Delphi Meets COM: Part 4

by Dave Jewell

By now, you should be feeling very relaxed about writing context menu handlers. Last time round, I said I'd mention some of the other types of COM-related shell extensions that are available. In the first part of this month's article, I'll do just that and then we'll go on to discuss dispatch interfaces (commonly called 'dispinterfaces') which form one of the foundation stones of OLE Automation.

A Shell Extension Roundup

Context menu handlers represent only one specific type of shell extension. Once you're familiar with the mechanics of writing one type of shell extension, you'll find that writing other types of extension isn't much different: it's really a matter of understanding what interfaces must be exposed by a particular type of extension, and what methods are significant.

I'll run through a few of the other types of shell extension which Windows Explorer supports.

Copy Hook Handlers

A copy hook handler is invoked by the shell whenever the user tries to copy, delete, rename or move a folder. Somewhat bizarrely, copy hook handlers are also invoked when a printer changes status, as when renaming a printer or modifying the port assigned to a printer. Using a copy hook handler, you can allow or disallow the operation from taking place. You can think of this as being somewhat like Delphi's `OnClose` event through which an application can allow or prevent a form from being closed.

As with all shell extensions, a copy hook handler needs to have appropriate entries in the system registry in order to work properly. For folders, copy hook handlers are registered under the following key:

```
HKEY_CLASSES_ROOT\directory\
  shellex\CopyHookHandlers
```

while for printers you'd replace `shellex` in the above registry path with `printers`.

Copy hook handlers are very easy to implement (even by Delphi's standards!) for two reasons. Firstly, they don't need to implement the `IShellExtInit` interface which (you'll remember) is used by context menu handlers to determine the file that's been right-clicked. Secondly, the only interface they need to implement, `ICopyHook`, has exactly one method called `CopyCallBack`. Note that when I say an interface has only one method, you should understand that this is in addition to the three standard methods required by `IUnknown`. The existence of these methods should always be taken as read. Since we're in the fortunate position of programming COM interfaces using Delphi, we often don't have to worry about these three interfaces at all. For example, see last month's context menu handler example.

Listing 1 shows what the `CopyCallBack` method looks like in Delphi terms.

```
function CopyCallback(Wnd: hWnd; wFunc, wFlags: UINT; pszSrcFile: PAnsiChar;
  dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD):
  UINT; stdcall;
```

➤ *Listing 1*

Briefly, the `Wnd` parameter is an API level Windows handle that should be used as a parent handle by any dialogs which need to be displayed by the extension. The `wFunc` parameter tells the copy hook handler what operation is being proposed (eg `fo_Delete` to delete the folder, `fo_Move` to rename the folder and so on). The `wFlags` parameter is a combination of informational flags that primarily give more information to the extension about the context of the operation. For example, is a progress dialog being displayed? Are individual file names being shown in the progress box? The shell extension can determine this information by looking at the `wFlags` value. As you'd expect, `pszSrcFile` and `dwSrcAttribs` contain pathname and file attribute information for the source path, whereas the remaining two parameters contain the same information for the destination path. Finally, the function result returned by `CopyCallBack` must be `IDYes`, `IDNo` or `IDCancel`, to indicate whether or not the operation may proceed. `IDCancel` aborts the entire operation whereas `IDNo` only gives the thumbs down to the current folder. Bear in mind that the user might potentially be dragging multiple folders in one go.

It's worth bearing in mind that multiple copy hook handlers can be installed. This means that when the user attempts a folder move, delete, rename or copy, the shell will attempt to call all registered copy hook handlers in order to determine whether or not an operation can proceed. Obviously, as soon as one handler has returned a value of `IDCancel` then the show is over: it's 'all in favour' or nothing. If you want to see an example of a copy hook handler written in Delphi, Borland include one in the Demos\ShellExt directory in a standard Delphi 3.0 installation.

Icon Handlers

Another type of shell extension is the icon handler. Using icon handler shell extensions, you can modify the icon that Windows Explorer uses to display files. At this point, you might be thinking 'big deal, I can do that by just changing a registry entry.' However, the icon handler has the flexibility to display files with different icons *even when the files are of the same type*.

To see how this works, you need to understand how the Explorer normally associates specific icons with different file types. As an example, take a look at `HKEY_CLASS_ROOT\DelphiProject` in the Windows Registry and you'll see that there's a sub-key called `DefaultIcon`. The value of this sub-key (at least on my system) is:

```
C:\Delphi 3.0\Bin\Delphi32.EXE,4
```

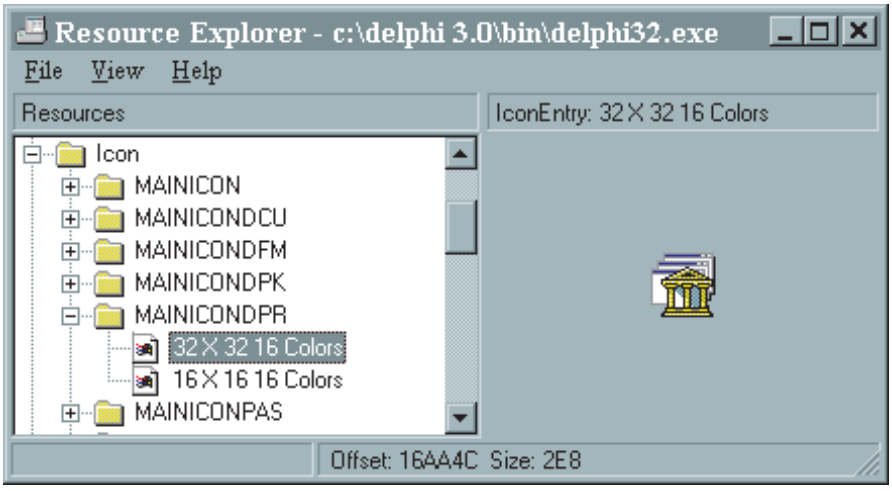
This information tells Explorer that the `DelphiProject` file type (which is defined elsewhere as a .DPR file) should be displayed using icon number 4 from the IDE's executable file. Sure enough, if you use a resource viewing tool to examine the icons in DELPHI32.EXE, you'll find that icon number 4 (counting from zero) is the icon for .DPR files (see Figure 1).

In order to change the Registry so as to use a custom icon handler, simply replace the value of the `DefaultIcon` key with the string `%1`. At the same time, to install the icon handler properly, you'd need to add a Registry entry of this form:

```
HKEY_CLASSES_ROOT\DelphiProject
  \shellex\IconHandler
```

The value of this entry will be a GUID which corresponds to the installed icon handler. Likewise, in the normal way, you'd need to add this GUID to the `CLSID` area of the registry with an `InProcServer` entry indicating where the icon handler DLL is located.

Icon handlers have to implement two different interfaces. The first interface, `IExtractIcon`, deals with the nitty-gritty of telling the Shell what icon to display. It implements two methods, `GetIconLocation` and `ExtractIcon`. Take care not to confuse the latter with a Windows API call of the same name, by the way. Which of these two methods you use depends on whether or not your icons are located inside a Windows executable or DLL. If they are, then you use `GetIconLocation` to tell the Shell where the icon is. If not, then you use `ExtractIcon` to give an API-level icon handle directly back to the Shell. Listing 2 shows the Object Pascal declaration for the `GetIconLocation` method.



➤ *Figure 1: In the system registry, the DefaultIcon entry associated with a file type is used to specify the name of an executable file and the index of the desired icon. It's possible to change this value to '%1' and associate an icon handler with the file type, displaying different icons on a per-file basis.*

```
function GetIconLocation (uFlags: UINT; szIconFile: PAnsiChar; cchMax: UINT:
  out piIndex: Integer; out pwFlags: UINT): hResult; stdcall;
```

➤ *Listing 2*

Are you wondering what those interesting-looking `out` keywords are? This is something that I haven't covered yet, but we'll be examining in detail later. For now, just think of an `out` parameter as being similar to a `var` parameter except that it only works one way. In this case, the two `out` parameters are results provided by the `GetIconLocation` method to the routine that called it.

The first parameter, `uFlags`, tells the shell extension whether the requested icon is for display in a shell folder (`gil_ForShell`) or for displaying the icon of an open folder (`gil_OpenIcon`). In return, the shell extension fills the buffer pointed to by `szIconFile` with the full pathname of a .EXE file or DLL which contains the icon. The `cchMax` field is used to indicate the maximum size of this buffer. In addition to this, the `piIndex` parameter is used to tell the shell which icon in the file should be used while the `pwFlags` parameter passes additional flags to the shell. Amongst other things, these flags indicate whether or not all files of the same type should use the same icon, or whether each file can have its own icon.

But wait, there seems to be something vital missing from this interface. How does the icon handler know what file is being referenced? Well, you'll remember I said that an icon handler must implement two interfaces. The first, `IExtractIcon`, is what we've been discussing so far. The other interface is `IPersistFile`. As the name suggests, `IPersistFile` relates to the storing of persistent data in disk files. The bad news is that this interface implements quite a number of different methods. The good news is that only one of the methods, `Load`, is of any concern to an icon handler, the rest can all be stubbed out.

Property Sheet Handlers

One of the most interesting types of shell extension is the property sheet handler. My computer has a 100Mb ZIP drive (highly recommended!) which I use for backup and off-lining data that I won't be using for a while. In Figure 2, you can see how the IOMega software makes good use of property sheet handlers to seamlessly add ZIP

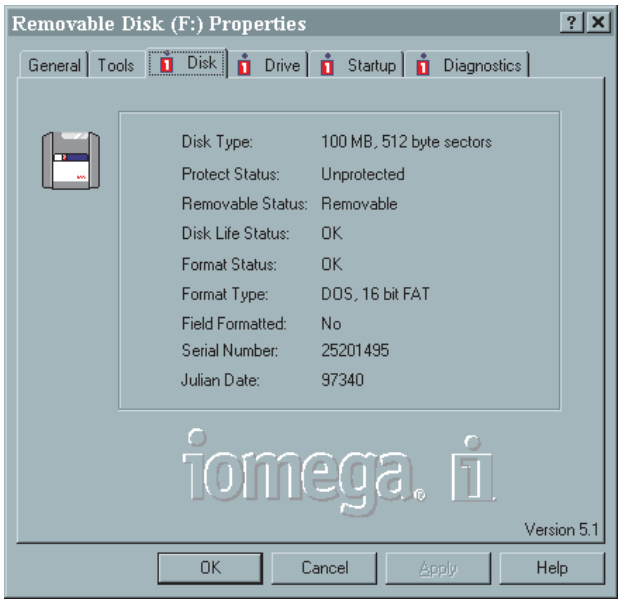


Figure 2: It's not difficult to think of good uses for the property sheet handler type of Shell extension. Here, IOMega have added no less than four new pages to this property sheet dialog.

drive specific property pages to the Explorer, all the pages marked with a red tab are custom property pages.

The property sheet handler interface allows you to add custom property pages not only to volumes, but also to folders and different file types. To create your own property sheet handler, you need to create a COM object which implements the `IShellExtInit` and `IShellPropSheetExt` interfaces like this:

```
TPropPageHandler =
  class(TComObject,
  IShellExtInit,
  IShellPropSheetExt)
```

As with other shell extensions we've seen, the `IShellExtInit` handler is used to pass the name of the file to the shell extension while the `IShellPropSheetExt` interface deals with the specifics of property sheet management.

Time for a *mini-rant:* why did Microsoft do things in this kack-handed way? I accept that the Explorer needs a consistent interface for passing filename information to an extension, but it would have made a lot more sense to define an abstract `IShellExtension` interface with one or more methods for filename acquisition. Shell extension interfaces such as `IShellPropSheetExt` would then derive from this common ancestor, adding any extension specific methods that are required. As we've seen, implementing multiple interfaces in a COM object is easy with Delphi 3.0, but for C++ developers a lot of extra aggregation glue-code is needed. They haven't even been consistent: Icon handlers need to implement `IPersistFile` just to get the file name! OK, I feel better now.

The `IShellPropSheetExt` interface defines two methods, `ReplacePage` and `AddPages`. The former is never used by property sheet handlers (it's actually used by Control Panel extensions, allowing standard controls panel pages to be replaced by custom ones) and it's sufficient to return a value of `E_NotImpl` to indicate that it's not implemented. More interesting is the `AddPages` method which is defined like this:

```
function AddPages(lpfnAddPage:
  TFNAddPropSheetPage;
  lParam: LPARAM): hResult
  stdcall;
```

As the name suggests, this method can be used to add multiple pages to a property sheet, as in my example of the ZIP drive Shell extensions. The first parameter is of type `TFNAddPropSheetPage` and is defined in the COMMCTRL.PAS file:

```
TFNAddPropSheetPage = function(
  hPSP: HPropSheetPage;
  lParam: Longint): BOOL
  stdcall;
```

In a nutshell, the `AddPages` method has to create one or more property pages using the `CreatePropertySheetPage` API routine. For each property sheet that it's created, it calls the `lpfnAddPage` hook supplied by the Shell. This adds the property sheet to the list of internal property sheets maintained inside the Shell. When calling through the `lpfnAddPage` hook, it's vitally important to set the second parameter of the call to the `lParam` value which was supplied to `AddPages` itself. Amongst other things, this helps the Shell to keep track of which property pages are associated with which property sheet handler, bearing in mind that you can have multiple property sheet handlers registered for a single file type. It's also important that if the call through the `lpfnAddPage` hook fails (indicated by a `False` function result) then `AddPages` code should destroy the property sheet and exit.

In theory, creating property sheet handlers should be nice and easy with Delphi. In practice, I've left it as an exercise for the reader! The big problem here is the way in which Microsoft have implemented property sheets themselves. Effectively, each property sheet page is its own little dialog, complete with dialog message handler and dialog template resource to define the number, size, position and type of the various dialog controls on the page. In other words, if you want to have a serious crack at writing property sheet handlers with Delphi, you'll have to abandon the cosy RAD world of forms and VCL components and journey into the relatively primitive jungle of barefoot API-level dialog implementation. One possible way around this would be to just put a push-button on the property sheet, and point that to some code which fires up a native DLL dialog. However, this obviously isn't an ideal solution. A much better way would be to programmatically transplant one or more VCL components onto a property page. I imagine that a perusal of the VCL source code relating to `TTabSheet` and `TPageControl` would

provide some useful ideas, maybe I'll take a look at this issue in a future article. For those adventurous souls who want to take things further, I've included the file PROPERTYSHEET.ZIP on this month's disk. This contains source for a public-domain property sheet handler written in Delphi. Do bear in mind that I haven't tested this code myself and it was also written for Delphi 2.0, meaning that you can make considerable simplifications to the PROPERTIES.PAS file if you're working with Delphi 3.0.

Introducing The Dispinterface

Microsoft soon realised that, deeply wonderful though COM was, there were shortcomings with the plain-vanilla COM interface that we've been discussing up to this point. These shortcomings were particularly significant when it came to using a simple programming language such as Visual Basic.

Back in those early days, the Microsoft developers were faced with the prospect of writing one set of wrapping code to enable Visual Basic to use one COM interface, another set of wrapping code to get it to work with another interface, and so on. They obviously didn't want to spend the rest of their lives writing wrapping code for all the different COM interfaces and adding it to the Visual Basic runtime library! Neither did users of Visual Basic want to have to keep waiting for a new release which implemented the COM interfaces they were interested in using.

What was needed was a way for Visual Basic to work with a COM interface that it hadn't previously encountered. If you think back to last month's simple OLE Automation example, you'll realise that this capability is an absolute necessity. Neither Delphi's developers nor the creators of Visual Basic know what Automation calls you're going to make from your program, nor what Automation servers you're going to communicate with, but everything has to work as advertised.

The solution to this problem was the Dispatch Interface, or for short, dispinterface. The dispinterface lies at the heart of OLE Automation and ActiveX technology and is based around an interface called `IDispatch`.

```
IDispatch = interface(IUnknown)
  ['{00020400-0000-0000-C000-000000000046}']
  function GetTypeInfoCount (out Count: Integer): Integer; stdcall;
  function GetTypeInfo (Index, LocaleID: Integer; out TypeInfo): Integer;
    stdcall;
  function GetIDsOfNames (const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DispIDs: Pointer): Integer; stdcall;
  function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
    Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): Integer;
    stdcall;
end;
```

➤ *Listing 3*

Listing 3 might look a bit overwhelming at first sight, and probably at second sight too! For starters, `IDispatch` is based around the same `IUnknown` interface that we all know and love; it has the same three standard `IUnknown` methods which all COM objects support. The key method here is `Invoke`. It's this method which forms a 'back--door' into the many other methods which the interface might support. Up until now, all the methods on a particular interface have been explicitly declared: with `IDispatch`, the above interface (or rather, another interface derived from it) might potentially support dozens of different methods, all of which are available through this `Invoke` back-door.

Confused? Think of it like this: by writing Visual Basic in such a way that it could understand and talk to an `IDispatch` interface, they immediately got the benefit of having VB able to talk to any dispinterface that anyone might write. In order to call one of these hidden methods, you need to have its ID, or `DispID`, as they're usually called.

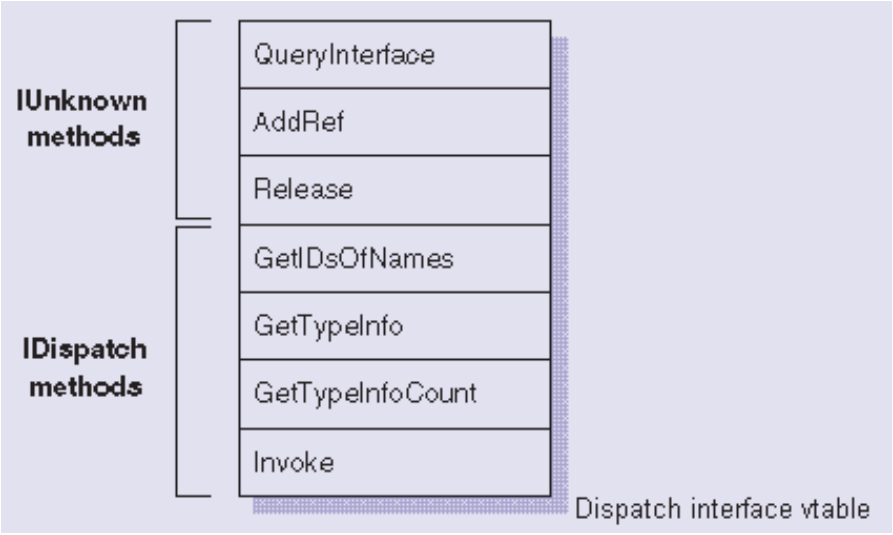
Just as `Invoke` represents the back-door through which we can call one of many different interface methods, so the `GetIDsOfNames` method represents a kind of 'telephone directory' which enables a caller to get an ID for each and every method that's supported by the interface.

So, imagine you want to call a method called `GetDirCount`. First, you call `GetIDsOfNames` to determine the dispatch ID for the `GetDirCount` method. Then, you would call the `Invoke` method to actually make the call. Get it? Telephone directory? Make the call? Oh, never mind...

It's natural to ask why you can't directly specify the required method name as a parameter to `Invoke`. The reason, of course, is one of speed and efficiency. The last thing the COM code needs to be doing is string comparisons each and every time a method is called. Using `GetIDsOfNames`, you can retrieve a `DispID` once and then call that method many times. In fact, the `GetIDsOfNames` method is written in such a way that you can use it to retrieve multiple `DispIDs` for multiple methods all in one go.

If you look at Figure 3 you'll see the layout of the `vtable` for the `IDispatch` interface. Including the three `IUnknown` methods, there are a total of seven methods and that doesn't change. Thus, it's relatively easy to write the runtime library code to interface the Visual Basic interpreter to an `IDispatch` interface. `IDispatch` is written in such a way that the `Invoke` call can accept a variable number of arguments, the `Params` parameter is essentially a pointer to a data structure which contains a variable number of arguments.

If you read my previous *Beating the System* columns on Delphi code generation (Issues 16 and 17, December 1996 and January 1997), you'll know that the Delphi compiler uses a very similar technique to implement routines such as `Format` which appear to take a variable number of parameters. At the grass roots level, what gets passed to the `Format` routine is a pointer to a data structure which is built at run-time immediately before the call. This data structure comprises a variable number of entries, each of which is tagged with a byte which indicates the type of the



➤ *Figure 3: This diagram (from the online help) shows the layout of a vtable when using the IDispatch interface. No matter how may methods hang off this interface, there will only be seven vtable entries, making it easy to code for this interface. However, see next month's explanation of dual interfaces for a scheme which gives the best of both worlds...*

entry such as long integer, string, or whatever. In practice, the parameter-passing technique used by `IDispatch` is more complex than the `Format` system because OLE Automation supports both named and optional arguments.

While on the subject of parameter passing, it's worth pointing out that by using `IDispatch`, you get the side benefit of automatic parameter marshalling.

Marshalling is something that we haven't discussed up to now. Put simply, marshalling is the process whereby COM transfers arguments between the processes on either side of a COM interface. In the simplest possible case, a COM server will be an in-process server, ie a DLL. The shell extensions we've been discussing up to now are in-process servers called by the Windows Explorer. Because an in-process server lives in the address space of its client, little or no marshalling is required in these circumstances because both client and server can directly access the same memory.

In the case of an out of process server, such as an OLE Automation server, things are more complex. Effectively, you've got one application talking to another and, as you know, Windows does its best to put a brick wall between different processes *[and some application software I use does its best to batter down that wall...! Ed]*. In the case of

Windows NT, this wall is several feet thick, electrified and has broken glass along the top! This is where marshalling comes into its own. The marshalling code has to take the client arguments, package them up and copy them into the address space of the server. After the actual target method has been called, any results of the call then have to be packaged up and copied back into the address space of the client. As you'd expect, similar things have to happen with DCOM (Distributed COM) except that now, the marshalling takes place between different machines on a network.

Delphi Support For Dispatch Interfaces

Microsoft are not, forgive me for being blunt, noted for their ability to create straightforward, easy to use programming interfaces. If you compare a Delphi application with the same thing written in C/C++ to use the Windows API, you'll probably laugh your socks off wondering why anybody should want to program that way.

Likewise, even an MFC-based application (MFC is Microsoft's relatively crude, rather lower level, C++ equivalent of the VCL library) is both larger in source terms and far more hieroglyphic than its Delphi equivalent.

But Microsoft *really* went to town with OLE. Look back at the `Invoke` call we saw earlier, eight parameters needed for each and every call to `Invoke`! Fortunately, this series is all about COM programming from a Delphi perspective and you should be really thankful for that! Borland have put a lot of work into making COM programming much easier than it would otherwise be and we're going to finish off this month's instalment by looking at some of the ways in which `IDispatch` 'awareness' has been built into the compiler.

For starters Delphi allows you to perform compile time binding rather than late binding on `IDispatch` methods. Let me explain; earlier I said that in order to call a method on a dispinterface, the controller (in Automation speak, the client is generally referred to as the controller) first had to get its `DispID`. This is called late binding or runtime binding, because the client doesn't bind to a particular method or property until run-time. Although this is very flexible, it isn't as efficient as it could be because of the overhead of calling `GetIDsOfNames` to obtain the necessary `DispIDs`. In many cases, Delphi eliminates this overhead by allowing you to bind to methods at compile time. The code in Listing 4 shows how it works.

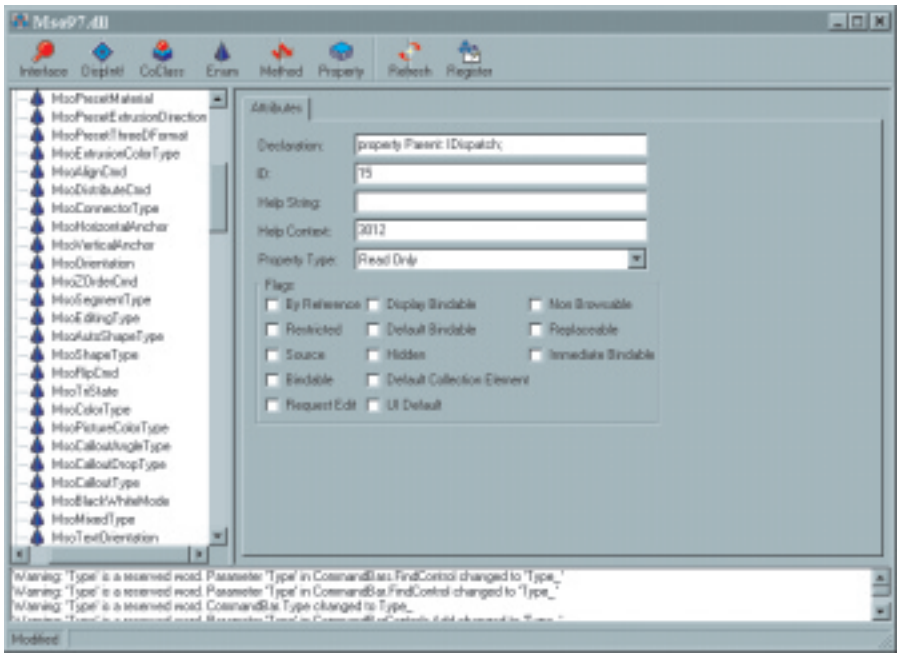
```
IAmbientDispatch = dispinterface
  ['{00020400-0000-0000-C000-000000000046}']
  property BackColor: Integer dispid DISPID_AMBIENT_BACKCOLOR;
  property DisplayName: WideString dispid DISPID_AMBIENT_DISPLAYNAME;
  property Font: IFontDisp dispid DISPID_AMBIENT_FONT;
  property ForeColor: Integer dispid DISPID_AMBIENT_FORECOLOR;
  property LocaleID: Integer dispid DISPID_AMBIENT_LOCALEID;
  property MessageReflect: WordBool dispid DISPID_AMBIENT_MESSAGEREFLECT;
  property ScaleUnits: WideString dispid DISPID_AMBIENT_SCALEUNITS;
  property TextAlign: Smallint dispid DISPID_AMBIENT_TEXTALIGN;
  property UserMode: WordBool dispid DISPID_AMBIENT_USERMODE;
  property UIDead: WordBool dispid DISPID_AMBIENT_UIDEAD;
  property ShowGrabHandles: WordBool dispid DISPID_AMBIENT_SHOWGRABHANDLES;
  property ShowHatching: WordBool dispid DISPID_AMBIENT_SHOWHATCHING;
  property DisplayAsDefault: WordBool dispid DISPID_AMBIENT_DISPLAYASDEFAULT;
  property SupportsMnemonics: WordBool dispid DISPID_AMBIENT_SUPPORTSMNEMONICS;
  property AutoClip: WordBool dispid DISPID_AMBIENT_AUTOCLIP;
end;
```

➤ *Listing 4*

This interface, `IAmbientDispatch`, is one of the dispinterfaces that are used in the implementation of ActiveX controls. As you can see, it implements a set of properties, some of which will have names that are familiar to you.

What might not be so familiar is the reserved keyword `dispinterface` at the beginning of the declaration and the `dispid` keyword associated with each property. The identifiers following each of these keywords are simply numeric constants. For example, the identifier `DISPID_AMBIENT_BACKCOLOR` has a value of `-701`. By associating the names of known `DispID` values with each property or method in this way, it becomes unnecessary to call the `GetIDsOfNames` routine at runtime, the compiler can simply generate code which references a specific `DispID` directly. This being the case, it

➤ *Figure 4: If you use the type library editor in Delphi 3.0 to browse the type libraries of existing files then be careful, because it has a habit of modifying the existing type library: see the warning messages at the bottom of the window.*



allows you to call dispinterface methods more quickly than you could otherwise. Borland refer to this technique as ID binding because, effectively, you're binding your code to specific ID numbers at compile time.

This raises the question of where these `DispIDs` come from. Is it necessary to laboriously call `GetIdsOfNames` on an interface in order to determine all the `DispID` numbers to plug into a declaration like that above? Thank goodness, the answer is no. It's at this point that we need to talk about type libraries.

A type library is essentially a chunk of binary data which encapsulates all the details of interfaces, methods, arguments and so on that are associated with a particular COM server. Delphi can read type libraries and automatically generate a Pascal definition of the associated interfaces, complete with those all-important `DispID` numbers. Type libraries can exist as 'stand-alone' files, or be compiled into the associated COM server as a resource. If you use your favourite resource editor to open up a .OCX control (for example) you'll see it contains a resource of type `TYPELIB`. This contains all the type information associated with the control.

Old fashioned development tools require you to use a somewhat baroque interface description language (IDL) to create an interface definition. This is then run through a special type of compiler which spits a type library out the other end. Happily, we don't need to bother with any of that nonsense: Delphi has a built in type library editor which can open and edit type libraries directly, whether they be stand-alone or included into COM server executables. If you want to try out the Delphi type library editor for yourself, select `File | Open` from the Delphi IDE, specify `Type Library` from the drop-down file filter combo-box and then find yourself a likely looking OCX control.

One of my favourites is the MSO97.DLL which comes with MS Office97, but be sure to say "No" when it asks you if you want to save any changes you've made!

Having established that Delphi is the bee's knees as far as COM programming is concerned, I need to inject some balance by pointing out that there are some limitations to dispinterfaces. The major limitation is on the allowable types of parameters that can be passed across a dispinterface. According to the Borland documentation, there are 13 basic types that can be passed across a dispinterface.

This, incidentally, is the primary reason why Borland's so-called "One-Step" ActiveX technology generally involves a few more steps than you might have expected. In other words, you can't just take any old VCL control, auto-magically turn it into an ActiveX control and expect it to wind up with exactly the same set of properties, events and methods that it had inside the Delphi IDE. For example, any properties, methods and events which involve user-defined types won't make it through the One-Step process. For a fuller discussion of this topic, check out the chapter on ActiveX control conversion in Ray Konopka's new *Delphi 3 Component Programming* book.

Conclusions

Last month you got some working code to play with, this month was mostly theory. Well, you have to take the rough with the smooth! Next month, I'll be going into more detail about how Borland have made life easier for you by encapsulating a lot of OLE Automation grunt code into easy to use classes. We'll also be rolling up our sleeves and examining how to use build OLE Automation capabilities into your own programs.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at Dave@HexManiac.com.