Beating The System: Taming The Windows Desktop, Part 1

by Dave Jewell

For some while now, the Editor has been nagging me to write an article about the Windows desktop. I originally discussed the desktop some while ago in *The Delphi Magazine*, but I didn't cover the all-important issue of how to programmatically arrange or rearrange desktop icons from a Delphi application. For reasons that will become obvious as you read through this article, such things aren't as trivially easy as they ought to be, thanks largely to the memory protection mechanisms built into Windows.

But why would you want to do this? Anyone who has experienced a major system crash (and, as software developers, we tend to experience more than most!) will know that Windows has a nasty habit of resetting the position of all the icons on your desktop after certain types of crash have occurred. This was a problem under Windows 95 and 98, it's still a problem under Windows ME, and it most definitely also happens under Windows 2000 and beta versions of Whistler.

System crashes aside, you might inadvertently select one of the `Arrange Icons` commands from the desktop window's right-click context menu: it doesn't have an `Undo` option! Also, application developers frequently like to check out the appearance of their program at a variety of screen resolutions. Here again, the oh-so-helpful Windows Explorer obligingly comes along and silently rearranges your desktop icons whenever the size of the desktop changes, meaning that you've got to go through the tedious process of putting them back where they were once you return the screen resolution to normal.

There are a number of shareware utilities around which will take a 'snapshot' of your desktop icon positions, and restore the original layout on demand, one such example being something called 12Ghosts SaveLayout, one of several utilities contained within the 12Ghosts package. This article is intended to show you how to create similar functionality from within a Delphi application.

But I'm Not Interested In The Desktop!

Even if you're not much interested in programmatically controlling the desktop icons, I'd encourage you to read this article. This is because it demonstrates a much more general technique called DLL injection. In other words, how to 'inject' a DLL into the address space of another, separate, Windows process. This is a powerful technique which, in special circumstances, can be used to break down the barriers between different processes, each running in its own address space. This technique can be useful for writing debugging or 'spy' tools, and for creating specialised utilities which tweak the operating system or other applications in various ways.

With that as an introduction, let's start at the beginning by reviewing the implementation of the Windows desktop. Start Delphi, click the `Win32` tab on your component palette and add a `TListView` control to your form. You'll notice that the default value of the `ViewStyle` property is `vsIcon`. If you bring up the online help for `vsIcon`, you will see that it states: *'Each item appears as a full-sized icon with a label below it. The user can drag the items to any location in the list view window.'* Hmm... that sounds a bit like the Windows desktop, doesn't it? Sure enough, this is exactly how Microsoft chose to implement the desktop, using a listview control. Of course, I'm not suggesting that the desktop is based around a Delphi control! It's important to remember that `TListView` is just a wrapper around the listview implementation contained within COMCTL32.DLL, and it's this underlying window class in the common controls library which implements the desktop.

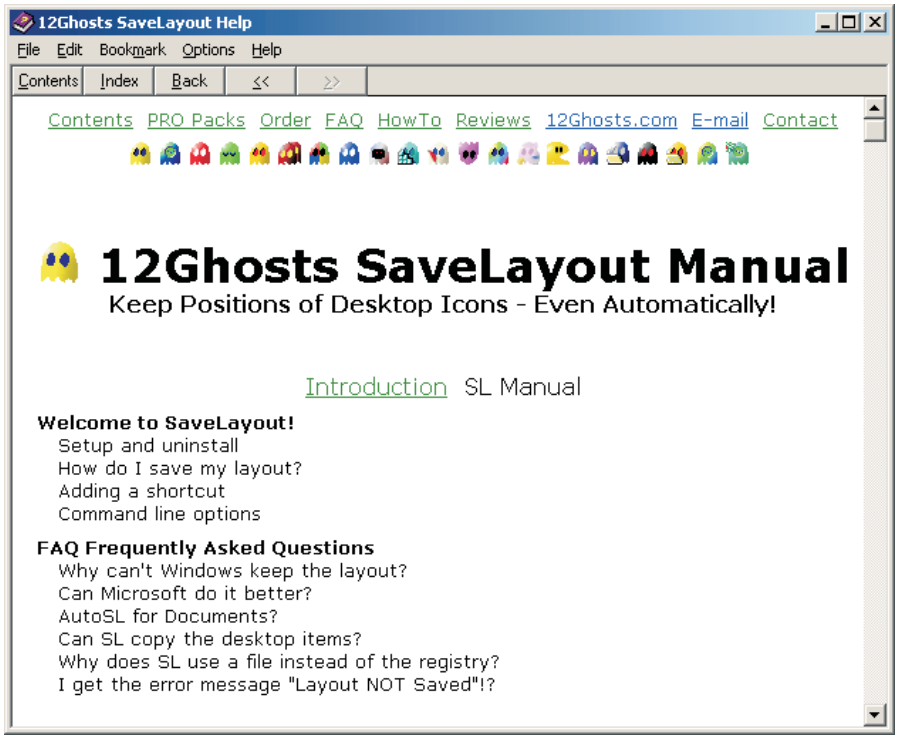
Before we can do anything with the desktop, we need to get the API-level handle of this listview control. Fortunately, this is quite easy to do, one possible implementation being shown in Listing 1.

This routine starts out by searching the list of top-level windows for a window which has a class name of `ProgMan`. This corresponds to the Windows Explorer itself, also known as the Windows shell. Once upon a time, Explorer was known as Program Manager and, for reasons of backward compatibility, Microsoft chose to retain the same class name when rewriting the shell for Windows 95.

Once we've got the shell's top-level window, we can easily obtain the window handle of the desktop.

➤ *Listing 1*

```
// Get a window to the ListView control which implements the desktop
function GetDesktopListView: HWnd;
var
  buff: array [0..255] of Char;
begin
  Result :=
    GetWindow (GetWindow (FindWindow ('ProgMan', Nil), gw_Child), gw_Child);
  if Result <> 0 then begin
    GetClassName (Result, buff, sizeof (buff));
    if buff <> 'SysListView32' then Result := 0;
  end;
  if Result = 0 then
    raise Exception.Create ('Desktop ListView control not found');
end;
```



➤ *Figure 1: 12Ghosts SaveLayout (from www.12ghosts.com) is an example of a shareware application which, broadly speaking, does the same sort of thing as discussed here, injecting its own DLL into the address space of Explorer.*

It's simply the 'grandchild' of the window handle we've got. In other words, the desktop listview window handle is simply the first child of the first child of the Explorer window. We can retrieve this handle by calling the API routine `GetWindow` twice, passing `gw_Child` in both cases.

For the technically curious, you might be wondering what the intermediate child (as opposed to grandchild) window is. The class name of this window is `SHELLDLL_DefView` and Microsoft refer to it as the *shell view router*. You can read more, a lot more, about the internal implementation of the Windows desktop at

```
http://msdn.microsoft.com/
  library/techart/webview.htm
```

This is a fascinating MSDN article by Dino Esposito, who seems to have become something of a guru in the field of shell desktop programming.

Returning to my `GetDesktop-ListView` routine, you will see that it includes an important sanity check. Taking the listview window handle, it calls `GetClassName` to ensure that this really is the handle of a listview control. If it isn't, then the function bottles out with an exception. I've tested this code under Windows 2000, and Windows 98, and it works fine in both environments.

One Man's Pointer Is Another Man's Poison

OK, so we've got a handle to the desktop listview control. What can we do with it? As it stands, not a great deal! For sure, you can do one or two straightforward things, such as determining how many items are on your desktop. Here's how to do this:

```
NumItems :=
  ListView_GetItemCount(
  GetDesktopListView);
```

The `ListView_GetItemCount` routine, incidentally, is defined inside the `CommCtrl` unit. It works by sending a `LVM_GETITEMCOUNT` message to the control via `SendMessage`. This works, and if you care to count up all your desktop items, you'll see that it's accurate.

Amusing though this all is, what we *really* want to do is examine the individual desktop items, figure out what captions they have, change their position, and so forth. And that's where we come unstuck! Consider the `ListView_-GetItemText` routine, whose prototype is:

```
function ListView_GetItemText(
  hwndLV: HWND; i, iSubItem:
  Integer; pszText: PAnsiChar;
  cchTextMax: Integer):
  Integer;
```

As arguments, this routine takes the window handle of our listview control, an integer which designates the item we're interested in and a sub-item parameter which should be set to zero to get the item's text. It also receives a pointer to a buffer to receive the text string, and an indication of the buffer size as the final parameter. Unfortunately, when we call this routine, the desktop listview will stubbornly refuse to give us the text we're after.

The problem, of course, is that we're trying to pass a pointer to a window that we don't own and didn't create: we're trying to pass a pointer to a different process in another address space. The address of the passed character buffer might make sense to us, but it doesn't make sense to Windows Explorer, and it's within the context of Explorer that this control is executing. Within the guts of Windows, the code determines that the passed address isn't valid and simply ignores our request.

The somewhat counter-intuitive solution to this problem is to use a technique called DLL injection. I'm not 100% sure where this terminology comes from, but I believe it's due to Jeffrey Richter who describes the technique in his excellent book, *Advanced Windows* published by Microsoft Press. This follows on from earlier work which he published in the *Microsoft Systems Journal*.

Basically, the idea is to create a DLL which is invoked from our own application. Once invoked, it searches for and finds the desktop

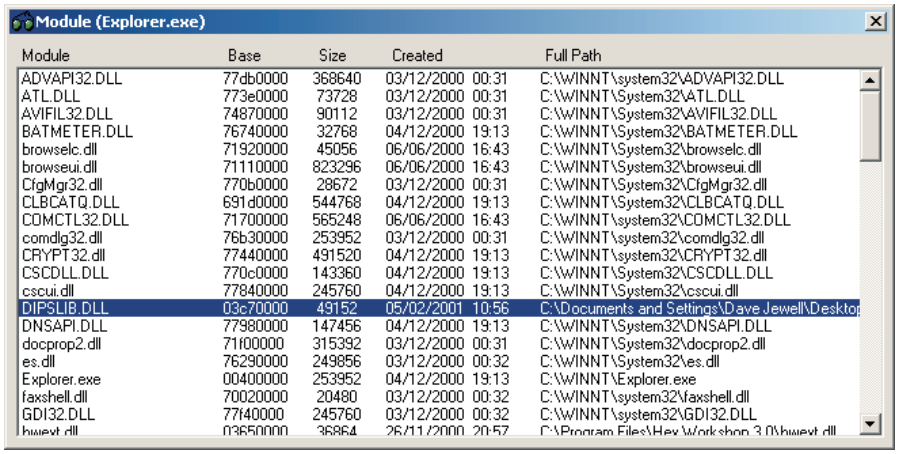
listview control referred to earlier. It then installs a Windows message hook, hooking the thread which owns the listview thread. This will obviously correspond to a thread inside the Explorer process itself. The key thing here is to ensure that the message handling function is located inside the DLL. This has the effect of 'pulling' the DLL into the process space of the hooked application, in this case Windows Explorer.

Once our DLL is executing within the context of the Explorer, we can then communicate with the desktop tree list control just as easily as if we'd created it ourselves. Effectively, the DLL has become a sort of go-between, connecting our own application with Explorer's address space. That's a very brief overview, but there's rather more to it than that, as we shall see.

Quiche Eaters: Cover Your Eyes!

This particular issue of *Beating The System* is unusual in more ways than one, not least because I've been forced into the position of using a DLL that wasn't written in Delphi! That's right: much of the code described below is based around Jeffrey Richter's original DLL which was written using Visual C++. What justification could I have for polluting this august publication with C++ code? Read on.

➤ *Figure 2: Here's the proof of the pudding. This screenshot (taken from a process viewer application) shows DISPLIB.DLL as a loaded module within the Explorer process.*



➤ *Above: Listing 2*

```
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
DWORD g_dwThreadIdDIPS = 0;
#pragma data_seg()
// Instruct the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/section:Shared,rws")
```

Below: Listing 3

```
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
DWORD g_dwThreadIdDIPS = 0;
#pragma data_seg()
// Instruct the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/section:Shared,rws")
```

I'm a dedicated Delphi zealot and I'd go so far as to say that, 99% of the time, Delphi will do everything that Visual C++ can do, and it'll do it a lot more quickly and elegantly. But, every so often, one comes up against that stubborn 1%, and when you do, there's sometimes no option but to reach for a C/C++ compiler. If you look at the source code for Jeff's original DLL code, you'll find the interesting looking snippet in Listing 2.

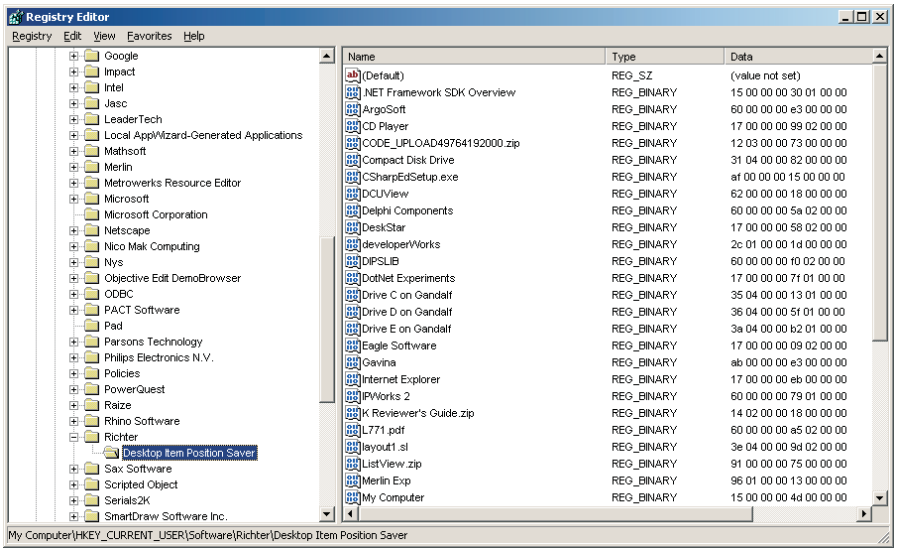
This is probably pure gobbledygook to the average Delphi developer, so I'll walk you through what it means. The first statement is a pragma, somewhat like the ability to embed compiler directives into a Delphi application. In this case, the pragma is telling the compiler that any data variables declared from this point on should be placed into a new data segment (or 'section', in 32-bit programming terminology). At the same time, the pragma allows us to specify a new name for our custom section, in this case `Shared`.

This is followed immediately by the definition of a couple of global variables. Both of these variables get placed into the `Shared` data section. For the moment, we won't worry too much about what these variables are there for. After these two declarations, we see another pragma, this time without an accompanying section name. This causes the compiler to restore the status quo, placing subsequent variables back in the default data section.

But wait, there's more! The final pragma statement introduces an embedded linker command in the form of a comment. When the Visual C++ development environment builds the DLL, it finds this command and appends it to the list of directives that are fed to the Microsoft linker. In this case, the directive tells the linker that the `Shared` section is to be created using the read, write and shared attributes. That's the meaning of the, somewhat terse, `rws` string within the comment.

So, the net effect of all these hieroglyphics is to create a new section which contains only the `g_hhook` and `g_dwThreadIdDIPS` variables. At the same time, the section is marked as being not only read and write, but also shared, and it's the shared attribute which is the important one. Remember I said that the DLL gets loaded into the process space of Windows Explorer *and* our Delphi application? Ordinarily, when an application loads a DLL, it receives a completely new data segment. But in this case, we want to be able to share certain variables, making them available when the DLL is called from within the context of our application, *and* when it's called by Explorer.

But hang on a minute, Dave, Explorer knows nothing about our



➤ *Figure 3: Jeffrey Richter's DLL stores desktop layout information into the Windows registry. Coming up with a more general storage mechanism is one of the things we'll be looking at next time.*

DLL, so how does it get to call the thing? That's why I mentioned the need to hook the message queue of the thread belonging to the desktop listview control. Once we've done that, then Explorer is essentially calling into our DLL each time it processes a message for the Windows desktop.

It's worth pointing out that when we call a routine in this shared DLL we can't just expect to dive into the routine and start mucking about in the address space of Explorer. The key word is *context*. When called from our app, we're running in the application's context. When called from Explorer, the DLL is running in Explorer's context.

In order to get useful information from the desktop listview, we have to call the DLL from the Delphi application, posting a message to the thread which owns the listview and passing the address of a shared buffer which will receive any relevant information. This message is then serviced by Explorer, causing our message hook to be invoked and allowing the relevant information to be posted back to the shared buffer.

But I'm jumping ahead of myself. The real reason why this DLL has to be written in C/C++ is because Delphi, unfortunately, doesn't give us anything like the same level of control when it comes to creating custom sections, marking them as shareable and so forth. Sad, but true.

A Walk On The Wild Side

The little program on this month's disk comes in two parts. DeskApp is an application built with Delphi 5, and DIPSLIB.DLL is a slightly massaged version of the DLL developed by Jeffrey Richter. So let's roll up our sleeves and take a tour of the code. The application's `FormCreate` handler begins by loading the shared DLL, raising an exception if it can't be found. This is immediately followed by a call to `SetDIPSHook`, the code for which is given in Listing 3.

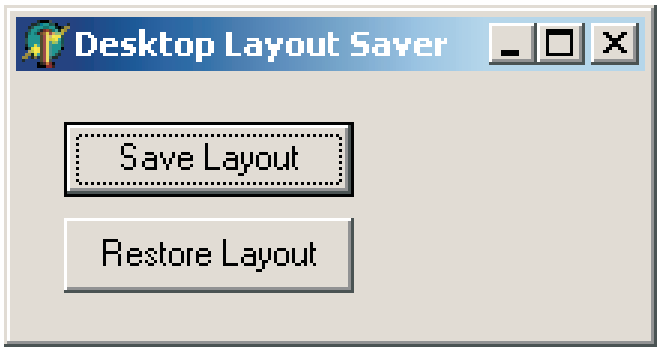
As an argument, this routine is passed a thread ID which is retrieved from the `GetDesktopThread` function, which uses the Windows API routine `GetWindowThreadProcessID` to retrieve the ID of the Explorer thread which is responsible for the desktop listview control. Inside `SetDIPSHook` we retrieve the address of the corresponding DLL function and call it, passing the required thread ID.

Inside the DLL (consult the C++ source code) the `SetDIPSHook` routine does one of two things depending on whether or not the supplied thread ID is non-zero. Assuming it's non-zero, the library code calls `GetCurrentThreadId` to retrieve the ID of the calling thread. This will obviously be the main thread ID of our Delphi application since we've just called into the DLL. Next, it uses the `SetWindowsHookEx` routine to hook into the message queue of the Explorer thread that owns the list view, passing the instance handle of the DLL module itself. At this point, the DLL hasn't yet been mapped into the address space of Explorer, but it's about to be.

If everything went OK, the library code uses `PostThreadMessage` to post a `WM_NULL` message to the Explorer thread, and it then exits, flagging success as the function result. This call to `PostThreadMessage` is very subtle: remember that at this point we're still running in the context of the Delphi application which loaded the DLL. Once that initial `WM_NULL` message is about to be processed, the Windows kernel is forced to load the DLL into the process address space of Explorer. You can see this very clearly in Figure 2, which shows the list of loaded modules with Explorer's address space.

Back inside the DLL, the `WM_NULL` message arrives at the `GetMsgProc` routine. This code contains a static variable, `fFirstTime`, which causes a hidden dialog window to be created on the first access to `GetMsgProc`. Once the dialog is created (remember, we're running in Explorer's context now), the code posts another `WM_NULL` message back to the original Delphi application using the thread ID that was stored when the library was first loaded. This is used to tell the application that the DLL is now initialised and open for business.

Back in the Delphi executable, the `FormCreate` code calls `GetMessage`, waiting for that magic `WM_NULL` message to arrive. Bear in mind that it would be a very bad idea to replace that `GetMessage` call with a reference to (say) `Application.ProcessMessages`. The Delphi application has to be blocked until the `WM_NULL` message arrives whereas `Application.ProcessMessages` is a non-blocking call: it will return immediately if the message hasn't arrived yet. If the subsequent `FindWindow` call were to



➤ *Figure 4: The story so far. Next month we'll add a lot more bells and whistles to the testbed app.*

execute before the DLL was ready, then the `hDeskWin` variable could end up as `Nil`, and that would be very bad news indeed.

This `FindWindow` call looks for a top-level window with a caption name of `Delphi Desktop 2001`, corresponding to the hidden dialog window that is created on the 'Explorer side' of the DLL. Because this dialog is executing in Explorer's context, we can send messages to it, and it will immediately be able to do anything it likes with the desktop listview because the listview control exists in the same process space. This is illustrated by the way in which Jeffrey Richter has implemented a couple of 'commands' which are piggy-backed onto the `WM_APP` message. By sending these messages from our Delphi application, we can cause the Windows desktop to be automatically saved and restored on demand.

The Full Enchilada

If you want to play with the code on this month's disk, go ahead, but be warned that this is a work in progress. The current implementation seems to work well under Windows 98 and Windows 2000, but I haven't tested it on any other platforms. I would especially caution you against trying to run multiple copies of the Delphi application, or using it in conjunction with another utility which hooks the Windows desktop in the way it is done here.

Of course, as Delphi programmers, we are used to a lot more functionality than this! For sure, it's nice being able to load and save the entire desktop on demand, and do it programmatically from a Delphi application, but wouldn't it be nice if we could, for example, retrieve the caption of a selected desktop item, move a particular item to a certain screen location and so forth. This, of course, is exactly what we'll be doing next month when I'll add a lot more functionality to our embryonic desktop interface and wrap the whole thing up into an easy to use Delphi component. So stay tuned...

Dave 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