Beating The System: Taming The Windows Desktop, Part 2

by Dave Jewell

Back in the previous issue, we looked at how to programmatically save and restore the position of all the icons on your Windows desktop. As you'll recall, this turns out to be a lot trickier than it ought to be, thanks largely to the fact that the listview control (which actually implements the desktop at the API level) is running in a different process space to that of our controlling application. This means that any API-level listview message which uses pointers won't work, because what looks like a pointer to one process is just a random number from the perspective of another process!

We were able to get around this problem using DLL injection, the technique pioneered by Jeffrey Richter. Using a slightly massaged version of his DLL code, I wrote a small Delphi host application which invoked the DLL and caused it to be injected into the process space of the Windows Explorer. It was then possible to call Jeffrey's hidden dialog window from our application, using the `WM_APP` message to indicate whether we wanted the desktop layout to be saved and restored.

➤ *Figure 1: Want to have a Windows desktop that looks this cool? Well, probably not, but in next month's article I'll show how to programmatically tweak the desktop in a wide variety of ways.*

Son Of DIPSLIB

Although this is great as far as it goes, I felt that it would be nice to implement a much more flexible mechanism. Jeffrey's DLL saves or restores the entire desktop in one go, and doesn't allow you to (for example) change or query the position or name of a single desktop item. Moreover, the existing implementation always saves the desktop layout information into the system registry at a fixed location. Some folks might prefer to use a different registry location, or even save the layout to a file or Delphi stream in a format of their own choosing. It was clear that moving the actual desktop load/save code out of the DLL was a high priority. What has to stay in the DLL, of course, is the code which actually communicates with the listview control itself.

While I was at it, I spent a bit of time trying to clean up the existing DLL code. For example, in the existing implementation of `SetDIPSHook`, Jeff saves the thread ID of the calling application (our little Delphi app) into the global `dwApplication Thread`. This thread ID is used later, inside `GetMsgProc`, to send back a `WM_NULL` message to the application. You'll recall from last month's discussion that this serves as a sort of 'I am ready' message, indicating that the injection process is complete.

It occurred to me that it ought to be possible to eliminate this global variable. I modified the `SetDIPSHook` code so that, rather than storing the result of `GetCurrentThreadId` into a global, I simply passed it as the `wParam` field in the call to `PostThreadMessage`:

```
fOk = PostThreadMessage(
  dwDesktopThread, WM_NULL,
  GetCurrentThreadId(), 0);
```

This means that, after the DLL injection has taken place, the `GetMsgProc` routine is called for the first time, and we can send the 'I am ready' message to the Delphi application by retrieving the `wParam` field of the passed message. Within the `WH_GETMESSAGE` hook routine, the `lParam` field contains a pointer to the `MSG` structure, so we can do the deed as easily as this without making recourse to a global variable. I tried this, and it worked just fine:

```
PostThreadMessage(((
  LPMSG)lParam)->wParam,
  WM_NULL, 0, 0);
```

You might wonder why I'm making so much fuss over the elimination of a single global variable, but recall last month's discussion: the whole reason why we need to use a DLL written in (spit!) C++ is because Delphi doesn't allow us to create fancy things such as shared data segments. If only that shared data segment could be eliminated, then the way would be clear to rewrite the DLL in a 'proper' language. Hmmm...

At this point, I was on a roll. The only other variable in the shared data segment is `hHook`, the hook variable that's used to implement message chaining for the `SetWindowsHookEx` mechanism. Could this variable be eliminated too? If you examine last month's code, you'll see that `hHook` is referenced on the 'application side' (that is, in the context of the calling Delphi application) in only two places: once when the hook mechanism is installed and again when it's removed. On the 'Explorer side' (that is, in the context of the Explorer process) it's only ever used to chain via `CallNextHookEx`.

I realised that I could use exactly the same technique that I'd employed in eliminating the shared `dwApplicationThread` variable. First, I removed the assorted C++ pragmas that I discussed last month, placing `hHook` directly into the default non-shared (one per instance) data segment. At the same time, I altered the aforementioned `PostThreadMessage` call to look like this:

```
fOk = PostThreadMessage(
  dwDesktopThread, WM_NULL,
  GetCurrentThreadId(),
  (LPARAM)hHook);
```

This time, we're not only passing the application thread ID in `wParam`, but we're passing the hook variable in `lParam`. I also edited the `GetMsgProc` routine to look as shown in Listing 1.

As a teenager, one of my great passions was electronics and amateur radio. While most of my contemporaries were at a disco or down the pub, I generally had my head buried in a schematic for some piece of electronics. No, I haven't changed very much, have I? One of the most intriguing concepts I learned about was a type of radio receiver where a single valve or transistor could be magically persuaded to perform two jobs at once.

It's occasionally possible to do the same thing in software: if you compare Listing 1 with last month's code, you'll see that I've eliminated the static `Boolean` variable `FirstTime` which Jeff used to determine whether this was the first invocation of `GetMsgProc`. This has now been replaced by a test of `hHook` which is `NULL` on the first call. Thus, `hHook` effectively does two jobs at once, telling us that this is the first call, and subsequently being used to pass messages on via `CallNextHookEx`.

If you're finding this all a bit confusing, think of the application instance of the DLL as Instance A, and Explorer's instance as Instance B. Obviously, these two instances share the same code, but because we've eliminated the shared data segment, they each get their own independent data segment, each of which contains an (initially zero) copy of `hHook`. When Instance A starts executing, the code inside `SetDIPSHook` calls `SetWindowsHookEx` to initialise A's copy of `hHook`. The only reason that we need to store the value of `hHook` inside Instance A is because it's needed later for the call to `UnhookWindowsHookEx`. Instance A passes the value of `hHook` to Instance B via the call to `PostThreadMessage`. Within Instance B, `GetMsgProc` is called, notices that B's `hHook` is zero and retrieves the required value from the `lParam` field of the message. From then on, it's plain sailing.

For those of you who want to see the final changes that I made to Jeff's code, it is included with this month's project files within RichterDLL.zip. However, once I got the DLL to the stage where I could ditch the shared data segment, I decided to ditch C/C++ too! The new Delphi-written DLL ended up about one third the size of the C/C++ equivalent, natch. While I was at it, I decided to move the `GetDesktopListView` and `GetDesktopThread` functions into the DLL. They're not strictly needed in the application, and doing things this way simplifies the DLL interface.

```
LRESULT WINAPI GetMsgProc (int nCode, WPARAM wParam, LPARAM lParam)
{
   if (hHook == NULL)
   {
      hHook = (HHOOK)((LPMSG)lParam)->lParam;
      // Create the server window to service client requests
      CreateDialog (g_hinstDll, MAKEINTRESOURCE (IDD_DIPS), NULL, DIPS_DlgProc);
      // Tell the original application that server is ready
      PostThreadMessage (((LPMSG)lParam)->wParam, WM_NULL, 0, 0);
   }
   return CallNextHookEx (hHook, nCode, wParam, lParam);
}
```

➤ *Listing 1*

Designing A New Interface

Right then, time to roll up our sleeves and add some improved functionality to the new DLL. In the original code, the `WM_APP` message was used to effect communication between the DLL and the host application. Mr Richter used the `lParam` field to distinguish between a call to save the desk layout and another call to restore the desk layout. In our new, improved, version, we want to provide a lot more functionality, and this inevitably entails a lot more messages. Although we could stick with Jeff's scheme (using `WM_APP` for everything) it means that we would lose the `lParam` field, which is really too high a price to pay. If you read the SDK documentation, you'll see that it defines `WM_APP` as defining the start of a *range* of messages which are all available for use by an application in sending messages to a private window class. The hidden dialog window implemented by the DLL certainly fits the definition of a private window class, since nobody knows about it except our DLL, not even Explorer, even though it is in fact running in Explorer's process context. Furthermore, the range of available `WM_APP` messages is quite large: `$8000` through to `$BFFF`, which should be more than enough for our needs!

Accordingly, I decided to use a different message number for each function. Let's work through an example to make this a bit clearer. If you look inside COMMCTRL.PAS, you will find a routine, `ListView_SetTextColor`, which looks as shown in Listing 2.

Essentially, this is just a wrapper function which uses the low-level `LVM_SetTextColor` message to set the foreground colour of the text labels which appear beneath every listview item when in 'icon' mode. We can define an equivalent message for our desk manager DLL like this:

```
DM_SetTextColor = wm_App + 1;
```

Having done that, we can then 'field' the message within the DLL as shown in Listing 3.

I decided to make life nice and simple for the client, so you will see that when the desktop text colour is changed, the DLL code automatically calls a routine called `RedrawItems` which forces the desktop to immediately update its on-screen display. Ordinarily, this wouldn't happen until the next `wm_Paint` message was received for each item. The implementation of the `RedrawItems` code is shown in Listing 4.

While I was busy writing this code, it occurred to me that the `GetDesktopListView` routine is being called extensively (eg three times in Listing 4) and since this routine makes use of `FindWindow` to search the top-level application window list, it might represent a performance bottleneck. Accordingly, I rewrote `GetDesktopListView` so as to store the listview

➤ *Listing 2*

```
function ListView_SetTextColor(hwnd:
HWND; clrText: TColorRef): Bool;
begin
  Result := Bool(SendMessage(hwnd, LVM_SETTEXTCOLOR, 0, clrText));
end;
```

```
case Msg of
  // -------- Set desktop text colour: lParam = new color
  DM_SetTextColor:
  begin
    Result :=
      Bool(SendMessage(GetDesktopListView, lvm_SetTextColor, 0, lParam));
    RedrawItems;
  end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
procedure RedrawItems;
var
  Count: Integer;
begin
  Count := SendMessage (GetDesktopListView, lvm_GetItemCount, 0, 0);
  SendMessage (GetDesktopListView, lvm_RedrawItems, 0, Count - 1);
  UpdateWindow (GetDesktopListView);
end;
```

window handle in a global variable, only calling `FindWindow`, etc, the first time that it's called.

This routine simply sends an `lvm_GetItemCount` message to the desktop to figure out how many items are present. It then issues a `lvm_RedrawItems` message, setting the `wParam` field to zero and the `lParam` field to the item count less one. This has the effect of adding the screen area of all the desktop items to the update region for the listview window, and the redraw operation takes place once the `UpdateWindow` API routine is called.

Of course, this is a relatively trivial function. It doesn't involve any pointers, and pointers are the whole reason why we're injecting a DLL into Explorer's process space. The above example is there simply to illustrate the direction that I'm going in.

Introducing WM_COPYDATA

Things get more interesting when we try to do something such as retrieving the caption of a specified desktop item. If you look at the SDK documentation for `LVM_GetItemText`, you'll see that we have to fill in the fields of a special `LVITEM` data structure and pass the address of that structure to the listview control using the aforementioned message number; this sounds pretty tedious. Fortunately, Borland provide a neat little wrapper routine (I believe it's implemented as a macro for C/C++ developers) called `ListView_GetItemText`, which eliminates the need to faff around with the `LVITEM` structure.

After some thought, I defined a new message, `DM_GetItemText`, as below:

```
DM_GetItemText = wm_App + 4;
```

This message uses the `wParam` field to pass the index of the desktop item that we're interested in. The `lParam` field is unused or, to put it another way, we make no attempt to pass a buffer address to the DLL. Why? Because the hidden dialog's window procedure executes in the context of Windows Explorer and not in the context of the Delphi application, so the address would be invalid. What, then, do we do with this message inside the DLL, and how do we pass the caption text back to the application?

All is revealed in Listing 5. This routine, `HandleGetItemText`, is triggered when a `DM_GetItemText` message is received by the dialog window. As parameters, it takes the handle of the dialog window and the index value passed via `wParam`. The first job is to determine the number of desktop items, this information being used to validate the passed index value. If everything checks out OK, the `ListView_GetItemText` routine is then called, returning the caption of the required item in `buff`.

```
function HandleGetItemText (DlgWnd: hWnd; Index: Integer): Bool;
var
  Count: Integer;
  cds: TCopyDataStruct;
  buff: array [0..255] of Char;
begin
  Result := False;
  Count := SendMessage (GetDesktopListView, lvm_GetItemCount, 0, 0);
  if (Index >= 0) and (Index < Count) then begin
    buff[0] := #0;
    ListView_GetItemText (GetDesktopListView, Index, 0, buff, sizeof (buff));
    cds.dwData := DM_GetItemText;
    cds.cbData := lstrlen (buff) + 1;
    cds.lpData := @buff;
    Result :=
      Bool(SendMessage(AppWindow, wm_CopyData, DlgWnd, Integer (@cds)));
  end;
end;
```

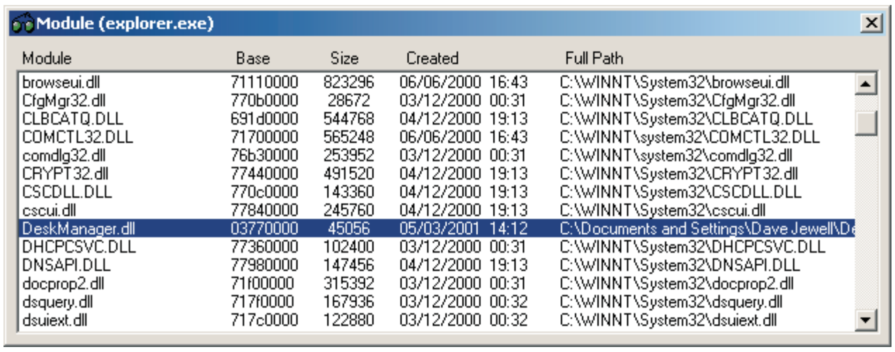
➤ Listing 5

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

➤ Listing 6

And then comes the sneaky bit. One of the least known, most overlooked, and yet most potentially useful Windows messages is `WM_COPYDATA`. This allows us to send data across process boundaries, without worrying about pointer problems. The `WM_COPYDATA` message takes two parameters in `wParam` and `lParam`. The former is the handle of the window that's sending the message, whereas `lParam` is a pointer to a special data structure, `TCopyDataStruct`, which describes the data we want to send to the other application, see Listing 6.

The Microsoft SDK documentation can't quite seem to make up its mind as to whether `dwData` is a simple integer or a pointer. In actual fact, it's an integer. You can use this optional field to pass an application-defined value back to the calling application. As you can see from Listing 5, I chose to pass back the message number of the original message, `DM_GetItemText`. This acts as a sort of 'handshake', making it easier for the calling application to verify that the received `WM_COPYDATA` message is the response to a `DM_GetItemText` message. The next field, `cbData`, contains the number of bytes of



➤ Figure 2: This may look like a screenshot from last month, but this time DeskManager.DLL (shown running in Explorer's process space) is written in 100% Delphi! Ah bliss...

Module (explorer.exe)

Module	Base	Size	Created	Full Path
browseui.dll	71110000	823296	06/06/2000 16:43	C:\WINNT\System32\browseui.dll
CfgMgr32.dll	770b0000	28672	03/12/2000 00:31	C:\WINNT\System32\CfgMgr32.dll
CLBCATQ.DLL	691d0000	544768	04/12/2000 19:13	C:\WINNT\System32\CLBCATQ.DLL
COMCTL32.DLL	71700000	565248	06/06/2000 16:43	C:\WINNT\system32\COMCTL32.DLL
comdlg32.dll	76b30000	253952	03/12/2000 00:31	C:\WINNT\system32\comdlg32.dll
CRYPT32.dll	77440000	491520	04/12/2000 19:13	C:\WINNT\System32\CRYPT32.dll
CSCDLL.DLL	770c0000	143360	04/12/2000 19:13	C:\WINNT\System32\CSCDLL.DLL
cscui.dll	77840000	245760	04/12/2000 19:13	C:\WINNT\System32\cscui.dll
DeskManager.dll	03770000	45056	05/03/2001 14:12	C:\Documents and Settings\Dave Jewell\De
DHCPCSVC.DLL	77360000	102400	03/12/2000 00:31	C:\WINNT\System32\DHCPCSVC.DLL
DNSAPI.DLL	77980000	147456	04/12/2000 19:13	C:\WINNT\System32\DNSAPI.DLL
docprop2.dll	71f00000	315392	03/12/2000 00:31	C:\WINNT\System32\docprop2.dll
dsquery.dll	717f0000	167936	03/12/2000 00:32	C:\WINNT\System32\dsquery.dll
dsuiext.dll	717c0000	122880	03/12/2000 00:32	C:\WINNT\System32\dsuiext.dll

data that we want to pass to the target application, and lpData is a pointer to the data itself. Armed with this information, you should be able to understand the rest of Listing 5.

It's easy to get into a 'deadly embrace' situation here. The SDK documentation states that you should use WM_COPYDATA synchronously rather than asynchronously. In other words, use SendMessage rather than PostMessage. This is important because it guarantees the validity of the data (in the context of the originating process) until the transfer is complete. Unfortunately, we've only been using PostThreadMessage to communicate with the Delphi application and a post is a post is a post!

I decided to play safe and modify the DLL interface further such that there are actually two interface routines:

```
function DeskManagerLoad(
  AppWindow: hWnd): Bool;
  stdcall;
  external 'DeskManager.dll';
function DeskManagerUnload:
  Bool; stdcall;
  external 'DeskManager.dll';
```

Their use should be self-explanatory, except for the window handle that's passed to DeskManagerLoad. This window handle is saved by the DLL and all subsequent WM_COPYDATA responses (and possibly more besides) are sent directly to this window handle. This corresponds to the new variable, AppWindow, which you can see used in Listing 5.

Why did I mention the dreaded deadly embrace? Well, imagine what would happen if you used SendMessage to synchronously send a DM_GetItemText message from the Delphi application. As we've seen, this would trigger a call to SendMessage from the DLL, using WM_COPYDATA to try and send the desktop caption text back to the Delphi app. This message couldn't be processed by the Delphi application because it would be blocked waiting for the original SendMessage to complete. Thus you get a deadlock situation where two processes are sat waiting for each other to complete. *'This application is not responding. Please click...'*

In principle, you could get around this by implementing a separate thread within the Delphi program, whose sole job is to 'drive' the DLL interface, but you'd then have to ensure that you sent messages from one thread and received them on another thread; nah: too messy. Since the DLL *has* to send WM_COPYDATA synchronously, I decided that the simplest solution was to use PostMessage on the application side for any message that is going to trigger a WM_COPYDATA response.

Of course, this means that the application has got to somehow post a message and then twiddle its thumbs waiting for a WM_COPYDATA reply to arrive. On the other hand, we want what looks like a nice easy synchronous API from the client perspective. How to achieve it?

Wait For It, Wait For It...

Well, quite easily, actually. Take a look at Listing 7, which shows the basic idea. GetItemCaption takes a single Index value and returns the corresponding item's caption string. From the perspective of the routine's caller, the operation is completely synchronous. Inside GetItemCaption, we call a routine called PendOnDeskTopMessage. This same routine is called whenever we want to wait for data to be returned from the desktop manager DLL.

PendOnDeskTopMessage begins by setting a private integer field of the form, dmResponse, to -1 and then posts the required message to the dialog window. Finally, it sits in a loop calling Application.ProcessMessages while waiting for the WM_COPYDATA response to be received.

If you have never used Application.ProcessMessages before, think of it as the 'message pump' which keeps your Delphi application alive. If you didn't call this routine, but just sat around waiting for something to happen, then nothing ever would: at least, in terms of received Windows messages!

You'll notice that while driving the message pump, the loop checks to see if dmResponse has been set to the value of the initiating message. You'll appreciate that this is the 'handshake' mechanism that I referred to earlier. From a purist's point of view, this little sanity-check is not particularly rigorous. Strictly speaking, if you wrote the code in such a way as to post multiple messages to the management DLL, there's no guarantee that the responses to those messages would be received in the same order that the messages were sent. For this reason, I've resisted the temptation to get too fancy, and every operation is essentially atomic.

For similar reasons, you might be wondering why I bother to

```
function TForm1.PendOnDeskTopMessage (Msg, wParam, lParam: Integer): PChar;
begin
  dmResponse := -1;
  PostMessage (hDeskWin, Msg, wParam, lParam);
  while dmResponse <> Msg do Application.ProcessMessages;
  Result := dmData;
end;
function TForm1.GetItemCaption (Index: Integer): String;
begin
  Result := '';
  if (Index >= 0) and (Index < GetItemCount) then
    Result := StrPas (PendOnDesktopMessage (DM_GetItemText, Index, 0));
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
procedure TForm1.WMCopyData (var Message: TWMCopyData);
begin
  // Make sure it's from the desktop manager DLL
  if Message.From = hDeskWin then with Message, Message.CopyDataStruct^ do begin
    ReallocMem (dmData, cbData);
    Move (lpData^, dmData^, cbData);
    dmResponse := dwData;
    Result := Integer (True);
  end;
end;
```

range-check the value of the `Index` parameter on entry to `GetItem-Caption`. After all, it's checked again within the DLL. But if you look at the way the DLL code has been written, an invalid index value means that the DLL will never send `WM_COPYDATA` back to the host application, which in turn means that `PendOnDeskTopMessage` would spin forever, waiting for a response that never comes. Obviously, this is a scenario that's best avoided.

The final part of the jigsaw is the `WMCopyData` routine in Listing 8. This is a standard message handler method, set up so as to be triggered whenever a `WM_COPYDATA` message is received, thus:

```
procedure WMCopyData(
  var Message: TWMCopyData);
  message wm_CopyData;
```

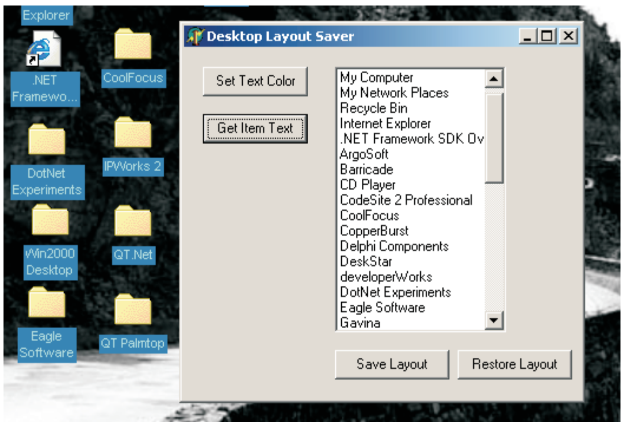
If the incoming message was indeed from the hidden dialog window, then the data pointed to by the private `dmData` field is reallocated to the size of the incoming data and the data then copied across. The SDK documentation stresses that this data is strictly transitory, if you want to copy the information, you have to do it at the point where the `WM_COPYDATA` message is processed. Finally, the `dmResponse` field is set to the passed `dwData` field, thus completing our simple-minded little handshake mechanism, and (hopefully!) taking `PendOnDeskTopMessage` out of its wait loop.

Just to illustrate how easy to use this is, Listing 9 shows a small code snippet which fills a listbox with the captions of all the currently defined desktop icons. When I wrote this code, and particularly relied upon the `WM_COPYDATA` mechanism, I was a little worried that performance might be less than

➤ *Listing 9*

```
procedure TForm1.Button2Click(
  Sender: TObject);
var
  Idx: Integer;
begin
  ItemNames.Clear;
  for Idx := 0 to GetItemCount-1 do
    ItemNames.Items.Add(
      GetItemCaption (Idx));
end;
```

➤ *Figure 3: For now, the test app allows you to tweak the desktop's foreground text colour and retrieve a list of desktop icon captions. Stay tuned for more next month!*



My Computer
My Network Places
Recycle Bin
Internet Explorer
.NET Framework SDK Ov
ArgoSoft
Barricade
CD Player
CodeSite 2 Professional
CoolFocus
CopperBurst
Delphi Components
DeskStar
developerWorks
DotNet Experiments
Eagle Software
Gavina

sparkling. After all, I've got over 40 items on my desktop (some folks have a lot more!) and getting the caption for each of these involves posting a message and then a wait for the DLL to send a response. I was concerned that it might be necessary to get the whole thing in one go, concatenating all the caption strings into one data block with separators between them. As it turned out, this proved to be quite unnecessary as the above code appears to execute virtually instantaneously on my ageing 500MHz Pentium III *[Dave checks the price of the latest 1.4GHz P4, blows the dust off his wallet, winces painfully and hastily shoves it back into his pocket... Ed]*.

That said, there's obviously a strong argument for treating the entire desktop as a single logical chunk of data. If your application gets the desktop icon count one minute, and then tries to access the caption of one of those icons half an hour later, you shouldn't be too surprised if the goalposts have moved in the meantime, and the desktop item you're interested in has been deleted. One way around this would be to take a 'snapshot' of the state of play, or get/set all the icons at the same time, as per Jeff's original code.

Next Month

I'd originally planned that this would be the second and final article on the subject of taming the Windows desktop. However, I have to confess that it won't be. This month, I succumbed to the tantalising puzzle of how to eliminate the shared data segment from Mr Richter's DLL and, once I'd solved that problem, the further temptation of rewriting the whole thing in Delphi was irresistible!

Even so, this definitely isn't time wasted. If you've worked through this month's and last month's columns, you should by now have a very firm grasp of how to break down the walls between Win32 processes using DLL injection, and you will likewise have learned a valuable technique for inter-process communication using `WM_COPYDATA`. I'm not saying that you should routinely set out to defeat the security mechanisms that Windows puts in your way, but it's nice to know how to do this kind of thing when you need to write an operating system utility that's a little out of the ordinary.

Now that we've got the architectural foundations well and truly in place, next month's third (and final!) article on desktop access and management will flesh out the existing code with a lot more functionality for getting/setting icon captions, positions, spacing, colours and assorted other properties. Also, I really *will* package the whole thing up into a reusable component for ease of use. Trust me, I'm a doctor...

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level Windows programming and cross-platform issues. He is the Technical Editor of *The Delphi Magazine*. You can contact Dave at TechEditor@itecuk.com