assume we will call the inherited
  if Msg.Msg = SX_MYMESSAGE then   // check for our user-defined message
  begin
    if WndProcCB.Checked then      // if WndProcCB checkbox is checked...
    begin
      // Let user know about the message.
      ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
        'WndProc']));
      // Call inherited only if we are not supposed to eat the message.
      CallInherited := not WndProcRB.Checked;
    end;
  end;
  if CallInherited then inherited WndProc(Msg);
end;

procedure TMainForm.SXMyMessage(var Msg: TMessage);
{ Message procedure for user-defined message }
var
  CallInherited: Boolean;
begin
  CallInherited := True;           // assume we will call the inherited
  if MessProcCB.Checked then       // if MessProcCB checkbox is checked
  begin
    // Let user know about the message.
    ShowMessage(Format(MessString, [SendPostStrings[Msg.WParam],
```

LISTING 3.2 Continued

```
      'Message Procedure']));
    // Call inherited only if we are not supposed to eat the message.
    CallInherited := not MsgProcRB.Checked;
  end;
  if CallInherited then Inherited;
end;

procedure TMainForm.DefaultHandler(var Msg);
{ Default message handler for form }
var
  CallInherited: Boolean;
begin
  CallInherited := True;           // assume we will call the inherited
  // check for our user-defined message
  if TMessage(Msg).Msg = SX_MYMESSAGE then   begin
    if DefHandCB.Checked then       // if DefHandCB checkbox is checked
    begin
      // Let user know about the message.
      ShowMessage(Format(MessString,
        [SendPostStrings[TMessage(Msg).WParam], 'DefaultHandler']));
      // Call inherited only if we are not supposed to eat the message.
      CallInherited := not DefHandlerRB.Checked;
    end;
  end;
  if CallInherited then inherited DefaultHandler(Msg);
end;

procedure TMainForm.PostMessButtonClick(Sender: TObject);
{ posts message to form }
begin
  PostMessage(Handle, SX_MYMESSAGE, 1, 0);
end;

procedure TMainForm.SendMessButtonClick(Sender: TObject);
{ sends message to form }
begin
  SendMessage(Handle, SX_MYMESSAGE, 0, 0); // send message to form
end;

procedure TMainForm.AppMsgCBClick(Sender: TObject);
{ enables/disables proper radio button for checkbox click }
begin
  if EatMsgCB.Checked then
  begin
    with TRadioButton((Sender as TCheckBox).Tag) do
```

LISTING 3.2 Continued

```
    begin
      Enabled := TCheckbox(Sender).Checked;
      if not Enabled then Checked := False;
    end;
  end;
end;

procedure TMainForm.EatMsgCBClick(Sender: TObject);
{ enables/disables radio buttons as appropriate }
var
  i: Integer;
  DoEnable, EatEnabled: Boolean;
begin
  // get enable/disable flag
  EatEnabled := EatMsgCB.Checked;
  // iterate over child controls of GroupBox in order to
  // enable/disable and check/uncheck radio buttons
  for i := 0 to EatMsgGB.ControlCount - 1 do
    with EatMsgGB.Controls[i] as TRadioButton do
    begin
      DoEnable := EatEnabled;
      if DoEnable then DoEnable := TCheckbox(Tag).Checked;
      if not DoEnable then Checked := False;
      Enabled := DoEnable;
    end;
end;

end.
```


CAUTION

Although it's fine to use just the `inherited` keyword to send the message to an inherited handler in message-handler procedures, this technique doesn't work with `WndProc()` or `DefaultHandler()`. With these procedures, you must also provide the name of the inherited procedure or function, as in this example:

```
inherited WndProc(Msg);
```

You might have noticed that the `DefaultHandler()` procedure is somewhat unusual in that it takes one *untyped* `var` parameter. That's because `DefaultHandler()` assumes that the first word in the parameter is the message number; it isn't concerned with the rest of the information being passed. Because of this, you typecast the parameter as a `TMessage` so that you can access the message parameters.

The Relationship Between Messages and Events

Now that you know all the ins and outs of messages, recall that this chapter began by stating that VCL encapsulates many Windows messages in its event system. Delphi's event system is designed to be an easy interface into Windows messages. Many VCL events have a direct correlation with `WM_XXX` Windows messages. Table 3.3 shows some common VCL events and the Windows message responsible for each event.

TABLE 3.3 VCL Events and Corresponding Windows Messages

VCL Event	*Windows Message*
`OnActivate`	`wm_Activate`
`OnClick`	`wm_XButtonDown`
`OnCreate`	`wm_Create`
`OnDblClick`	`wm_XButtonDblClick`
`OnKeyDown`	`wm_KeyDown`
`OnKeyPress`	`wm_Char`
`OnKeyUp`	`wm_KeyUp`
`OnPaint`	`WM_PAINT`
`OnResize`	`wm_Size`
`OnTimer`	`wm_Timer`

Table 3.3 is a good rule-of-thumb reference when you're looking for events that correspond directly to messages.

TIP

Never write a message handler when you can use a predefined event to do the same thing. Because of the contract-free nature of events, you'll have fewer problems handling events than you will handling messages.

Summary

By now, you should have a pretty clear understanding of how the Win32 messaging system works and how VCL encapsulates that messaging system. Although Delphi's event system is great, knowing how messages work is essential for any serious Win32 programmer.