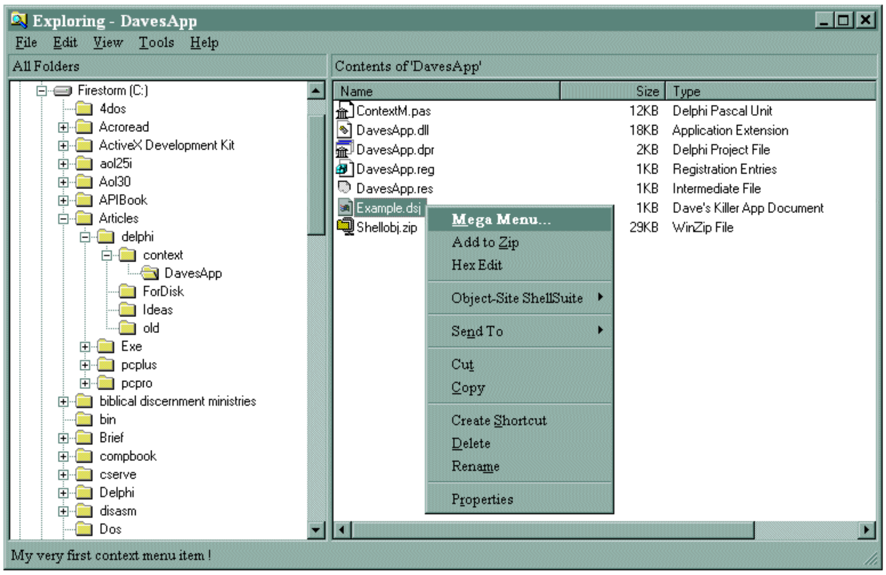
Beating the System: Getting Things Into Context!

by Dave Jewell



Name	Size	Type
ContextM.pas	12KB	Delphi Pascal Unit
DavesApp.dll	18KB	Application Extension
DavesApp.dpr	2KB	Delphi Project File
DavesApp.reg	1KB	Registration Entries
DavesApp.res	1KB	Intermediate File
Example.dsj	1KB	Dave's Killer App Document
Shellobj.zip	29KB	WinZip File

My very first context menu item !

➤ *Figure 1: Here you can see what happens when you click on a file with an extension of* `DSJ`*. The* `Mega Menu` *item has been added by our custom DLL. Note the custom file description to the right of the selected file and the status bar at the bottom of the window.*

This month, as promised, I'm going to describe how to use Delphi to implement context menus. A context menu is a popup menu displayed by the Windows 95 (and now NT 4.0) Explorer when you right-click on an item. Explorer has a number of built-in context menu items such as `Cut`, `Copy`, `Properties` and so forth. However, Explorer is a very customisable piece of software and it's possible to add your own items to a context menu depending, naturally, on the type of file which has been clicked on. In fact, it's even possible to add your own property sheets to the dialog which hangs off the `Properties` item of a context menu.

To see how useful this is in practice, it's worthwhile getting a clear picture of how Explorer implements context menus and looking at a couple of examples of how this functionality can be extended. If Explorer knows how to open a file of a specific type, it normally displays `Open` as the first item in the menu. This applies to EXE files, any document types for which an association exists and also folders. If Explorer doesn't know how to open a particular item, you'll see a context menu item marked `Open With`. This leads to a dialog box where you tell Explorer which application to associate with a particular file extension.

WinZip is an example of a program which makes good use of context menus. If you select one or more files in directory, WinZip will add an `Add To Zip` item to the menu, making it very convenient for adding a bunch of files to an archive. Similarly, if you examine the context menu for a selected ZIP file, you'll see two items: `Open` and `Extract To`. Both these items cause WinZip itself to be started.

In a similar vein, Microsoft have recently created their own new compression format: CAB files. CAB technology has been incorporated into the new version 3.0 Internet Explorer such that when a newly encountered Java class is downloaded, it's generally a CAB file that comes down the wire, thus minimising the amount of data that needs to be transferred and improving system response. As part of the CAB initiative, Microsoft have distributed a viewer program, CABVIEW, which includes full source code to a DLL that can be used to extend the functionality of Explorer, making it possible to examine the contents of a CAB file in a natural manner.

As a final example, I've recently come across a set of shell extensions called *Object-Site Shell Suite*. This adds lots of useful goodies to Explorer's context menus. My favourite is the ability to right-click on a folder and immediately start an MS-DOS command-line session with the directory set to the clicked-on folder – magic!

Registries, DLLs And COM...

Hopefully, the foregoing should have convinced you that being able to add to the context menu is a useful technique. Even if you're not in the business of writing ZIP or CAB utilities, you'll often find that there are operations which can be performed on a program's document files which can be very conveniently done from Explorer, such as validating a database file or checking a diary for urgent appointments. As with WinZip, these things may well invoke the main program behind the scenes, but the end-user perceives things as being done directly by Explorer, thus giving the impression that your application is almost a part of Windows itself.

OK, enough theory, what about the practice? The bad news is that adding shell extensions isn't a trivial job. There are three reasons for this. Firstly, you need to add some entries to the registry. These entries allow Explorer to look up a particular file extension and determine whether that file type has a custom context menu. If so, the registry entries also tell Explorer where to find the software which builds the context menu. The second reason is that this software must be in the form of a DLL and the third reason is that the DLL has to conform to the conventions of Microsoft's COM architecture. Let's start with the easy bit by looking at the required registry entries. Some familiarity with the registry architecture is assumed.

In simple terms, when you right-click on a file of a certain extension, Explorer looks up that extension in the registry. Let's suppose that the file extension is DSJ (my initials). Firstly, Explorer looks for a subkey called DSJ under the \HKEY_CLASSES_ROOT tree of the registry. If found, this key will contain a value string named Default. This string might be set to DavesApp, it tells Explorer what application is associated with this file type. Bear in mind that this isn't necessarily the human readable name of the file type, it's simply another key in the registry.

Note: As an aside, Explorer looks up this string (DavesApp in this case) under the \HKEY_CLASSES_ROOT tree to obtain the human readable name of the file. This might yield a full name such as Dave's Killer App File. It's the string that appears in the Explorer window under the Type column. It should be obvious that, for performance reasons, this registry lookup isn't happening all the time: a lot of the information is cached within Explorer. I'm just describing things this way for the sake of simplicity.

Once it has the DavesApp string, Explorer then looks for a context menu handler. It does this by looking for a registry entry named:

```
\HKEY_CLASSES_ROOT\DavesApp\
 shellex\ContextMenuHandlers
```

This architecture allows several different context menu handlers (and indeed, other types of shell extension) to be associated with the same file type, but we'll only consider the simple case here. Having looked up this entry, the Default value of the entry contains the name of the context menu handler to use. This might be DavesMenu, for the sake of argument. Assuming that it is, Explorer will then look for the key:

```
\HKEY_CLASSES_ROOT\DavesApp\
 shellex\ContextMenuHandlers\
 DavesMenu
```

This registry entry will contain a single Default value, this being the CLSID of the context menu object which we're going to implement in our DLL. If you haven't seen a CLSID before, a typical example looks like this:

```
{D2AF7A60-4C42-11CE-B27D-00AA001F73C1}
```

At this point, we are nearly there (or rather, Explorer is). Armed with the CLSID of the required COM object, it performs one final registry lookup:

```
HKEY_CLASSES_ROOT\CLSID\
 {D2AF7A60-4C42-11CE-B27D-
 0AA001F73C1}\InprocServer32
```

Finally, this key contains a Default value string which provides the full pathname of the DLL which implements the required COM object. Things can sometimes be even more complex than this. For example, it's possible to associate a context menu with any file extension. In the above discussion, you'd do this by replacing the string DavesApp with a single asterisk. It's also possible to associate a context menu with a folder rather than a file, but we'll stick to the most straightforward case here.

The reason that I've taken you through this blow-by-blow account is so that you understand exactly how the system associates a given file extension with a particular context menu. You need this information in order to add the correct entries to the registry as part of your installation procedure.

In order to make it easier for you to install my demonstration software, the ZIP file on this month's cover disk includes a file called DAVESAPP.REG, shown in Listing 1. If you haven't used REG files before, you can add the contents of this file to the registry simply by using the Import Registry File" option in the File menu of REGEDIT.EXE. On my system, right-clicking a REG file from an Explorer window will give you a context menu with a Merge option. This is more convenient, but quite frankly there are now so many add-ons installed into my Windows 95 setup that I'm not sure whether or not this is standard behaviour!

Writing The DLL

If you install the REG file as described above, Explorer will look for an in-process COM server called DAVESAPP.DLL. It expects to find this server in a folder which is called:

```
c:\articles\delphi\context\DavesApp
```

Of course, this probably isn't where you decide to put the DLL: just be sure to change the REG file to agree with whatever path you use before merging with the system registry. Once you've done it, if you create a small file with an extension of DSJ, you should find

➤ *Listing 1: DAVESAPP.REG*

```
REGEDIT4
[HKEY_CLASSES_ROOT\.dsj]
  @="DavesApp"
[HKEY_CLASSES_ROOT\DavesApp]
  @="Dave's Killer App Document"
[HKEY_CLASSES_ROOT\CLSID\{C0A3EA22-1C7A-11d0-BF05-444553540000}]
  @="Dave's Killer App Document Context Menu"
[HKEY_CLASSES_ROOT\CLSID\{C0A3EA22-1C7A-11d0-BF05-444553540000}\InprocServer32]
  @="c:\\articles\\delphi\\context\\DavesApp\\DavesApp.dll"
“ThreadingModel”="Apartment"
[HKEY_CLASSES_ROOT\DavesApp\shellex\ContextMenuHandlers]
  @="DefMenu"
[HKEY_CLASSES_ROOT\DavesApp\shellex\ContextMenuHandlers\DefMenu]
  @="{C0A3EA22-1C7A-11d0-BF05-444553540000}"
```

that Explorer reports the file type as Dave's Killer App Document.

The main source to the DLL is given in Listing 2. This file is called DAVESAPP.DPR, so as to produce a DLL with the required name. The resulting DLL file is just under 18Kb in size. You'll notice that the code listing starts off with the declaration of a const item, OurCLSID. This is the CLSID by which the COM object implemented by the DLL is known. This constant must agree with the CLSID definitions used in the associated REG file. Strictly speaking, you should generate a new ID using Microsoft's UUIDGEN.EXE utility before installing the software. Do bear in mind that not all possible numbers are valid: you can't just randomly dream up a CLSID of your own. If you pass an invalid CLSID, your new COM object will be politely ignored. If you do change the CLSID, bear in mind that it appears three times in the REG file.

In order to conform to the COM specification, an in-process server must export two functions: DllGetClassObject and DllCanUnloadNow. The names of these functions must not be changed, the system expects to see them! Both these routines are implemented within the DAVESAPP.DPR code. The first, DllGetClassObject, is responsible for creating a new instance of a special type of class called a class factory. As the name suggests, a class factory is used to create other objects. This might seem like a weird way of doing things. You might be tempted to ask why the DLL can't directly create objects of the type we're interested in. The reason for this is quite complicated, but suffice to say that it gives the system more flexibility and allows us more control over the structure of the DLL.

The DllGetClassObject function simply checks to ensure that it's being called with the correct CLSID and that a compatible interface is being requested. If so, it instantiates a copy of the shell factory object and exits. The DllCanUnloadNow routine is even simpler: it checks a couple of global variables maintained by the ContextM unit (which we'll look at in a moment) to determine whether or not the DLL can be unloaded. This is to prevent the DLL from being unloaded while there are still instantiated objects belonging to the DLL: a potentially disastrous scenario!

At this point, we've only described the exported interface to the class factory. The important thing about class factories is the type of objects which they themselves can create. In order to look deeper, we need to examine the code for the ContextM unit, this is given in Listing 3. This is quite a lengthy file and I've only included the most important methods: the full file is on this month's disk.

Interfacing To Explorer

The class factory's sole job is to create objects of type TContextMenuObject for anybody who wants one, in this case, Explorer. The TContextMenuObject is, in turn, only a wrapper around two other objects. These two objects are derived from IContextMenu and IShellExtInit. You can find information on these standard classes on the MSDN disk and in the Win32 SDK documentation. Put simply, IContextMenu is responsible for adding commands to Explorer's context menu and is also called when a custom menu item has been selected. The IShellExtInit class is mainly responsible for getting the name of the selected file from Explorer.

You might find this all a bit confusing. Why do we need all these classes? Well, the important thing is that whatever object is created by the class factory, it must be able to supply the two interfaces mentioned above. Since Object Pascal doesn't support multiple inheritance (thankfully!) the simplest method of providing two interfaces from a single object is to construct that object as a wrapper around two other objects which implement the interfaces we're interested in. That's all that TContextMenuObject is used for: it's just a convenient way of achieving the effect we want. When you examine the listing, you'll notice that the two child objects (for want of a better word) both implement the IUnknown interface (the call to QueryInterface) by simply calling back to the owner's QueryInterface handler. In the same way, the

➤ Listing 2: DAVESAPP.DPR

```
library DavesApp;
uses Ole2, Windows, ContextM;
const
  { This ID MUST match what we put into the registry }
  OurCLSID: TGUID = (D1:$C0A3EA22; D2:$1C7A; D3:$11d0;
                     D4:($BF, $05, $44, $45, $53, $54, $00, $00));
{ Exported function - Create the class factory }
function DllGetClassObject (const clsid: TCLSID; const iid: TGUID; var ppv):
  HResult; stdcall export;
var ShellClassFactory: IShellClassFactory;
begin
  Pointer (ppv) := Nil;
  Result := Class_E_ClassNotAvailable;
  { Validate the passed CLSID }
  if IsEqualIID (clsid, OurCLSID) then begin
    { Validate the interface ID }
    Result := E_NoInterface;
    if (IsEqualIID (iid, IID_IUnknown)) or
      (IsEqualIID (iid, IID_IClassFactory)) then
      try
        // Instantiate the class factory
        ShellClassFactory := IShellClassFactory.Create;
        Result := ShellClassFactory.QueryInterface (iid, ppv);
        if Result < 0 then ShellClassFactory.Free;
      except
        Result := e_OutOfMemory;
        Pointer (ppv) := Nil;
     end;
  end;
end;
{ Exported function - See if DLL can be unloaded }
function DllCanUnloadNow: HResult;
begin
  if (LockCount = 0) and (ObjCount = 0) then Result := 0
  else Result:= 1;
end;
exports
  DllGetClassObject, DllCanUnloadNow;
begin
end.
```

AddRef and Release methods are passed straight back to the owner. In the TContextMenuObject.QueryInterface routine, the owner object looks to see what sort of interface is being requested and points the caller at the appropriate child object. By doing things like this, an instance of TContextMenuObject looks and behaves just like a single object as far as interested parties are concerned, but the effect has been achieved without any of the unpleasantness of multiple inheritance!

From here on, things are fairly plain sailing. The QueryContextMenu method of the owned IContextMenu object is responsible for adding your own custom menu items to the context menu. You're supplied with a conventional API-level menu handle and told where to start placing items (the indexMenu parameter). In this simple example, we only add a single item to the menu, but in a more complex example such as WinZip, you might do something more sophisticated. Do exercise some restraint though: remember that the idea is for your application to appear to seamlessly integrate into Windows 95, not to take over the entire operating system. If you find yourself adding a dozen items to the context menu, you're probably going about things the wrong way! The QueryContextMenu returns the total number of items added as its function result. Strictly speaking, this is an HResult type, but since the other fields are all zero anyway, we can get away with passing an ordinary integer value.

The next important method of TOwnedContextMenu is the GetCommandString function. Explorer calls this when a particular menu item is selected but hasn't actually been chosen. In other words, the user hasn't clicked the mouse or pressed Enter at this point. This gives the context menu handler an opportunity to supply a hint string which is displayed in Explorer's status bar along with other ordinary menu hint information. After checking that Explorer is requesting a hint string (the value of the uType parameter) the method copies the appropriate string into the buffer supplied by Explorer. Because we've only added one menu item, the idCmd parameter is only going to be zero. However, if we'd added more menu items, then we'd need to implement a full case statement.

The final IContextMenu method of interest is InvokeCommand. At this point, the punter really has made up his/her mind and clicked on a menu item! We can tell which menu item has been selected by examining the low-order word of the lpVerb field inside the passed data structure. Again, in this simple case it's just going to be zero, but you'd normally implement another case statement. In this example, I've simply displayed a rather pointless message box, but you'd normally do something more significant. In the case of most applications such as WinZip, you'll want to call CreateProcess (or WinExec if you prefer the simple life) to launch your primary application. Typically, you'll pass the pathname of the required document and maybe an additional switch parameter to indicate which of several context menu items was selected.

The final piece in the jigsaw is TOwnedShellExtInit, our derived version of IShellExtInit. The most important method here is Initialize. My code here has been more or less taken from the MSDN documentation and converted it to

➤ *Listing 3*

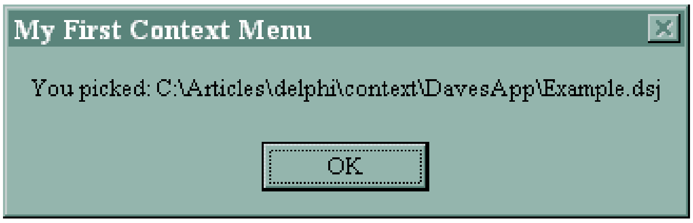
```
function TOwnedShellExtInit.Initialize(pidlFolder: Pointer;
  pdobj: IDataObject; hKeyProgID: HKey): HResult;
var fmte: TFormatEtc;
    medium: TStgMedium;
begin
  Result := E_Fail;  { Assume the worst! }
  if pdobj <> nil then begin
    fmte.cfFormat := cf_hDrop;
    fmte.ptd := nil;
    fmte.dwAspect := dvAspect_Content;
    fmte.lindex := -1;
    fmte.tymed := tymed_hGlobal;
    { Use given IDataObject to get a list of filenames }
    Result := pdobj.GetData(fmte, medium);
    if Result < 0 then
      Result := E_Fail  { Ensure only one file is selected }
    else
      if DragQueryFile(HDrop(medium.hGlobal),
        UInt(-1), Nil, 0) = 1 then begin
        { Stash the filename }
        SetLength (owner.fName, 512);
        DragQueryFile(HDrop(medium.hGlobal), 0,
          PChar(owner.fName), 512);
        Result := 0;
      end else
        Result := E_Fail;
    ReleaseStgMedium (medium);
  end;
end;
{ Add commands to a context menu }
function TOwnedContextMenu.QueryContextMenu(hMenu: hMenu;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UInt): HResult;
begin
  { add our new menu item }
  InsertMenu (hMenu, IndexMenu, mf_String or
    mf_ByPosition, idCmdFirst, '&Mega Menu...');
  Result := 1;  { return number of items added }
end;
{ Execute a given menu command }
function TOwnedContextMenu.InvokeCommand(
  lpici: PCMInvokeCommandInfo): HResult;
var sz: array [0..255] of Char;
begin
  Result := E_Fail;
  if HiWord(LongInt(lpici.lpVerb)) = 0 then begin
    if loWord(lpici.lpVerb) > 0 then
      Result := E_InvalidArg
    else begin
      { Normally, you'd case out on the menu identifier here }
      case loWord(lpici.lpVerb) of
        0: begin
             wvsprintf(sz,
               'You picked: %s', @owner.fName);
             MessageBox(lpici.hwnd, sz,
                'My First Context Menu', mb_ok);
             Result := 0;
           end;
      end;
    end;
  end;
end;
{ Return a menu item hint string }
function TOwnedContextMenu.GetCommandString(idCmd, uType:
 UInt; var res: UInt; pszName: LPStr; cchMax: UInt): HResult;
{ Explorer is requesting a menu hint string }
const gcs_HelpText = 1;
begin
  { If uType = gcs_HelpText, return menu hint string for Explorer }
  Result := e_NotImpl;
  if uType = gcs_HelpText then begin
    { Case out on the menu item }
    case idCmd of
      0: begin
           lstrcpy (pszName,
             'My very first context menu item!');
           Result := 0;
         end
      else Result := E_InvalidArg;
    end;
  end;
end;
```

Pascal. This method has only one aim in life, to extract the name of the selected file from the passed IDataObject and store it in the fName member field of the owner object. Notice that there's an explicit check that only one file is selected. If you select more than one DSJ file and then right-click the mouse, you won't see any additions to the context menu. Again, this may or may not be what you want: the WinZip context menu handler will allow you to select multiple files and add them to a new archive in one quick operation. It all depends on the nature of your application and what you want to do.

Conclusions

Although the ContextM unit is only about 350 lines of source code, don't be fooled! I worked hard to achieve this level of apparent simplicity! In practice, most Delphi applications that want to add to Explorer's namespace (a term which includes custom context menus, property sheet extensions and much more) will need to use a large unit called SHELLOBJ.PAS. This unit, in turn, calls another unit called REGSTR.PAS. These units should have been supplied by Borland but weren't. Instead, some enterprising individual took the trouble to convert the C/C++ definitions into Object Pascal and released the files into the public domain. I've included the necessary units on this month's disk as a file called SHELLOBJ.ZIP.

Strictly speaking, I should have employed these additional units in my example program, but I thought you might be overwhelmed by the number of large units needed to do the job. Instead, I stripped away everything except the bare minimum needed for writing context menu handlers and added the necessary class and method definitions to the implementation part of ContextM. In this way, the unit is stand-alone and we both have some chance of understanding what's going on! I do encourage you, though, to delve deeper into the SHELLOBJ.PAS unit since there are many other goodies in there which you can use to extend Explorer's functionality in various ways.


My First Context Menu

You picked: C:\Articles\delphi\context\DavesApp\Example.dsj

➤ *Figure 2: Our fully featured example!*

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave by email as DaveJewell@msn.com, DSJewell@aol.com or on CompuServe as 102354,1572