Getting The Message

by David Baer

Those using Delphi in its RAD capacity may remain blissfully unaware of the frenzy of messages flying around in any Windows program. But to attempt anything sophisticated, one will soon encounter the messaging aspects of the Windows environment. Few readers of this publication would be surprised that in this area, as in many others, Delphi and the VCL up the ante to provide a value-added set of capabilities that can significantly amplify the efficiency of our development efforts.

We'll explore some of those messaging augmentations in this article, focussing in particular on the VCL's `CN_...` and `CM_...` messages, which are an often overlooked, but extremely valuable, resource for the component developer. Following that discussion, I'll present a component that can be used to eavesdrop on Delphi internal message flows in a far more intimate way than an external spy tool like WinSight32.

A Better WAPI Than WAPI

Before we dive right in, I want to clarify the scope of what we'll cover. This article will not attempt an overview of the Delphi messaging technology. For that you should seek out a copy of *Delphi Component Design* by Danny Thorpe. Sadly, this book has gone out of print, but I still occasionally see a copy sitting on bookstore shelves. The book's chapter on messaging in Delphi is alone well worth the cover price. Although the book's basis is Delphi 2, nearly all the information in that chapter is currently accurate and pertinent. *Delphi Component Design* is an invaluable work that should be regarded as required reading for any serious Delphi developer, even one who has no component writing responsibilities.

That out of the way, let's begin. In Windows, a message consists of a 4-byte `Cardinal` message number, followed by two 4-byte data fields, the sub-allocations of which depend upon the nature of the message. In conventional Windows programs, messages are sent to message processing procedures associated with a window's handle. As many Delphi components are wrappers (of varying 'thickness') around stock Windows controls, this message flow must be preserved. But Delphi and the VCL perform several extremely clever manipulations which allow the Windows messaging to seamlessly integrate with Delphi's powerful object oriented environment.

For starters, a Windows procedure is just that: a *procedure*. But Delphi components are classes, and the procedures of classes are *methods*. The calling protocols of the two are different: methods need a pointer to the object instance passed in a call along with any parameters. Delphi manages to transform the former into the latter by dynamically creating a small piece of code which maps incoming procedure calls to method calls. The procedure to which these outside calls are mapped is the virtual method `WndProc`, which is declared in `TControl`. The great thing about our window procedure being a method is that we now have access to all the per-instance data of our object.

Another great thing about this is that Delphi lets us have it two ways. We can get messages delivered using the API call `SendMessage`, which is available to non-Delphi participants. Or, we can use the `TControl` method `Perform`, to polymorphically invoke `WndProc` directly. Of course, in using `Perform` and bypassing the API, Windows (and any active message monitoring programs) will have no knowledge of this communication. Apart from that, the following two lines of code accomplish the same thing:

```
SendMessage(AControl.Handle,
  A_MESSAGE_NBR, 0, 0);
```

```
AControl.Perform(
  A_MESSAGE_NBR,0,0);
```

One final detail before we move on. You may suspect that, somewhere in all of this, a stock Windows control inside a class wrapper needs to actually *receive* its messages. Don't worry, Delphi doesn't let us down. This happens in the `DefaultHandler` method of `TWinControl`, assuming processing upstream has not preempted the normal flow. The details are beyond the scope of this discussion, but, again, refer to *Delphi Component Design* and, of course, the VCL source, for further edification.

Look Ma, No Handles!

If you're paying *very* close attention, you may have noticed a seeming contradiction in the above explanation. On the one hand, we've got `WndProc` and `Perform` methods declared in `TControl`, but window handles for controls don't appear in the class hierarchy until `TWinControl`. What's going on here? Simply, the VCL is again demonstrating its largesse. It can be quite convenient at times for lowly graphic controls like labels to be plugged in to the messaging network. But in the normal world of Windows code, messaging requires handles, and objects with window handles consume valuable finite resources (even though this is far less of a concern now that we've gratefully arrived in the bounteous land of Win32).

But having `WndProc` first appear in the `TControl` class gives us this extra capability without the extra expense. Even without a window handle, we can use `Perform` to send messages to these handle-less controls. Naturally, lacking a window handle, the `SendMessage` API call is not an option.

But wait, there's more. It doesn't stop with `TControl`, because any Delphi object can be the recipient of a message courtesy of the `Dispatch` method of `TObject`. `Dispatch` is the method which undertakes the responsibility of finding methods associated with message numbers and invoking them (ie

message handlers are those methods declared with the message key word). If no handler exists, Dispatch just calls the TObject virtual method DefaultHandler.

Dispatch is less judgmental about the structure of the messages it fields. Unlike the other participants mentioned so far, Dispatch only requires that the first four bytes of the data be the message number. It assumes that the sender and recipient mutually understand the structure of the data involved. Dispatch is just the mail man.

In most cases, messages to Delphi controls will pass through both WndProc and Dispatch, but this is not always the case. A message may be 'eaten' in WndProc and never make it to the lower level Dispatch. Alternatively, a message sent via Dispatch will never be seen by WndProc, as it enters the system at this lower level. A grep of the VCL will show you that although Dispatch calls do appear here and there, its use is quite limited.

One place I've found Dispatch to be especially convenient is in situations where a component needs to send a notification to its Owner, which could be either a form or a data module (a TComponent derivative). As TComponent has no WndProc (recall, that doesn't appear until TControl), Dispatch allows easily coding the notification without concern as to what sort of Owner is in place.

Commonly Neglected

Next let's move on to the subject of the VCL's value-added messages. These come in two varieties: control messages (the CM_... group) and control notifications (the CN_... group). Listing 1 presents a partial list of their message number declarations (extracted from controls.pas).

Both groups of messages can be quite helpful in building component functionality, but regrettably, they are largely undocumented. Ray Lischner did devote a number of pages to the subject in his book *Secrets of Delphi 2*, which has unfortunately become nearly impossible to find. Although admirable in every other way, *Delphi Component Design* barely mentions these messages in passing. Danny suggests that a brief examination of the VCL will allow you to find out whatever you might need to know about any particular message (which, in *his* case, I'm certain is perfectly true).

```
CM_BASE                  = $B000;
CM_ACTIVATE              = CM_BASE + 0;
CM_DEACTIVATE            = CM_BASE + 1;
CM_GOTFOCUS              = CM_BASE + 2;
CM_LOSTFOCUS             = CM_BASE + 3;
CM_CANCELMODE            = CM_BASE + 4;
CM_DIALOGKEY             = CM_BASE + 5;
CM_DIALOGCHAR            = CM_BASE + 6;
CM_FOCUSCHANGED          = CM_BASE + 7;
CM_PARENTFONTCHANGED     = CM_BASE + 8;
CM_PARENTCOLORCHANGED    = CM_BASE + 9;
...
CN_BASE              = $BC00;
CN_CHARTOITEM        = CN_BASE + WM_CHARTOITEM;
CN_COMMAND           = CN_BASE + WM_COMMAND;
CN_COMPAREITEM       = CN_BASE + WM_COMPAREITEM;
CN_CTLCOLORBTN       = CN_BASE + WM_CTLCOLORBTN;
CN_CTLCOLORDLG       = CN_BASE + WM_CTLCOLORDLG;
CN_CTLCOLOREDIT      = CN_BASE + WM_CTLCOLOREDIT;
CN_CTLCOLORLISTBOX   = CN_BASE + WM_CTLCOLORLISTBOX;
CN_CTLCOLORMSGBOX    = CN_BASE + WM_CTLCOLORMSGBOX;
CN_CTLCOLORSCROLLBAR = CN_BASE + WM_CTLCOLORSCROLLBAR;
CN_CTLCOLORSTATIC    = CN_BASE + WM_CTLCOLORSTATIC;
```

➤ *Listing 1*

As for myself, I was well into my quest to become an accomplished component developer before I began to comprehend what these wonderful devices were and what they could do for me. Let's begin with the component notification messages.

The first big mystery one might encounter in trying to understand them is discovering where and when they originate. A grep of the VCL source reveals nothing. Search for CN_KEYDOWN, for example, and you'll find about a half dozen places where code is present to handle this message, but nowhere will you find a statement that does a Perform, Dispatch, SendMessage or anything else! However, the mystery is immediately solved if you look a bit more closely at the form of the declarations in Listing 1. Each of the CN_ messages is defined as standard Windows message number (WM_xxxx) to which is added the constant offset CN_BASE.

To find where the CN_ messages originate, grep for CN_BASE and you'll be immediately rewarded. Unfortunately, having that layer of mystery removed, your reward will be yet another layer of curious goings on. For example, quite a few of the CN_ messages are sent in response to a WM_ message by way of the function DoControlMsg in controls.pas. DoControlMsg is called from quite a few places, and the motivation behind all of this is not immediately apparent.

But don't give up just yet. The one thing that the component notification messages seem to have in common is that they offer a *previewing* capability. This previewing not only gives the potential recipient control a 'heads up', the control then also has the opportunity to alter the message data before the message continues along its way.

Let's consider one example. Say your edit component lives on a form which has a shortcut defined for some menu item, but the menu action is not one your component is always prepared to support. Furthermore, when a key combination is defined as a shortcut, your edit component will never see the WM_xxxx key events in the first place. It will, however, see the preview messages associated with key strokes (CN_KEYDOWN, CN_CHAR and CN_KEYUP). Forewarned is forearmed.

Completely Mysterious

Although it can occasionally be useful to tap into control notifications, it's the control messages (the CM_ series) that are the truly useful lot. Their use is widespread and their purposes are quite diverse. But there are two things for which we can be grateful: their names are self-explanatory for the most part, and we can find exactly

where and when they originate with a bit of grepping.

As examples, let's start with a truly helpful pair of them: CM_MOUSEENTER and CM_MOUSELEAVE. Suppose you had a label component containing underlined text that you wanted to have behave like a browser link. When the mouse travels over it, you'd like it to change color. With these two handy messages, the solution could hardly be any easier.

Next let's look at a somewhat more obscure situation by examining CM_FONTCHANGED. Consider what goes on to produce and send this message. There's two dozen or so instances of message handlers for this message throughout the VCL, but all the handlers are declared in classes having TControl as an ancestor, and TControl is where the internal helper TFont object is introduced.

TFont has an inherited event OnChange, which it triggers when there's a change to one of its own property settings. TControl hooks up its method FontChanged to be the event handler (it's allowed to do so since the event is strictly internal and not surfaced for use by the component user). In FontChanged, several state management chores are completed and a CM_FONTCHANGED is sent to Self. Clever, eh?

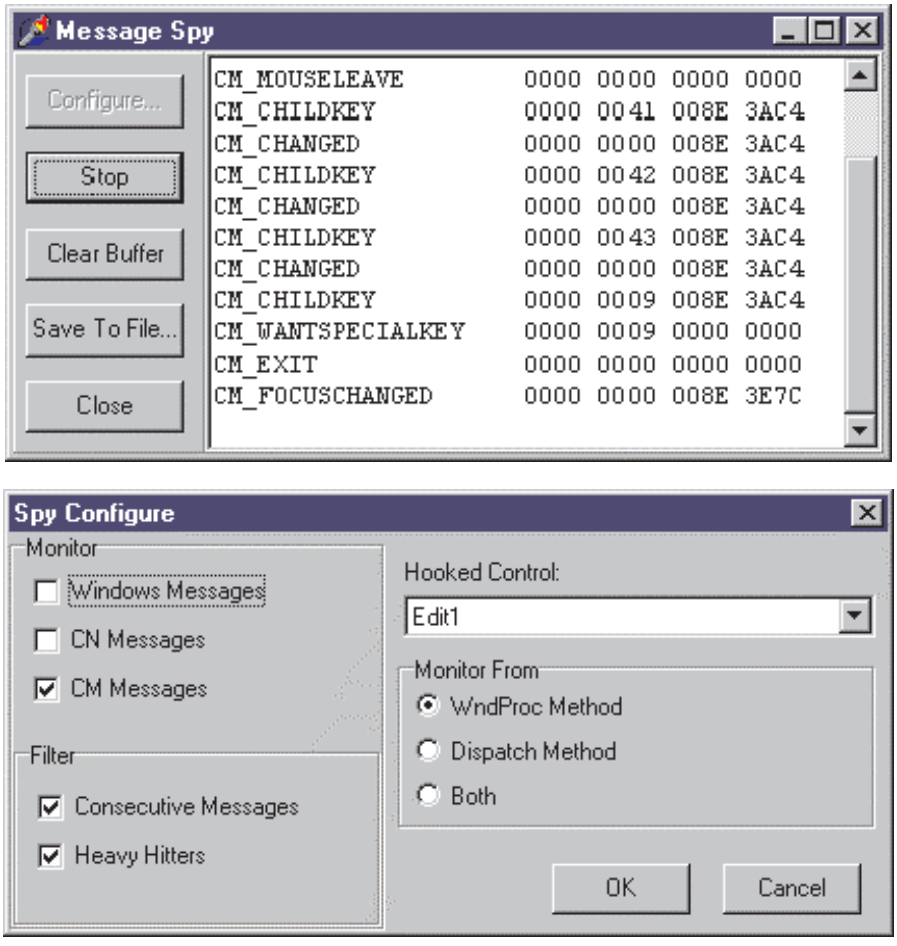
But let's play devil's advocate for a moment and consider whether this solution isn't a rather baroque one. After all, why not just declare a virtual FontChanged method (or a dynamic one if you're worried about VMT size) and be done with it? Certainly a direct method invocation would be more efficient than invoking the VCL message machinery.

There are several possible arguments in favor of a message approach, however. For one thing, in this situation at least, a message simply *feels* right. One might conveniently group multiple message constants in a case list inside a WndProc case statement for shared treatment, thus eliminating the need for multiple message handler methods. Also, messages are a very clean mechanism with respect to loose coupling. A message can always be sent without concern that the recipient cares anything about it. If it doesn't, the message is ignored and all we've done is to squander a couple hundred nanoseconds. But there's one additional way in which the message solution is concise and elegant.

In a number of CM message handlers, the message, once received, is immediately passed on to the parent, potentially travelling all the way up the parent chain. In other cases, the message is broadcast to child controls. The child control recipient may in turn broadcast it to its child controls. Once again, the loosely coupled nature of the message approach allows such code to be succinct and comprehensible.

By now, you may be convinced that Delphi's value-added messages are indeed something upon which you can and should capitalize in your component development efforts. So, what's the next step? You could, perhaps, hole up in a cave with the VCL source for several months and attempt to understand all their intimate details. But wouldn't it be easier if you could just check out the message flows in your component's proposed base class to see what's there to utilize? That's where TMessageSpy may prove useful.

➤ *Figure 1*



TSmiley's People

I briefly considered naming this component after John Le Carré's signature character, but then remembered that the name had already been taken. Oh well. In what follows, I'll describe the functions provided by TMessageSpy, and we'll take a look at the several techniques it uses to accomplish its covert operations. All the code is on the disk accompanying this month's issue, along with a readme file containing installation instructions and any last minute errata. One caution: TMessageSpy will not work with any Delphi release prior to Delphi 4.

Figure 1 shows the two forms supplied. The component is used

by placing it on the form containing the controls you'd like to monitor. It appears as a button which, when clicked, shows the message display form. From that form, you need to click the Configure button and select the control you're interested in. TMessageSpy will allow only one control at a time to be spied upon, and only one message spy may be active at one time. We'll see shortly how this limitation allows us to avoid a serious identity crisis.

In addition to selecting the control to monitor, the configuration form allows several options to be selected. The message types can be selected: Windows Messages, CN Messages and/or CM Messages. Actually, these designations denote the message number ranges $0000-$AFFF, $BC00-$FFFF and $B000-$BBFF respectively.

Two filtering mechanisms are provided. We can suppress the display for successive messages with the same number, and/or we can suppress four messages which will arrive with great frequency: WM_NCHITTEST, WM_SETCURSOR, WM_MOUSEMOVE and CM_HITTEST. Finally, we can choose to look at message arrivals into WndProc, Dispatch or both. When we select both, the message display prefixes each message line with a W or D to denote where the message was intercepted.

Finally, an event, OnMessageReceipt, is supplied that may be used to impose custom filtering and to add lines to the message display window. The parameter list contains a var Boolean item which specifies whether the message is to be filtered. Another parameter, Lines, provides a reference to the Lines property of the message window: it can be used to add your own information to the output display.

I won't undertake a soup to nuts explanation *[And I thought us Brits had some quaint phrases... Ed]* of the code associated with this component. The two forms are supported by fairly simple code. A third unit, MessageDict, is used to isolate the message numbers and associated names, and to provide lookup services. The interesting activities take place in the MessageSpy unit, which defines two classes. TMessageSpy is the component itself, and THooker is the actual spy engine.

THooker, the declaration for which is shown in Listing 2, is a singleton. As mentioned earlier, we will allow only one spy to be active at any time. THooker accomplishes its snooping by using one officially sanctioned hooking mechanism and several sneaky hacks. Given the nature of these activities, we'll try to be extra careful nothing unexpected happens by building in several safety features. The code for TMessageSpy was developed on my home Windows 95 system. Need I say more?

You might think that we'd need to resort to assembler code to accomplish these sort of activities, but that's not a very desirable solution. For one thing, we'd be discouraged by long-standing Borland tradition against documenting this most inscrutable of code. But, as you'll see, we can avoid assembler with a little creative use of the hacker's best friend: the pointer.

Hooking WndProc is fairly easy to do. Delphi 3 introduced a little-heralded enhancement: the WindowProc property of TControl. By assigning a method of THooker to this property, the hook is easily set in place, as can be seen in Listing 3. Method HookWndProc redirects the hooked control's WndProc to THooker's WndProcHook method. Then in WndProcHook, we pass the message through to the original WndProc, after we've given the active TMessageSpy the opportunity to examine and possibly display it.

Listing 3 code also illustrates the first safety feature we'll put in place. Hooks can be dicey where multiple hooks are present. If their settings and subsequent re-settings happen in an interleaved fashion, all manner of nastiness can result. Therefore, THooker will simply decline to offer its services if it discovers that the control already has a hook in place.

Briefly, this check works by finding the control's WndProc, and comparing this address with that in WindowProc. THooker is derived from TControl solely for this purpose. As a TControl, it has its own WndProc, which it can locate in its class's VMT. Since WndProc will be at the same VMT offset in all TControl-derived VMTs, THooker knows where to look in the hooked control class's VMT to perform its check. In approaching the problem

➤ *Listing 2*

```
PPointer = ^Pointer;
EHookerError = class(Exception);
TDispatchMethod = procedure(var Message) of Object;
TFreeInstanceMethod = procedure of Object;
TWndProcMethod = procedure(var Message: TMessage) of Object;
TDispatchMessageEvent = procedure(const Message) of Object;
TWndProcMessageEvent =
  procedure(const Message: TMessage) of Object;
THookeeDestructing = TNotifyEvent;
THooker = class(TControl)
  private
    Hookee:                 TControl;
    OnDispatchMessage:      TDispatchMessageEvent;
    OnWndProcMessage:       TWndProcMessageEvent;
    OnHookeeDestructing:    TNotifyEvent;
    ClientList:             TList;
    TrueDispatchMethod:     TDispatchMethod;
    TrueFreeInstanceMethod: TFreeInstanceMethod;
    TrueWndProcMethod:      TWndProcMethod;
    procedure DispatchHook(var Message);
    function DispatchVMTAddr(AControl: TControl): Pointer;
    procedure FreeInstanceHook;
    function FreeInstanceVMTAddr(AControl: TControl):
      Pointer;
    procedure HookDispatchMethod(AControl: TControl);
    procedure HookFreeInstanceMethod(AControl: TControl);
    procedure HookWndProcMethod(AControl: TControl);
    procedure SetHookee(AControl: TControl);
    procedure UnhookDispatchMethod;
    procedure UnhookFreeInstanceMethod;
    procedure UnhookWndProcMethod;
    procedure WndProcHook(var Message: TMessage);
    function WndProcIsHooked(AControl: TControl): Boolean;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    class procedure AttachSpy(MS: TMessageSpy);
    procedure DetachSpy(MS: TMessageSpy);
    procedure HookControl(MS: TMessageSpy; AControl:
      TControl;
    ParamOnWndProcMessage: TWndProcMessageEvent;
    ParamOnDispatchMessage: TDispatchMessageEvent;
    ParamOnHookeeDestructing: TNotifyEvent);
    procedure UnhookControl;
    procedure ViewerShowing(Showing: Boolean);
end;
```

```
procedure THooker.HookWndProcMethod(AControl: TControl);
begin
  if WndProcIsHooked(AControl) then
    raise EHookerError.Create('Cannot attach to control; ' +
      'the control currently has a WindowProc hook active');
  TrueWndProcMethod := AControl.WindowProc;
  AControl.WindowProc := WndProcHook;
end;
procedure THooker.WndProcHook(var Message: TMessage);
begin
  // fire the wndproc message event
  if Assigned(Hooker.OnWndProcMessage) then
    Hooker.OnWndProcMessage(Message);
  Hooker.TrueWndProcMethod(Message);
end;
function THooker.WndProcIsHooked(AControl: TControl):
  Boolean;
var
  P: PPointer;
  WPPosition: Integer;
  WPMethod: TWndProcMethod;
begin
  // get address of our class's WndProc method by assigning
  // it to the event variable; the first 4 bytes of this is
  // the address we need to find in our class's VMT
  WPMethod := WndProc;
  // get the address of our class's VMT
  P := Pointer(Pointer(Self)^);
  // interate through the VMT until we find the entry that
  // equals that of our WndProc
  while Pointer(TMethod(WPMethod).Code) <> Pointer(P^) do
    Inc(P);
  // the offset result is the address at which our WndProc
  // was found minus the start of our VMT
  WPPosition := (PChar(P) - PChar(Pointer(Self)^)) div 4;
  P := Pointer(Pointer(AControl)^);
  // add offset of WndProc to the pointer; the offset of
  // WndProc will be the same for all TControl derived
  // classes
  Inc(P, WPPosition);
  // finally, check to see if AControl's WindowProc property
  // does not equal that WndProc address in AControl's VMT;
  // if it does not then the control has a WndProc hook
  // active
  Result := (TMethod(AControl.WindowProc).Code <> P^);
end;
```

➤ *Listing 3*

in this way, we can be reasonably confident that the code will be compatible with future Delphi releases. The code that does this is in method `WndProcIsHooked`. It is documented in detail, and will hopefully not be too mysterious.

Can You See The Real Me?

At this point, let's consider what is going on when `WndProcHook` is called. The caller is invoking a method of the hooked control, and in doing so the compiler will generate code to have that object's instance pointer passed in as `Self`. So we may be inside a method of `THooker`, but at this point we have no way to access our `THooker`'s instance data, because `Self` ain't us! This is where the singleton implementation saves the day. As a singleton, we can declare a global instance reference in the unit and use that to access the instance data.

Setting a hook for `Dispatch` is a bit trickier than setting one for `WndProc`. In Delphi 4 we see that `Dispatch` has, for the first time, been declared as `virtual`. Why this was done is a mystery, as `Dispatch` is not overridden anywhere in the distributed VCL source code. But no matter, we'll use the `virtual` declaration to our advantage. Being virtual, `Dispatch` appears in the VMT of all classes. We can hack the VMT to introduce the hook. If this approach sounds familiar, you may have seen the technique explained by Cyril Jandia in his article in Issue 24 (August 1997).

```
procedure THooker.DispatchHook(var Message);
begin
  // if Self is the Hookee then fire the dispatch message event
  if (Hooker.Hookee = Self) and Assigned(Hooker.OnDispatchMessage) then
    Hooker.OnDispatchMessage(Message);
  // set the true dispatch method's object reference to the "self"
  // passed in to this method
  TMethod(Hooker.TrueDispatchMethod).Data := Self;
  Hooker.TrueDispatchMethod(Message);
end;
function THooker.DispatchVMTAddr(AControl: TControl): Pointer;
begin
  // get address of AControl's class's MVT
  Result := Pointer(Pointer(AControl)^);
  // subract offset of Dispatch to the pointer
  Inc(PChar(Result), vmtDispatch);
end;
procedure THooker.HookDispatchMethod(AControl: TControl);
var
  P:    Pointer;
  M:    TMethod;
  Cnt:  Cardinal;
begin
  // set P to the control's class's VMT address of Dispatch
  P := DispatchVMTAddr(AControl);
  // save it in TrueDispatchMethod
  TMethod(TrueDispatchMethod).Code := Pointer(P^);
  // set the VMT addr of the control's class's MVT Dispatch address
  // to that of our own Dispatch
  TDispatchMethod(M) := DispatchHook;
  WriteProcessMemory(GetCurrentProcess, P, @M.Code, SizeOf(Pointer), Cnt);
end;
```

➤ *Listing 4*

Unlike the `WndProc` hook explained earlier, which hooked a single instance of a control, introducing redirection into the VMT hooks all instances of a class. So our approach here must be a little different.

Listing 4 shows the code associated with this activity. The `HookDispatch` method sets the hook using the `DispatchVMTAddr` method to find the VMT location into which the redirection is placed. The code uses the constant offset `vmtDispatch`, which is defined in classes.pas, to ensure compatibility with future Delphi releases.

Once the hook is set, all `Dispatch` invocations on instances of the hooked control's class will come through `THooker`'s `DispatchHook` method. First of all, we need to check which control the message belongs to before allowing `TMessageSpy` to work with it. We only want to get `TMessageSpy` involved if the message belongs to the hooked control. We also need to play more identity games.

When the `Dispatch` hook was set, the true `Dispatch` method address was saved in the method variable `TrueDispatchMethod`, which is of type `TDispatchMethod`. `Method` variables contain two 32-bit words, the first containing the address of the procedure and the second containing the instance address. Although we saved the original code address in `TrueDispatchMethod`, we need to set the instance address each time we pass an invocation through. Once again, documentation in the code will

hopefully allow the interested reader to follow along.

One final thing to mention is the other safety measure provided in `THooker`. A hook into the hooked object's class's `FreeInstance` method is inserted. If the object is being destroyed, we need to remove ourselves from the picture while we still can. The `FreeInstance` hook technique is the same as used for `Dispatch`.

In From The Cold

Well, that's about it. I hope some of you will find `TMessageSpy` to be a useful research tool.

I also hope you'll forgive me for the hacks. After all, they *were* used for good, not evil, and I *did* try to provide some extra protections to compensate for them. Because, if I've learned one thing in my many years of programming, it's this: if one must engage in hacks, one should always try to practise safe hacks.

David Baer is Chief Software Architect at Spear Technologies in San Francisco. If you send an email to dbaer@speartechnologies.com he will attempt to answer it with... err... dispatch.