Delphi Internals: Using And Writing DLLs

by Dave Jewell

Over the coming months, the ***Delphi Internals*** column is going to peer under the hood of Delphi, examining many of the low-level aspects of Delphi programming. Some of the issues we're going to cover include dynamic link libraries, debugging Delphi applications, the inside story behind exception handling, how to interface to other programming languages and much more!

If there's anything specific that you'd like to see covered, then please write to me *c/o The Delphi Magazine* and tell me what you're interested in. Do bear in mind though, that my mission in life is to cover low-level techie programming. I may not react favourably if asked how to go about designing an accountancy package in Delphi! Alternatively, you can send email to me via the Internet at djewell@cix.compulink.co.uk

Using DLLs With Delphi

OK, enough of the social niceties – let's roll up our sleeves and get down to business. In this issue we're going to look at Dynamic Link Libraries, or DLLs for short.

I'm assuming that you've got no previous experience of using Borland Pascal, but that you are familiar with the basic concepts behind DLLs and that you've had some exposure to Windows programming. We'll therefore cover the essentials of using and creating DLLs from a Pascal developer's point of view before looking in more detail at how you would build a DLL in Delphi.

Whenever you examine a source file that's been created by Delphi, you'll see a USES statement near the top. This identifies which Pascal units are required to compile the file. Almost invariably, the first two units specified in the USES statement are WINTYPES and WINPROCS. These files contain the type definitions and routine declarations needed by Pascal in order to call the Windows API. Since the Windows API is implemented using DLLs, we need actually look no further than the routine declarations in the WINPROCS unit – this tells us all we need to know about calling a DLL from Delphi's Object Pascal.

Calling An External Routine In A DLL

If you look at the WINPROCS.PAS source code, you'll see that there are a huge number of routine declarations there – so many that it's difficult to see the wood for the trees. In order to make things clearer, I've written a small unit called TINYAPI.PAS which declares just a single routine, SetRect. The code for this unit is shown below. Let's see how it works.

```
unit TinyApi;
interface
uses WinTypes;
procedure SetRect(
  var Rect: TRect;
  X1, Y1, X2, Y2: Integer);
implementation
procedure SetRect; external
  USER  index 72;
end.
```

The interface part of the unit is followed by a USES clause for WINTYPES. This is then followed by the procedure declaration for SetRect itself. The important part, of course, is the actual implementation of this routine in the implementation part of the unit. The first thing you'll notice is that you don't need to repeat the list of parameters required by the SetRect routine. You can repeat the parameter list if you wish, but if you do so, then you must make sure that it exactly matches the previous definition. For example, if you refer to the four integers as X1, Y2, X2, Y2 in the interface declaration, then the compiler won't allow you to refer to those same integers as left, top, right, bottom in the implementation part of the unit, even if these are perfectly acceptable names for the parameters.

The EXTERNAL keyword is more interesting. This tells the compiler that the actual code for the interface routine isn't in the unit being compiled, but is somewhere else. DOS-based Pascal developers frequently use the EXTERNAL keyword to link a program with separately compiler assembler code. However, thanks to the magic of dynamic linking, we can tell the compiler that the code isn't going to be statically linked at all.

The next part of the statement specifies the name of the DLL containing the SetRect routine – in this case, it's the USER DLL, one of the core components of Windows itself. The compiler doesn't need to verify this information at compile time. It doesn't care at this point whether the USER DLL exists, or whether it really contains the routine we're saying is there, all these checks take place at run-time.

The final part of the implementation statement specifies an ordinal value for the routine. You should know that all routines exported by a DLL have an associated name called the ordinal number. When you import a DLL routine (as we're doing here), you need to somehow tell Windows which routine you're interested in. By specifying an ordinal number, our application will end up asking Windows for routine number 72 in the USER library. This is called linking by ordinal. Alternatively, you could omit the INDEX keyword and the ordinal number itself. You'd then be linking by name. In general, linking by name is easier (since you don't have to mess about with ordinal numbers) but it results in a slightly larger executable file and is fractionally slower at run-time. If you want to know what routines are exported by a particular DLL, and what ordinal values they have, you can use a Microsoft utility such as EXEHDR to dump the list of exported routines.

Import Units And Custom DLLs

The WINPROCS unit and the TINYAPI unit that we looked at earlier were examples of import units. An import unit has only one mission in life – its job is to take a set of DLL routines and make those routines available to the application in which it is linked. For example, when using the FlashWindow API call inside a Delphi routine, you can just call FlashWindow as if it were a local routine. You don't know, and don't care, that it's actually implemented inside the USER library. That's what an import unit is for: to make DLL routines more immediately accessible.

At this point, we've only discussed how to interface to the standard DLLs, but naturally, you can call custom DLLs just as easily. These DLLs might have been written using C, Pascal, assembler or Delphi itself – it really doesn't matter. As an example, here's a couple of procedure declarations used in one of my own programs, referring to routines in a custom DLL called TFRAME.DLL:

```
procedure FixLibrary; far;
  external 'TFRAME' index 1;
procedure UpdateTopLevelWindow(
  fDraw: Boolean); far;
  external 'TFRAME' index 2;
```

There are two important things to notice about these two procedure declarations. Firstly, you'll see that they include the procedure parameters along with the EXTERNAL keyword, DLL name and index information. That's because these declarations aren't part of an import unit. If you want to call a custom DLL, you don't have to create an import unit if you don't want to. Using the approach shown here, you can just go right ahead and put the DLL procedure declarations after the USES clause of your main program. Alternatively, you could put these same declarations at the beginning of the implementation part of a unit – that way, you'd be providing the unit with access to private routines that it needs to do its job, but you wouldn't be making the existence of those routines known to any other parts of your program.

The second point to notice is the use of the FAR keyword. When you declare routines in the interface part of a unit, they're always far. Any routine exported by a Delphi unit is a far routine, and can only be accessed by a far call. However, when you're declaring DLL routines outside of an import unit, you must use the FAR keyword. Not doing so will result in a compile error.

Avoiding The Windows API

Having just explained in some detail how it is that Delphi calls the Windows API, let me stress that you shouldn't go overboard on using the API. In fact, if you can find an equivalent call or method in Delphi's Visual Component Library (VCL) which will do the same job, then you should use the VCL call in preference to calling the API. What's the reason for this API-phobia? In one word – portability.

Borland went to a lot of trouble to make the VCL library as portable as possible. The great majority of your 16-bit Delphi applications will be able to move effortlessly across to 32-bit Windows/NT and Windows 95, provided that you've minimised the use of calls to the Windows API. In particular, you should avoid using API calls which send or receive messages. This is because, under 16-bit Windows, Microsoft often packed more than one quantity into the 32 bits of the lParam field. Under Win32, however, window handles are now 32-bits wide and many Windows messages have therefore had to adopt a different arrangement of values in the wParam and lParam fields of a message. C/C++ programmers get around this by using message cracker macros which pack and unpack the field of a message while retaining portability between the two different APIs. Under Delphi, the proper approach is to use VCL wherever possible.

Writing DLLs With Delphi

One of the many interesting things about Delphi is its ability not only to use DLLs but to create them. Using Delphi, you can create a DLL that's used by more than one of your programs, thus reducing the amount of disk space required when building a suite of programs.

Alternatively, you can use DLLs to provide functionality to your users in bite-sized pieces. For example, you might decide to sell an application which views and processes graphics files. You could build the capability to read and write some common graphics file formats into the application itself, but you might also want the flexibility to sell add-on packs which operate on even more formats. By packaging these add-on packs as DLLs, you can easily arrange for the main application to detect the presence of these add-ons and use them in a seamless fashion. Incidentally, much of Windows operates in this way; device drivers, Control Panel applets, even fonts, are specialised forms of DLL.

When news of Delphi first began to leak out, rumours were rife that you could just plug Delphi components into C/C++ applications – one American programming journal even enthused about this in its editorial. Of course, this just isn't possible – a Delphi component is at heart an Object Pascal unit. The Pascal compiler inside Delphi generates DCU files whereas C/C++ development systems use .OBJ files. It's really a case of never the twain shall meet. However, thanks to the magic of DLLs, it's possible to write a sophisticated user interface using Delphi components and call it from a C/C++ application. Actually, you could equally well call it from a straight Pascal program, or even from Visual Basic – a DLL completely breaks down the language barriers and can be used from any development system that supports DLL calls.

The Structure Of A DLL

The remainder of this article will demonstrate how to create a simple DLL using Delphi. In Delphi, or Borland Pascal, a DLL is structured somewhat as shown below:

```
library MyDLL;
uses WinTypes, WinProcs;
procedure MyFirstProc; export;
begin
  MessageBeep (0);
end;
exports MyFirstProc index 1;
begin
end.
```

If you look in the project file (the file with extension .DPR) of any Delphi application, you'll see that it starts off with the reserved word PROGRAM. By contrast, DLLs always begin with the reserved word LIBRARY. This is then followed by a USES clause for any needed units. In this simple example (probably the simplest DLL that it's possible to make), there then follows a procedure called MyFirstProcedure which does nothing except sound a beep.

You'll notice the procedure declaration uses the EXPORT specifier. This tells the compiler that the procedure is going to be called from another module – the compiler will then generate the special prologue and epilogue code which ensures that the processor's data segment register is properly set up on entry to the procedure. Any routines exported from a DLL must include the EXPORT specifier. All call-backs such as window procedures, hooks and enum procedures also need this specifier.

Finally, at the end of the source code we find an EXPORTS statement. This lists the routines that are actually exported from the DLL and assigns a unique ordinal value to each routine. This ordinal value can then be used to call routines in a DLL as discussed earlier. At this point, you're probably thinking why do we need a separate EXPORTS statement when we've already told the compiler that MyFirstProc is exported by using the EXPORT specifier? That's a good question! Remember that the EXPORT specifier is used for all call-backs and exported routines. However, we don't actually want to make call-backs and window procedures visible outside the DLL. It's the EXPORTS statement at the end of the library source file which tells the compiler what is visible and what isn't.

If you compile this simple DLL, you can then call it using the techniques I described earlier. However all that happens when you call the MyFirstProcedure routine is a beep sound – big deal. In the next installment, we'll conclude this discussion of DLLs by looking at the more exciting stuff – how to write DLLs that contain Delphi forms and components, and how to call those DLLs from other applications and development systems.

Dave Jewell is a freelance consultant/programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk