Delphi Meets COM: Part 3

by Dave Jewell

We've spent the last couple of months looking at COM fundamentals in some detail. As promised, this month we'll set the theory aside for a while, roll up our sleeves and look at a couple of practical examples of COM programming. This time round, we'll be majoring on how to develop context menu handlers using Delphi.

An OLE Automation Taster...

But wait! Before you fire up your Delphi development system, there's just one last teensy weensy bit of theory which we need to cover (trust me, I'm a doctor!). Up until now, I've talked about COM in general terms without mentioning how the Windows registry fits into the picture. As you'll undoubtedly know, the registry stores all sorts of important information about your Windows setup, the configuration settings for all your favourite programs and much more.

The registry is also absolutely vital to the operation of COM. Without the registry, the COM system wouldn't work. In fact, it was principally the development of COM (see the first article in this series) which was responsible for the forsaking of text-based .INI files, and the move towards the more efficient, hierarchical, one-stop registry system.

To see how vital the registry is, let's take a brief, appetite-whetting, look at *OLE Automation*. Later in this COM series, we'll be covering Automation in detail. If you've not heard the term before, suffice to say that it's a powerful technique which (put very simply!) allows an application to be operated 'by remote control' from another program.

Probably the simplest way of retrieving an OLE automation object is to make a call to the easy-peasy `CreateOLEObject` function. This routine is provided by Delphi in the `COMOBJ` unit, and the declaration for it is:

```
function CreateOleObject(
  const ClassName: string):
  IDispatch;
```

You'll remember that GUIDs which are used to identify classes are called CLSIDs? OLE Automation simplifies things even further by introducing the concept of the PROGID which is simply a human-readable class identifier. With OLE Automation, we can do something like that shown in Listing 1.

This example, (condensed from one of Borland's sample programs) shows how easy it is to interact with Microsoft Word using OLE Automation. Firstly, the `CreateOLEObject` routine is called with the PROGID `Word.Basic`, thus retrieving an automation object which can be used to interact with WordBasic. Next, the code queries Word to determine the current language version, and finally three WordBasic methods are called to make Word visible, create a new file and write the current language version string into the file.

Simple as this little demo is, it should raise a number of important questions in your mind. How, for example, does the underlying COM/OLE code know that the PROGID `Word.Basic` is associated with Microsoft Word? How does the OLE manager convert the PROGID string into the all-important GUID needed to reference the target class? How does the operating system know where Microsoft Word is located, and how does it know how to invoke Word in response to an OLE Automation request? The answer to all these questions can be summed up in just two words: *the registry*.

Enter The Registry

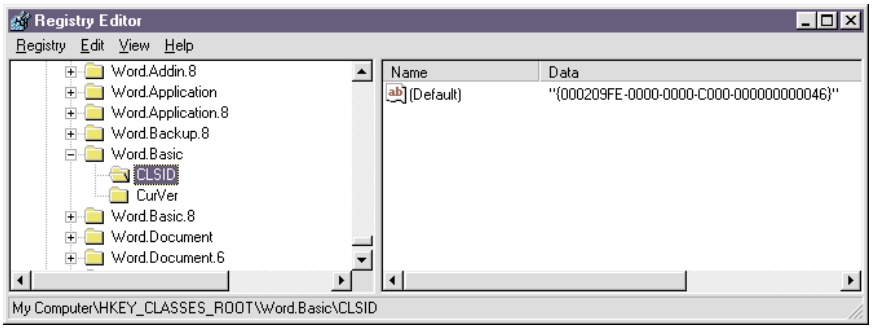
Let's try and answer those various questions in more detail. If you run the `RegEdit` utility and then open up the `HKEY_CLASSES_ROOT` tree, you'll find that, sure enough, there's a registry key named `Word.Basic` in this tree. If you open up this registry entry, you will then see the corresponding GUID (see Figure 1).

Likewise, if you examine the source code for the `CreateOLEObject` routine (inside the `COMOBJ` unit), you'll find that it calls `ProgIDToClassID`.

As the name suggests, this new routine (which is also a part of the `COMOBJ` unit) has the job of converting a human-readable PROGID into the equivalent CLSID. The `ProgIDtoClassID` routine, in turn, is just a wrapper around a lower level routine, `CLSIDFromProgID` which is part of Microsoft's internal COM code. We haven't got access to the source code for this routine but,

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Lang: string;
  MSWord: Variant;
begin
  try
    MsWord := CreateOleObject('Word.Basic');
  except
    ShowMessage('Could not start Microsoft Word.');
    Exit;
  end;
  try
    Lang := MsWord.AppInfo(Integer(16));
  except
    ShowMessage('Couldn''t get Word language version');
  end;
  MsWord.AppShow;
  MSWord.FileNew;
  MSWord.Insert(Lang);
end;
```



➤ *Figure 1: The HKEY_CLASSES_ROOT part of the system registry allows Windows to map PROGID's onto GUID's and also map GUIDs onto PROGIDs. Here we can see the entry for Word.Basic.*

clearly, `CLSIDFromProgID` is simply examining the `HKEY_CLASSES_ROOT` tree for the required PROGID entry and returning the associated GUID.

Incidentally, if you want to go the other way (ie convert a GUID into the corresponding PROGID), you can call the `ClassIDToProgID` function which wends its way down onto another Microsoft routine, `ProgIDFromCLSID`. This reverse mapping is equally straightforward because a few more moments with `RegEdit` will show you that every registered CLSID can be found as a distinct key in the sub-tree, and Automation entries within that tree contain a sub-key, `ProgID`, which contains the human-readable class identifier.

So far so good: we can see how to map both ways between a PROGID and a GUID, so we've answered at least the second of our three questions. To answer the other two questions, just look more carefully at the entries in the aforementioned `HKEY_CLASSES_ROOT\CLSID` sub-tree. You'll see that, in the case of in-process COM servers (such as ActiveX controls and shell context menu extensions) there's a sub-key called `InProcServer32`. This sub-key contains the full pathname of the server which implements the designated COM object. Simple!

Context Menus: As Viewed From The Shell

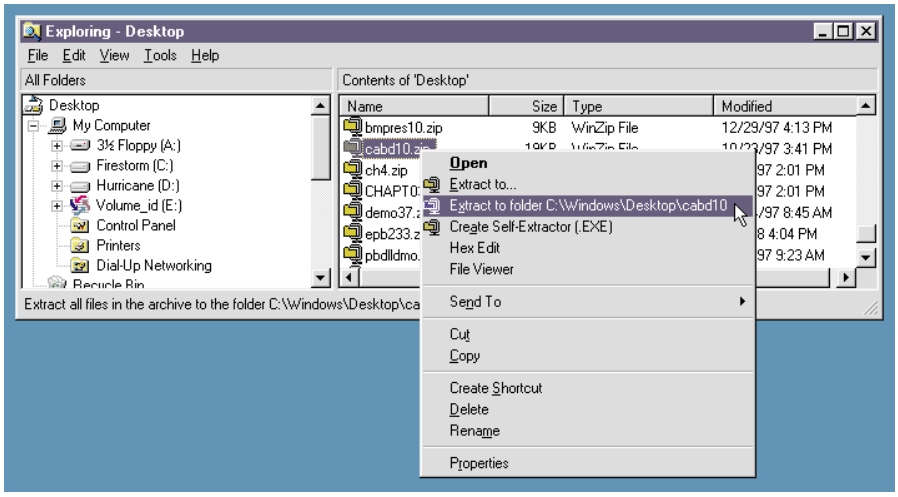
We could continue to talk in abstract terms about how the registry stores information on available COM objects, but instead let's take a real-world example of how the Windows 95 Explorer shell integrates with context menu handlers. In Figure 2, you can see a dropped-down Explorer context menu. As you know, the Explorer program drops down a context menu whenever one or more items are selected and you right-click on them using the mouse. The context menu is so called because its contents are determined by the context, ie by the specific type of file that's been selected.

In Figure 2, you can see a number of context menu entries that you might not normally be familiar with. The topmost menu item, `Open`, is displayed whenever Explorer has a file association for the selected item(s). Because a ZIP file is selected, and because WinZip (Nico Mak Computing Inc.) is installed on my system, a file association exists and therefore an `Open` item is shown. The `Extract to...`, `Extract to folder` and `Create Self-Extractor` menu items are all provided courtesy of WinZip and its sidekick, the WinZip self-extractor program, which is capable of converting ZIP files into self-extracting EXE files. In addition to these items, the `Hex Edit` entry is provided by a shareware hex editor program called Hex Workshop and `File Viewer` is part of the shell enhancements provided by the Delphi add-in Merlin *[which is reviewed in the February issue of Developers Review, plug, plug. Ed]*.

All these different programs implement a context-menu handler as an in-process server (that is, as a DLL) which can be invoked from the Explorer. How do these different add-ons integrate into the Explorer without treading on each other's toes, and how does Explorer know how to activate them at the appropriate time? For a blow-by-blow explanation of how this works, read on.

As mentioned above, Explorer fills in the various context menu entries according to what file types have been selected. For the sake of argument, let's assume that you've selected a single ZIP file and right-clicked the mouse. When you first right click the mouse, Explorer looks at the name of the selected file and extracts the file extension along with the preceding period: `.ZIP` in this instance. It then tries to find this string as a sub-key in the registry under the `HKEY_CLASSES_ROOT` branch of the

➤ *Figure 2: On my system, a context menu has no less than six extra custom entries when a ZIP file is selected. Explorer builds the context menu 'on the fly' by calling all context menu handlers that are registered for the selected file type.*



tree. So, in this case, Explorer is looking for an entry called HKEY_CLASSES_ROOT\ZIP.

If no such entry is found, then there's no special file association. However, on my system, there is indeed a registry key with this name. The default value of the key is WinZip which tells Explorer what application is associated with .ZIP files. You need to realise that this isn't necessarily a human-readable application name. It's simply the name of another key in the registry. If a file association is found, then Explorer adds the Open item to the context menu.

Incidentally, Explorer uses this same key (WinZip in this example) to obtain the human readable description of a particular file type. It does this by looking for a key called HKEY_CLASSES_ROOT\WinZip. If found, the Default value of the key gives the description string that Explorer displays next to the file. With WinZip installed, this is WinZip File.

You also need to bear in mind that, for performance reasons, this registry lookup isn't necessarily happening all the time: I imagine that Explorer maintains an internal, in-memory cache of file description strings and context-menu associations. I'm explaining things in a sequential manner for the sake of simplicity.

Once the WinZip application key has been obtained, Explorer then needs to see if there's a context menu handler associated with the application. It therefore looks for a key at the following location:

```
\HKEY_CLASSES_ROOT\WinZip\shellex\
  ContextMenuHandlers
```

In fact, WinZip doesn't have a key here. That's because, rather than registering a context menu handler for ZIP files only, WinZip registers a context menu handler for *any* file type. This is important because it means (amongst other things) that WinZip can display a context menu entry Add to Zip when the item clicked on is not a ZIP file. In order to look for wildcard context menu handlers, the Explorer also looks for registry keys of the form:

```
HKEY_CLASSES_ROOT\*\shellex\
  ContextMenuHandlers\?????????
```

You'll notice the asterisk * which indicates that any context menu handlers found in this sub-tree are to be applied to all file types. WinZip keeps a registry key here that looks like this:

```
HKEY_CLASSES_ROOT\*\shellex\
  ContextMenuHandlers\WinZip
```

The Default value for the key is:

```
{E0d79300-84be-11ce-9641-
  444553540000}
```

Now that the GUID of WinZip's context menu handler is known, the server DLL can be located by examining the corresponding key in the HKEY_CLASSES_ROOT\CLSID sub-tree as previously described. The full registry key of what we're after is shown in Listing 2.

The Default value of this key (on my system, your mileage may vary) is C:\WINZIP\wzshlext.dll which is the full pathname of the in-process server DLL that implements the context menu hander associated with ZIP files. I should emphasise again that neither WinZip nor any other COM object needs to know the physical location of the COM object server. Once the CLSID has been obtained, then things are relatively straightforward from the viewpoint of the COM client.

Context Menu Registration

From the foregoing, it should be very obvious that when you supply a context menu handler for the shell, you also have to somehow register the context menu in the registry on the user's machine. If you take a look at Borland's example context menu handler (\DEMOS\SHELLEXT) you'll find a file called Contmenu.reg which contains the code in Listing 3.

This is a registration script which defines the registry entries needed by the context menu handler. Explorer understands that .REG files are registration scripts and will merge their contents into the system registry either by double-clicking the file or else by right-clicking and selecting Merge from the context menu that results.

In this particular case I strongly recommend that you *don't* merge these registry entries into your system! Borland made a basic error when they wrote the above .REG file, and it should be pretty obvious what it is. In case you haven't spotted it, there's no path to the DLL, it's just contmenu.dll with no indication of where the file is located. Consequently, when Explorer attempts to load the DLL, Windows will search all the standard DLL locations and then report an error, causing Explorer to assume that the context menu handler no longer exists.

Personally, I reckon that supplying .REG files to users is a bad idea anyway. You don't want end users to have to muck about double-clicking REG files and if you *do* use REG files, you've also got to modify the file (as part of the installation procedure) so that the pathname of the server DLL agrees with the directory where the user has

```
HKEY_CLASSES_ROOT\CLSID\{e0d79300-84be-11ce-9641-444553540000}\InProcServer32
```

➤ *Listing 2*

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{8e3e0f0a-0fcc-11ce-bcb0-b3fd0e25381a}]
@= "Delphi 3.0 Context Menu Shell Extension"
[HKEY_CLASSES_ROOT\CLSID\{8e3e0f0a-0fcc-11ce-bcb0-b3fd0e25381a}\InProcServer32]
@= "contmenu.dll"
"ThreadingModel" = "Apartment"
[HKEY_CLASSES_ROOT\DelphiProject\shellex\ContextMenuHandlers\{8e3e0f0a-0fcc-11ce-
bcb0-b3fd0e25381a}]
@= ""
```

➤ *Listing 3*

chosen to locate your product. This is an error prone exercise. A better idea is to write some code which directly plugs the necessary values into the registry during installation. Depending on the capabilities of your installer, this registry configuration code could be part of the install script itself, or else located in a seperate DLL or program called by the installer.

As an example of how to do this, I've provided source code for a simple, non-visual component in Listing 4. This component has one aim in life, the registration of context menus. It takes four properties; the COM class GUID (expressed as a string in curly brackets), a description string, the full pathname of the associated server DLL and another string which represents the type of file associated with the context menu handler. An example of how to use the component is given in Listing 5. Bear in mind that if you use `Ctrl-Shift-G` within the Delphi IDE to create new GUIDs, you must remove the outermost square brackets before assigning to the `GUID` property of the component. The code for this component and the test program are contained in the file REGCONTEXT.ZIP on this month's disk.

```
unit RegContextMenu;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComObj;
type
  TRegContextMenu = class(TComponent)
  private
    fGUID: String;
    fDesc: String;
    fPathName: String;
    fFileType: String;
  protected
  public
    procedure Write;
  published
    property GUID: String read fGUID write fGUID;
    property Description: String read fDesc write fDesc;
    property PathName: String read fPathName write fPathName;
    property FileType: String read fFileType write fFileType;
  end;
procedure Register;
implementation
resourcestring
  sBadGUID     =      'GUID string not set';
  sBadDesc     =      'Description string not set';
  sBadPath     =      'Pathname string not set';
  sBadFileType =      'FileType string not set';
procedure TRegContextMenu.Write;
begin
  { Make sure everything has been set }
  if fGUID = '' then raise Exception.Create (sBadGUID);
  if fDesc = '' then raise Exception.Create (sBadDesc);
  if fPathName = '' then raise Exception.Create (sBadPath);
  if fFileType = '' then raise Exception.Create (sBadFileType);
  { CreateRegKey will raise EOleError on failure }
  CreateRegKey ('CLSID\' + fGUID, '', Description);
  CreateRegKey ('CLSID\' + fGUID + '\InProcServer32', '', fPathName);
  CreateRegKey ('CLSID\' + fGUID + '\InProcServer32', 'ThreadingModel',
    'Apartment');
  CreateRegKey (fFileType + '\shellex\ContextMenuHandlers\' + fGUID, '', '');
end;
procedure Register;
begin
  RegisterComponents ('COM', [TRegContextMenu]);
end;
end.
```

➤ *Listing 4*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with TRegContextMenu.Create (Self) do try
    GUID := '{8e3e0f0a-0fcc-11ce-bcb0-b3fd0e25381a}';
    Description := 'Delphi 3.0 Context Menu Shell Extension';
    PathName := 'c:\Delphi 3.0\demos\shellext\contmenu.dll';
    FileType := 'DelphiProject';
    Write;
  finally
    Free;
  end;
end;
```

➤ *Listing 5*

Real World Context Menus

With all the above in mind, we're now in a good position to start writing our own context menu handlers. Context menus are great because they server as a gentle introduction to practical COM programming without being too complex. Here's one important tip though: if you don't plan to keep the context menu handlers that you develop, you should make a note of the different GUIDs you use so that, later, you can go back through the registry and delete the registry entries no longer used.

Most of us Delphi programmers have huge numbers of Pascal files scattered all over our hard disks, I know I do! Often, it's convenient being able to view a Pascal file from the Windows shell without having to fire up Delphi itself. By the time Delphi 3 has loaded itself into memory, you could have taken a look at four or five different Pascal files using Notepad. Accordingly, this next context menu example shows how to change the file association for .PAS files, while at the same time associating a `View Source` context menu entry with .PAS files. Of course, this means that double-clicking a .PAS file will no longer launch Delphi, but this isn't something I tend to do much anyway. I prefer to explicitly launch Delphi by double clicking a Delphi project file.

Figure 3 shows the basic idea. Whenever a .PAS file is selected, the `View Source` item will appear in the context menu and clicking this item will fire up the Notepad program with the selected file ready loaded. The source code for the context menu handler is given in Listing 6. If you compare this code to Borland's original context menu demo program, you will find that I've added some comments to make it clearer what each method does.

The real heart of the code is the call to `WinExec` inside the `TContextMenu.InvokeCommand` method. Also, the badly named `TContextMenu.GetCommandString` method provides

a hint string which is displayed at the bottom of the Explorer window when the custom menu item is selected but not yet clicked. The corresponding .REG file which adds the necessary registry entries is shown in Listing 7. As always, adding these entries programmatically is left as an exercise for the reader!

You might be thinking that this effect could be achieved with virtually no effort simply by tweaking the registry such that .PAS files are treated the same way as .TXT files. In other words, just go to the `HKEY_CLASSES_ROOT\.txt` entry and set the `{Default}` value to `txtfile`. Well, that would certainly work, but it rather misses the point. This context menu example is intended to serve as a simple demonstration which you can then extend any way you like. For example, you could add several different custom menu entries such as `Compile` to run the designated .PAS through the DCC command-line compiler, `Browse` to fire up your all-singin', all-dancin' Pascal browser program, `Beautify` to run a Pascal source beautifier and so forth. The world is your COM-based oyster! Source to this example is included on the cover disk as PASMENU.ZIP. Be sure to change the pathname to the DLL, as appropriate, before running the registry script!

Sailing Close To The Wind! Or, Are You Listening, Delphi?

As my final context menu example, I want to address a problem that irritates many Delphi programmers. As you'll know, .PAS files are associated, by default, with the Delphi IDE. If you're browsing through your source code and you want to do some work on a project, double-clicking a .PAS file will launch the IDE.

That's ok if the IDE wasn't already running but, if it was, Windows will launch a second copy of the IDE, which is almost certainly what you don't want.

Wouldn't it be great if you could select a .PAS file and use a context menu to instantly load it into an already running copy of Delphi? That's exactly what I'm going to demonstrate here. I've seen various other solutions to this problem and they all seem to rely upon the installation of a 'listener' package into Delphi. What this means is that you write a custom package which subclasses Delphi's main window and then sits around looking for a custom notification message which includes a pointer to a designated Pascal source file. Once that message has been received, the 'listener' can then use Delphi's Open Tools API to load the source file so it appears as a new tab on the edit window.

Although this approach works, I think it's rather inelegant. It involves two distinct chunks of code: the context menu handler which sends the notification

➤ *Listing 6*

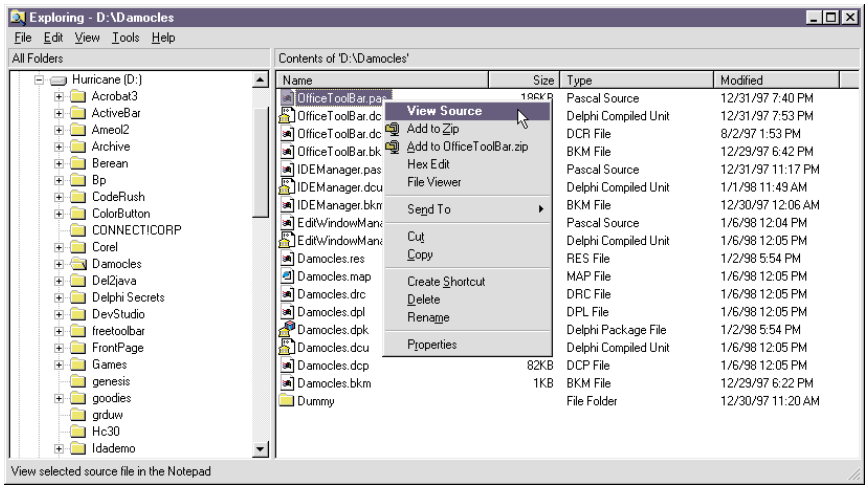
```
unit ContextM;
interface
uses
  Windows, ComObj, ComServ, ShlObj, ActiveX, ShellApi,
  SysUtils, Registry;
const
  CLSID_ContextMenuShellExtension: TGUID =
    '{A955FDC0-8819-11D1-AB26-D0E304C10000}';
type
  TContextMenu = class(TComObject, IShellExtInit,
    IContextMenu)
  private
    szFile: array [0..Max_Path] of Char;
  public
    function QueryContextMenu(Menu: hMenu; indexMenu,
      idCmdFirst, idCmdLast, uFlags: UInt):
      HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo):
      HResult; stdcall;
    function GetCommandString(idCmd, uType: UInt;
      pwReserved: PUInt; pszName: LPSTR; cchMax: UINT):
      HResult; stdcall;
    function Initialize (pidlFolder: PItemIDList; lpdobj:
      IDataObject; hKeyProgID: HKEY): HResult; stdcall;
    end;
implementation
{ The Shell calls this method when it's time for the context
  menu handler to add its own custom menu entries to the
  menu itself. We return the number of entries we've added.
function TContextMenu.QueryContextMenu (Menu: hMenu;
  indexMenu, idCmdFirst, idCmdLast, uFlags: uInt): HResult;
begin
  InsertMenu (Menu, indexMenu, mf_String or mf_ByPosition,
    idCmdFirst, 'View Source');
  Result := 1;
end;
{ The Shell calls this method when our custom menu item has
  been clicked by the user. In other words it's time to do
  the business... }
function TContextMenu.InvokeCommand (var lpici:
  TCMInvokeCommandInfo): HResult;
begin
  // Ensure we're not being called by an application
  Result := E_Fail;
  if HiWord (Integer (lpici.lpVerb)) <> 0 then Exit;
  // Verb can only be zero: we only installed one menu item
  Result := E_InvalidArg;
  if LoWord (lpici.lpVerb) <> 0 then Exit;
  // Execute the notepad with the specified file
  Result := NoError;
  WinExec (PChar (Format('Notepad %s', [szFile])),
    lpici.nShow);
end;
{ The Shell calls this method to get a 'hint' string for the
  custom menu item }
function TContextMenu.GetCommandString (idCmd, uType: uInt;
  pwReserved: puInt; pszName: LPSTR; cchMax: uInt): HRESULT;
begin
  Result := E_InvalidArg;
  if idCmd = 0 then begin
    strCopy(pszName,
      'View selected source file in the Notepad');
    Result := NOERROR;
  end;
end;
function TContextMenu.Initialize (pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult;
var
  medium: TStgMedium;
  fe: TFormatEtc;
begin
  with fe do begin
    cfFormat := cf_HDrop;
    ptd := Nil;
    dwAspect := dvAspect_Content;
    lindex := -1;
    tymed := Tymed_hGlobal;
  end;
  // Fail the call if lpdobj is Nil.
  Result := E_Fail;
  if lpdobj = Nil then Exit;
  { Render the data referenced by the IDataObject pointer to
    an HGLOBAL }
  // storage medium in CF_HDROP format.
  Result := lpdobj.GetData(fe, medium);
  if Failed (Result) then Exit;
  { If only one file is selected, retrieve the file name and
    store it in szFile. Otherwise fail the call }
  if DragQueryFile(medium.hGlobal, $FFFFFFFF, Nil, 0) = 1
    then begin
    DragQueryFile (medium.hGlobal, 0, szFile,
      SizeOf(szFile));
    Result := NOERROR;
  end else
    Result := E_Fail;
  ReleaseStgMedium (medium);
end;
initialization
  TComObjectFactory.Create (ComServer, TContextMenu,
    CLSID_ContextMenuShellExtension, '',
    'Delphi 3.0 ContextMenu Example', ciMultiInstance);
end.
```

message, and the separate listener package. I thought that it would be much neater if you could dispense with the listener package and send notifications directly to Delphi itself. But how to do it?

While playing around with the IDE, I remembered that Delphi will allow you to drag and drop files onto an edit window. This being the case, I realised there must be a window in the Delphi IDE which can respond to `WM_DROPFILES` messages. After a bit of investigation, it turned out to be the edit control itself. If you're not familiar with the anatomy of the Delphi IDE, suffice it to say that a Delphi edit window contains a standard `TTabControl` and this, in turn, contains a special edit control which acts as a wrapper around Borland's low-level text editing engine.

I wrote a couple of routines that enabled me to track down the API-level handle of the edit control, assuming that a copy of Delphi was running. So now, it was just a simple matter of sending a `WM_DROPFILES` message to this window. Well, not quite! Unfortunately, as part of the message, the API dictates that you have to provide a global memory handle which points to... an undocumented data structure. It's this data structure which contains a list of all the files (there can potentially be more than one) which are being dropped onto the target window. Windows API calls such as `DragQueryFile` are used to "peel apart" this secret data structure and return one file at a time to interested parties.

Of course, not knowing the format of this data structure, I was stuck. Without this vital information, I couldn't build a valid `WM_DROPFILES` message which would be accepted by the IDE. However, while browsing through the source code of Borland's context menu sample, I realised that a `DragQueryFile` compatible global handle is available at the time the context menu receives its `Initialize` call and this happens every time the menu is displayed. Accordingly, I modified the context menu code so as to make a copy of



➤ *Figure 3: Our first real-world context menu handler adds a 'View Source' item to the menu associated with .PAS files. Elsewhere in this article, you'll find information on a cunning (and slightly risqué!) technique for adding PAS files to the IDE's edit window.*

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{A955FDC0-8819-11D1-AB26-D0E304C10000}]
@= "Shell Extension PAS File Viewer"
[HKEY_CLASSES_ROOT\CLSID\{A955FDC0-8819-11D1-AB26-D0E304C10000}\InProcServer32]
@= "c:\Delphi 3.0\demos\shellext\contmenu.dll"
"ThreadingModel" = "Apartment"
[HKEY_CLASSES_ROOT\.pas]
@= "PasFile"
[HKEY_CLASSES_ROOT\PasFile]
@= "Pascal Source"
[HKEY_CLASSES_ROOT\PasFile\shellex\ContextMenuHandlers\{A955FDC0-8819-11D1-
  AB26-D0E304C10000}]
@= ""
```

➤ *Listing 7*

```
function TabControlEnumerator (Wnd: hWnd; cm: TContextMenu): Boolean; stdcall;
var
  szBuffer: array [0..255] of Char;
begin
  Result := True;
  GetClassName (Wnd, szBuffer, sizeof (szBuffer));
  if CompareText (szBuffer, 'TTabControl') = 0 then begin
    Result := False;
    cm.TabControlWindow := Wnd;
  end;
end;
function EditControlEnumerator (Wnd: hWnd; cm: TContextMenu): Boolean; stdcall;
var
  szBuffer: array [0..255] of Char;
begin
  Result := True;
  GetClassName (Wnd, szBuffer, sizeof (szBuffer));
  if CompareText (szBuffer, 'TEditControl') = 0 then begin
    Result := False;
    cm.EditControl := Wnd;
  end;
end;
procedure TContextMenu.DropOnDelphi;
var
  EditWindow: hWnd;
begin
  EditWindow := FindWindow ('TEditWindow', Nil);
  if EditWindow <> 0 then begin
    TabControlWindow := 0;
    EnumChildWindows (EditWindow, @TabControlEnumerator, Integer (Self));
    if TabControlWindow <> 0 then begin
      EditControl := 0;
      EnumChildWindows (EditWindow, @EditControlEnumerator, Integer (Self));
      if (EditControl <> 0) and (hGlobal <> 0) then begin
        SendMessage (EditControl, wm_DropFiles, hGlobal, 0);
      end;
    end;
  end;
end;
```

➤ *Listing 8*

the global handle, and then pass this copy on to the IDE's edit control at the time the menu item is clicked. To my surprise, it worked rather well!

Do bear in mind that this particular approach is sailing rather close to the wind in terms of what you can and can't do with the Windows API. You'll notice, for example, that I never release the memory allocated for my copy of the global handle. That's because, inside the IDE, Borland's code will (quite correctly) call the API routine `DragFinish` when it has finished and `DragFinish` is essentially just a wrapper around `GlobalFree`.

For the sake of brevity, I haven't included the full source code to this new context menu handler. The full story can be found in the file PASDROP.ZIP on this month's cover disk, but the most interesting part of the code is given in Listing 8. This code first searches for an active edit window (the IDE's edit windows are top-level windows, so they can be found using `FindWindow`) and then searches the children of the edit window for a child of type `TTabControl`. If the tab control is found, then *its* children are searched, in turn, for the edit control itself. Finally, if the edit control is found, then the context menu handler sends a `WM_DROPFILES` message to Delphi, fooling the IDE into thinking that a .PAS file has just been dropped onto the edit window.

Well, that's about it for this time. By now, you've probably had quite enough of context menus! In next month's instalment, we'll discuss some of the other shell extensions available, and I'll also be introducing you to the delights of dispinterfaces, dual interfaces and more. See you then...

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 as Dave@HexManiac.com.