July 1996

MICROSOFT SYSTEMS JOURNAL

# Extending the Windows Explorer with Name Space Extensions

David Campbell

*David Campbell is a Support Engineer on the Microsoft Premier Developer Support team who specializes in Windows shell extensions as well as Microsoft Visual C++. He also likes cheese.*

Windows¨95 and Windows NT¨ 4.0 contain a mechanism that allows information to be integrated into the internal hierarchical data structures of the Windows Explorer. This hierarchical data, the "name space," contains information about objects displayed in the Explorer's panes. A name space is a collection of symbols, such as database keys or file and directory names.

The Windows 95 shell uses a single hierarchical name space to organize all objects such as files, storage devices, printers, network resources, and anything else that can be viewed using Explorer. The root of this unified name space is the Windows 95 desktop. While similar to a file system's directory structure, the name space contains more types of objects than just files and directories.

OK, but why do I care? This name space mechanism can be extended to include new items. You can write a name space extension to add your own custom data, and custom views on that data, into the Explorer's internal name space. With a freshly installed Windows 95 or Windows NT 4.0, a standard name space is part of the product. This name space includes the standard components you see with a clean install of Windows. There is code in the operating systems that interact with Explorer to represent the standard name space. Therefore, your name space extension is additional code that allows Explorer to represent your additional name space data.

Since the name space extension mechanism is based on COM, I will assume you are familiar with COM. You should also be familiar with writing shell extensions. If not, read Jeff Prosise's article "Integrate Your Applications with the Windows 95 User Interface Using Shell Extensions" in the March 1995 issue of *MSJ*.

Note that the information presented here is subject to change and is based on the Cab File Viewer sample recently released by Microsoft. The Cab File Viewer was released as part of the Power Toys collection of applications and extensions to enhance Windows 95. Download the Power Toys collection from http://www.microsoft.com/windows/software/powertoy.htm. The Cab File Viewer extends the name space to allow Explorer to browse into CABinet files. These are compressed archive files, similar to ZIP files, that Microsoft uses to distribute software, including Windows 95. The Cab File Viewer name space extension provides code that Explorer can use to display CAB file contents. You can get source for the Cab File Viewer along with preliminary documentation on Explorer's name space mechanism from http://www.microsoft.com/win32dev. To build a name space extension, you will also need an updated version of ShlObj.H, which is included in the Cab File Viewer sample source and should also be in the next release of the Win32¨ SDK.

The name space is a very large topic, and even the subset implemented in the Cab File Viewer is too large to be adequately covered in this article. Rather than walk you painfully through the code, I will instead cover information on the name space mechanism itself to allow you to understand the basics of name space extensions. You'll then be able to review the sample code on your own along with the new header files, and explore some of the more advanced topics, including the ability to generate per item context menus, icon handlers, and support for drag and drop.

## What is the Shell Name Space?

Windows 95 and Windows NT 4.0 use a data structure-the name space-that represents the hierarchy of objects all the way from the desktop to every item that can be seen in Explorer. From the desktop you can view Network Neighborhood, My Briefcase, Recycle Bin, and My Computer. From My Computer you can get to drives, Control Panel, Printers, and Dial-Up Networking. Look at the left pane of Explorer to see the hierarchy of objects clearly. These items are called virtual folders-virtual because they refer to items in the name space that can contain other name space items, as opposed to groups of files.

Explorer uses COM interfaces to let the user access and manipulate internal name space data and to display it in a graphical form. In the COM paradigm, you are supposed to manipulate and extend the internal name space structures using COM interfaces instead of directly accessing the name space data.

You may have already noticed that the EnumDesk sample from the MSDN CD and the Windows Explorer look almost exactly the same. This is because both Explorer and EnumDesk walk though the name space and display the information in the name space. Both Explorer and EnumDesk call into the name space to enumerate the contents of a folder. They can then step through the items in the folder and call another interface to find out what to display. I will go through these steps in more detail later in this article.

## Types of Name Space Extensions

There are two high-level topics to discuss on name space extensions before I jump into the details: rooted versus nonrooted name space extensions and the point of entry.

The difference between rooted extensions and nonrooted extensions is how they're used. There is no code difference between the two. A rooted extension is meant to stand alone. Essentially, the extension is the root of the tree and can only see its own branches. To see an example of a rooted extension, right-click on the Windows 95 taskbar, choose Properties, click the Start Menu Programs tab, and click the Advanced button. You will see an Explorer-like window with the Start Menu folder as the root, as in **Figure 1**. In the case of the Cab File Viewer (see **Figure 2**), it is also a rooted extension, since the window shown does not let you step backwards to the CAB file.

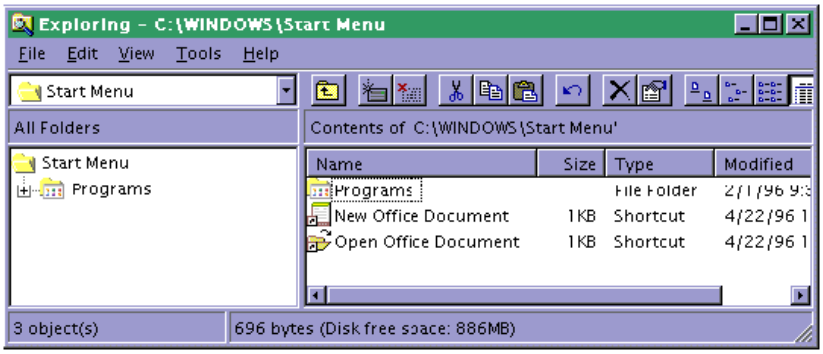


**Figure 1 Rooted name space**

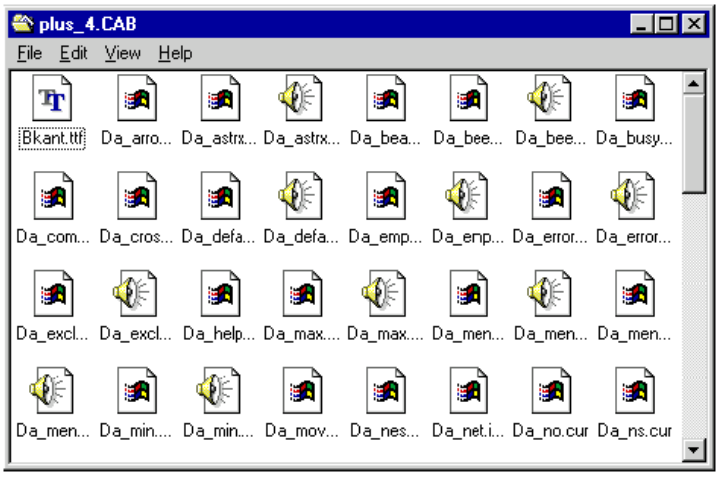


**Figure 2 CAB File Viewer**

A nonrooted use of a name space extension keeps the entire name space in mind. Right click on the Windows 95 Start button on the taskbar and select Explore from the pop-up menu. In this case, the exact same name space extension is shown in an Explorer window (see **Figure 3**), except you can step back from the Start Menu folder and traipse around the rest of the name space.

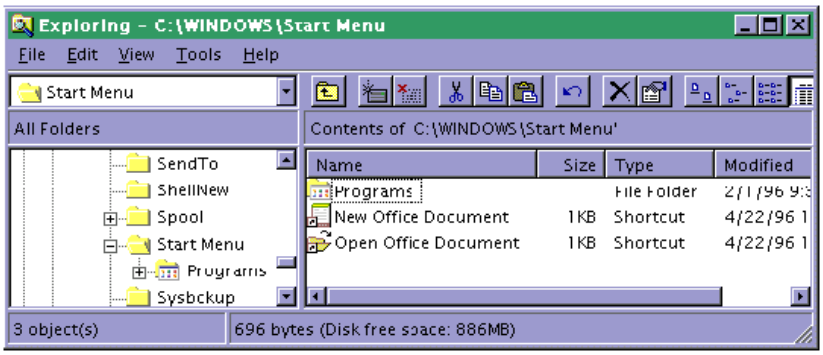


**Figure 3 Nonrooted name space**

The implementation of the name space extension is basically the same for both kinds. Which method you use depends on your extension and is a matter of style and common sense as much as anything else.

Now let's talk about entry points. The root, or top level, of your name space is referred to as the junction point. In the case of the Cab File Viewer, the junction point is the CAB file itself. There is a restriction in the current Explorer-any name space that is implemented with a file (like CAB) as its junction point must be rooted, since the Explorer does not support exploring directly into files.

An alternative is to use a directory as the junction point. To do this, you must create a directory, change the directory's attributes to read-only, and place a file called Desktop.ini into it. This INI file then specifies the CLSID of your extension:

```
[.ShellClassInfo]
CLSID={CLSID}
```

This file basically replaces the CAB entry in the registry.

Yet another alternative is to use the CLSID of your name space extension as the file extension of a folder. If you wanted to create a folder called Cheese that was really the junction point of a name space extension, you would actually create a folder named "Cheese.{CLSID}".

Another difference between name space implementations is the actual "entry point" from the user's point of view. The user can enter via an icon like the Recycle Bin on the desktop or in the My Computer folder, for example. Or the entry point can be a file type (or directory) that is registered to your extension, like the Cab File Viewer.

You can place an entry point on the desktop or in the My Computer folder in several ways. First, you can put information in the registry that results in an icon being placed on the desktop, the icon coming from the CLSID in this registry entry:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
Explorer\Desktop\Namespace\{CLSID}\(default) =
    "Description of my extension"
```

This is what you'd use to place the icon in the My Computer folder:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
Explorer\MyComputer\Namespace\{CLSID}\(default) =
    "Description of my extension"
```

So should you implement a rooted or nonrooted extension? And should the entry point be on the desktop or somewhere else? It depends on your extension. Does your extension fit into the logical hierarchy of the name space? Does it make sense to move up a level in the hierarchy from your name space? If not, a rooted extension may make more sense.

Suppose you are implementing an extension to browse Usenet newsgroups. Does it make sense to have more than one entry point? Not really, so an entry point on the desktop seems reasonable. What about rooting? You could argue either way, but I would think that it should be rooted. What if you wanted to browse into a database file of some sort? Well, since it's reasonable for more than one to exist on your machine, it makes sense to use the location of the file itself as the entry point. As for rooting, if you use the file as the junction point, a rooted extension is your only option.

### Registering the Extension

Like all shell extensions, a name space extension is registered in the Windows registry so that Explorer can determine its location and what services are available.

Here's the basic layout and options for the registry. First, if you are registering a file type to be treated like a name space extension, you need:

```
HKEY_CLASSES_ROOT\.EXT\(default)=CLSID\{CLSID}  ;
```

This is only necessary if you are registering an extension for a specific file.

The rest is similar to other shell extensions,

```
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\InProcServer32\(default) =
    "C:\\WINDOWS\\SYSTEM\\ShellExt\\CabView.dll"
```

and

```
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\InProcServer32\"ThreadingModel" =
    "Apartment"
```

Of course you must also have a CLSID entry that matches

```
HKEY_CLASSES_ROOT\CLSID\{CLSID}
```

and the keys shown in **Figure 4** under it.

The Cab File Viewer registers the extension under the file extension for CAB files, similar to the way you register context menu handlers, property sheet extensions, and other shell extensions.

```
HKEY_CLASSES_ROOT\.cab\(default)="CLSID\\{0CD7A5C0-
    9F37-11CE-AE65-08002B2E1262}"
```

Then a key is created for the CLSID,

```
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}
```

Next, the following entries are added. Note the reference to the CLSID for the CabView.dll and the rooted Explorer.

```
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\InProcServer32\(default)=
    "C:\\WINDOWS\\SYSTEM\\ShellExt\\CabView.dll"
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\InProcServer32\"ThreadingModel" =
    "Apartment"
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\DefaultIcon\(default)=
    "C:\\WINDOWS\\SYSTEM\\ShellExt\\CabView.dll"
HKEY_CLASSES_ROOT\CLSID\{0CD7A5C0-9F37-11CE-AE65-08002B2E1262}\shell\open\command\(default)= "Explorer /
    root,{0CD7A5C0-9F37-11CE-AE65-08002B2E1262},%1"
```

Explorer invokes your extension using the exact command line you registered, so you can test your command line by executing it directly with the Run command on the Start menu. For example, executing

```
Explorer.exe /root, {your CLSID}, [path]
```

should start up the Explorer and your extension.

### The Difference Between a Shell Extension and a Name Space Extension

Shell extensions are code that modifies or enhances the functionality of the entire operating system shell. Name space extensions are just one type of shell extension. Therefore, all name space extensions are shell extensions. Because of this, there is some standard shell extension code you will need to use when writing your name space extension (which I will cover in a moment).

Let me clear something up. The terms "shell" and "Explorer" are often used interchangeably in some of Microsoft's documentation. This is because Explorer.exe is the entire Windows shell. A shell extension is in fact an Explorer extension. It is confusing because most people think of the Explorer as the window (shown in **Figure 5**) that explores the name space. Explorer is also the application that owns and controls the desktop and the name space data. When you're implementing a name space extension (or any shell extension), think of the big picture as just a collection of shell extensions, all owned by Explorer. Other parts of Windows 95 system code also use these extensions, including the File Open and File Save common dialogs.

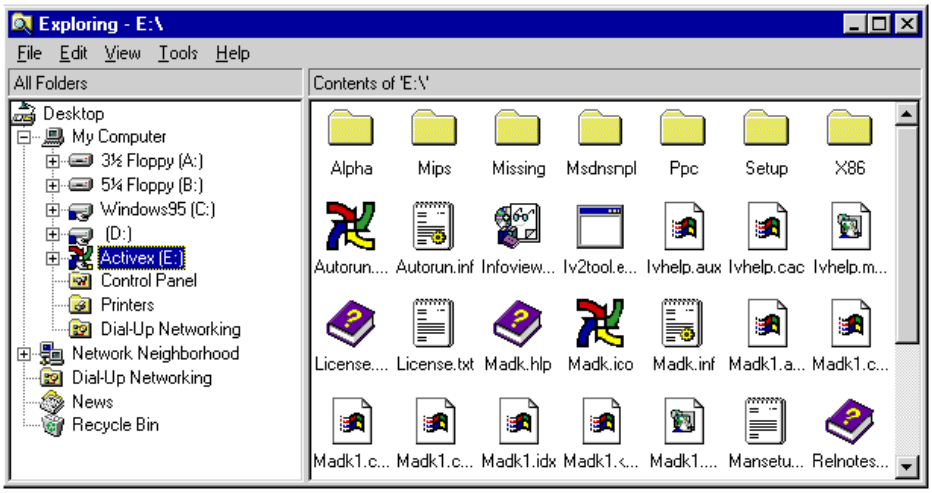


**Figure 5 Explorer**

A name space extension is to Explorer as an interpreter is to a tourist. Since your custom name space data is in a language Explorer does not natively know, your name space extension will translate your custom data to a format that Explorer can understand.

OK, so what's a browser? A browser is an application that interacts with the name space to provide the user with a visual representation and a mechanism to manipulate the data. The EnumDesk sample (see **Figure 6**) from the MSDN CD is a browser. The Explorer window in **Figure 5** is also a browser. The goal of name space extensions is to eliminate the need to write a complete browser.

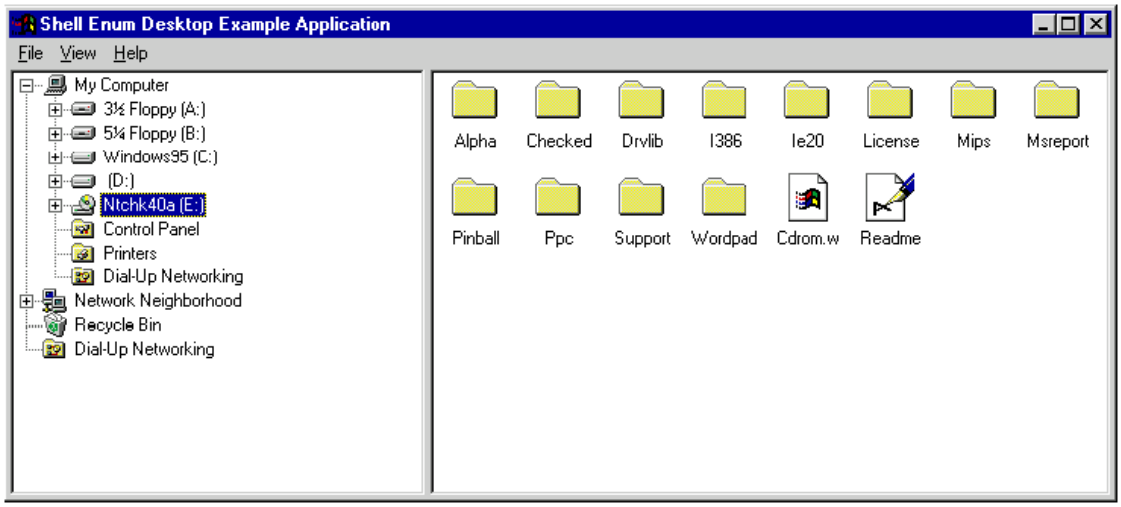


**Figure 6 EnumDesk**

### Why Write a Name Space Extension?

A name space extension enables Explorer to let users view and manipulate custom data objects in familiar Explorer-like ways. Until now, developers wanting this level of integration had to implement their own custom browser (like EnumDesk) that handled their own data type information along with the standard name space. This meant that the user needed a different browser for each custom data type! Further, the custom browser solution doesn't allow you to take advantage of future enhancements in Explorer's browser technology (when Windows 2001 comes out, users won't want to be forced to still use EnumDesk!). With a name space extension, not only does the user only need one browser (Explorer's), but users will be able to take advantage of enhancements to Explorer and continue to use your data.

### Anatomy of a Name Space Extension

A name space extension allows you to define a new object that a name space browser like Explorer can explore and allows the user to interact with. Your extension code, which is implemented as an OLE inproc server, manages the display of the icon images and text that the user sees while viewing your data, as well as the menus and toolbars the user can use to interact with your data. This new object can be used for any number of things, including displaying the contents of your email inbox or Internet newsgroups, the contents of a zip file or a database of some sort, document management system or whatever, assuming that the information can be presented reasonably in a graphical way. Very sophisticated name space extensions can actually do things like take the symbol in your name space and decode it on the screen. An example would be if there was a URL in your name space, but you displayed the Web page in the browser instead of the URL. In this example, the data being displayed by the browser is not the actual data in the name space; instead it is something referenced by the data in the name space. However, deviating from name space extensions that essentially list items and attributes to full-featured document-style editing is better handled through ActiveX, which is not covered in this article.

To see an extension in action, download and install the Cab File Viewer, and then browse into a CAB file, such as the ones on your Windows 95 install CD or the Plus! Pack. **Figure 7** shows a directory containing CAB files. Explorer recognizes the CAB file format once the viewer is installed, and displays the icons registered to the extension in the registry under the {CLSID}\DefaultIcon key. (In this case the icon is similar to the one displayed for directories.)

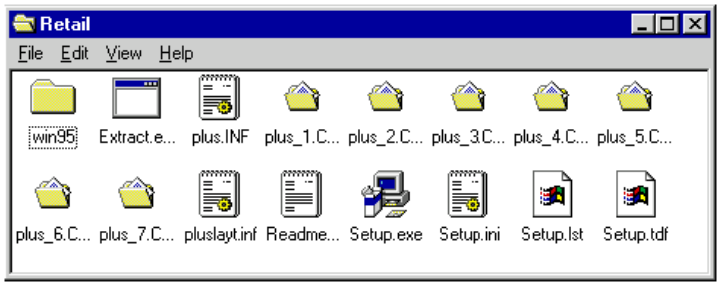


**Figure 7 CAB Files**

Nothing magic so far, since you can register an icon for almost any type of file. Open one of the CAB files and you'll see the extension in action (refer back to **Figure 2**). Notice how the contents of the CAB file appear as if the files were in a directory.

Cab File Viewer works by registering its file type, CAB, as a name space extension and provides the information required by Explorer to display its name space contents. In this case CAB File Viewer provides a list of file names, icons, and attributes just as if the CAB file was a directory containing files.

### Implementing an Extension

So what do you have to implement to create a name space extension? Your extension has to provide Explorer several things:

1. A list of the items in your name space, called the ID list.
2. A view of the items, usually icons and text labels
3. Optional custom context menus for the items, and other UI goodies such as drag and drop functionality.

You probably feel like you have seen this before-the MFC document/view architecture works on a very similar fashion. The list of the IDs in the name space correlate to the MFC document object, and the view of the items relate to the MFC view object.

### Name Space Extension Core Code

As mentioned earlier, all shell extensions (and hence your name space extension) must be implemented as OLE inproc DLLs. They must contain at least the functions DllMain, DllGetClassObject, DllCanUnloadNow, and the standard IUnknown interface. Finally, all shell extensions must be implemented using the apartment threading model. The apartment threading model allows your process to contain more than one thread of execution because it uses message passing to deal with multiple objects running concurrently.

That's what's required to create a basic, albeit bland, OLE inproc DLL that can be used as the frame for an extension. From there, extensions deviate since the interfaces and methods they support depend on the type of extension being implemented.

A name space extension must implement the IShellFolder, IEnumIDList, IPersistFolder and IShellView interfaces. Their methods (see **Figure 8**) are used to interact with Explorer. Explorer makes calls into the extensions using the IShellFolder and IShellView interfaces, and the extension can call back into Explorer using its IShellBrowser implementation.

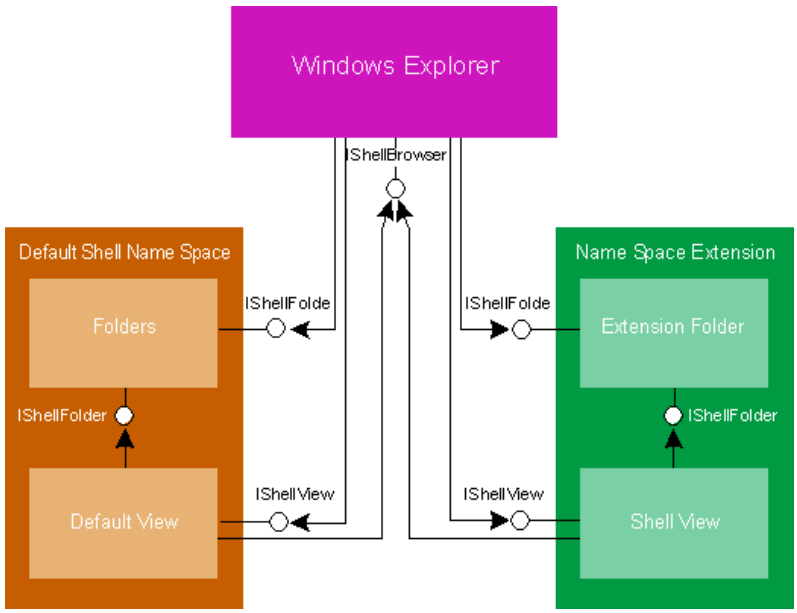


**Figure 8 Calling into a Name Space Extension**

### Name Space Extension Specialized Code

Once you have your core code, then you need to add the code that is specific to your name space. The "Name Space Extensions-Quick Reference" pullout (between pages 48 and 49) lists the key interfaces and their methods, as well as brief descriptions. The Win32 SDK and the MSDN CD contain complete descriptions of these interfaces, including parameter descriptions, return values and some sample usage code in some cases. These interfaces and methods are defined in ShlObj.h and ShlGuid.h, which have recently changed to include the new interface definitions necessary to implement name space extensions. These files will be included in the next Win32 SDK, but in the meantime you can find them with the sample code for this article.

When Explorer goes to display the contents of your name space, it loads your extension via OLE, using the CLSID you registered to locate your inproc server DLL. It then calls your extension's QueryInterface to get the interfaces and methods it needs. For example, Explorer will query for IPersistFolder once your DLL is loaded. If Explorer receives a valid interface pointer, it will initialize your code by calling the IPersistFolder::Initialize member and will pass your extension a pointer to an ID list (or PIDL). A PIDL points to a data structure that identifies your code in the overall name space.

### PIDLing Around with ID Lists

An ID list is a group of shell item IDs. Each identifier is an array of bytes that contains data specific to the name space code that allocated the identifier. The first two bytes of this array must be a byte count that defines the length of the item, but the rest of the data is not used by any code outside the name space extension. The rest of the data identifies the item to the name space extension that allocated it. For example, an identifier that points to a file on disk may be as follows:

| | |
|---|---|
| Bytes 0-1 | Byte count |
| Bytes 2-*n* | Full path |

If the name space extension wants to cache away details, it could make the identifier as follows:

| | |
|---|---|
| Bytes 0-1 | Byte count |
| Bytes 2-258 | Path and file name |
| Bytes 259-260 | File's size in bytes |
| Bytes 260-261 | Create date |
| Bytes 262-263 | Attributes |
| Bytes 264-*n* | Icon |

In this case, caching away the file details speeds up the name space's responses to Explorer's queries, which improves UI performance.

When these identifiers are formed into an ID list, they are simply concatenated one after another and a final ID with a zero byte count is used as a terminator. An ID list that contains only one ID is called a simple ID list, one with more than one ID is called a complex ID list. For consistency, most methods that accept a shell item ID as a parameter take a PIDL, although you must watch out since some methods expect simple ID lists and others can handle complex ID lists. For example, CompareIDs, GetAttributesOf, GetUIObjectOf, and SetNameOf expect simple ID lists, while BindToObject, BindToStorage, and GetDisplayNameOf can be passed complex ID lists. The documentation indicates which methods handle complex ID lists.

Using a directory structure as an analogy for the name space, a PIDL can be compared to a path. Just as there are absolute and relative paths in DirectoryStructureLand, there are absolute and relative PIDLs. A relative PIDL is only meaningful inside its name space location. In the case of a name space extension's items, the PIDL is pointing to a buffer of data that only has meaning to the name space extension itself. Since this item can be any structure you wish, you may want to include information such as a friendly name, actual location, attributes, and other details that the extension can use for speedy access.

### ID List Rules and Regs

An important note about ID lists is that they must not be dependent on where they are stored in memory. They cannot contain pointers to data within themselves or any data external to them. This is because an ID list can be saved to persistent storage (for example, shortcut files contain a persisted ID list), and then read back into memory later and presented back to the IShellFolder that first enumerated the item (except the ID list is now in a different chunk of memory). An ID list can also be duplicated into another memory block, and then be presented to the IShellFolder. In the example of an identifier that refers to a file on disk, this means that the identifier/structure pointed to by the PIDL must contain the actual path, not just a pointer to a character string that contains the path. The identifier must be

```
USHORT; char[__MAXPATH];
```

not

```
USHORT; char*;
```

This also means all IShellFolder implementations need to be both backward- and forward-compatible with ID lists that they write out. That is, if the ID list format for a folder is revised, it must work well (not crash!) with old and new implementations of the IShellFolder. Be very careful when designing the format of your ID lists. A very good way to do this is to have an ID type field immediately after the byte count. For example, while bytes 0-1 are used to indicate the byte count for the ID, bytes 2-3 should be used to "tag" the rest of the data in the ID with some format code. If the name space extension does not recognize the format code, it knows (from the byte count) how many bytes to skip, and therefore can ignore the ID gracefully.

### Enumerating the ID List

Once initialized, Explorer will request an IShellFolder interface to access information about your name space's contents. Explorer uses IShellFolder::EnumObjects to get a list of your extension's contents, IShellFolder::GetUIObjectOf to get the appropriate icons and menus, and IShellFolder::GetAttributesOf to get various attributes of each item, including whether or not that item has subfolders. Since the contents of your name space are known only to your name space extension, all information required by Explorer regarding your data must be implemented by your name space extension code.

How does Explorer walk through the items in your name space and identify each one to your code when it needs a menu or an icon? This is where the PIDL comes in. Explorer walks through your identifiers using your name space extension's exposed IEnumIDList interface to get each item's PIDL.

Because Explorer calls IShellFolder::CompareIDs to locate an item in your name space using an identifier you returned earlier, you must be consistent with your IDs. CompareIDs is also used when sorting lists such as the tree in the Explorer. Explorer can also use CompareIDs to check whether or not two PIDLs actually point to the same object.

### Implementing a View of Your Namespace

The IShellBrowser interface (also shown in the "Name Space Extensions-Quick Reference" pullout) is implemented within Explorer, not your name space extension code. It provides a way for your extension to communicate back to the Explorer with visible feedback to the user, including menus, status text, and toolbars. IShellBrowser also gives you a private stream to store persistent state information, such as the layout of items, the viewing mode selected (large icon, small icon, list, or details) or whatever the extension might need the next time this name space is explored.

When the user enters or opens a name space extension, Explorer must create a view object to display the contents. This is done by calling the name space extension's IShellFolder::CreateViewObject member function and getting an IShellView interface. The Explorer then calls IShellView::CreateViewWindow, which tells the extension to create the view of its folder's contents. Typically you would use a listview control to display your contents, as the Cab File Viewer does. Your extension uses the IShellBrowser::GetWindow method to get a window handle of its parent and it passes a RECT pointer to determine the window's position.

When the Explorer calls IShellView::CreateViewWindow, it passes an IShellBrowser interface pointer, which allows the extension to call back into Explorer to add status information, menus, and toolbar buttons. The relationship between the IShellBrowser and IShellView interfaces is similar to that of IOleInPlaceFrame and IOleActiveObject.

The UI mechanism used is similar to OLE's in-place activation mechanism, with a few differences. First, the view window, which the

extension creates, can exist even though it may not have the input focus. It has to maintain three states: deactivated, activated with focus, and activated without focus. The view window can present a different set of menus depending on the state. Explorer notifies the extension of any state changes by calling its IShellView::UIActivate member. The Shell View object, in return, calls IShellBrowser::OnViewWindowActivate when the view window is activated by the user with a mouse click.

Unlike a typical OLE in-place activation, Explorer does not support any type of layout negotiation, but it does let the view window add toolbar buttons and set status bar text, and the view window can communicate with these controls through calls to IShellBrowser::GetControlWindow or IShellBrowser::SendControlMsg.

Finally, Explorer allows the view window to add menu items to its pulldown menus and insert top-level pulldown menus. The view object is allowed to insert menu items to submenus returned from IShellBrowser::InsertMenusSB. For Explorer to dispatch menu messages correctly, menu item IDs must be between FCIDM_SHVIEWFIRST and FCIDM_SHVIEWLAST, which are defined in the ShlObj.h file. While OLE's standard in-place mechanism allows the UI active object to add pull-down menus (where you don't need to worry about menu item IDs), the shell's UI negotiation mechanism also allows the active view to add menu items to parents' pull-down menus. Action menu items (such as Open or Cut) should only be available if the view is activated with focus. This is done by manipulating the menu attached to the control window via standard Win32 menu APIs.

When you navigate around the name space on your computer, you are going to be bouncing around all different types of name spaces and associated views. For example, you may click on the Control Panel folder in the left pane of Explorer and then click on the folder CHEESE.CHZ under the Drive C: folder. In this example, your view changes from a Control Panel applet view to a file directory view.

To keep the user from going batty, Explorer strives to keep the same look and feel when switching views. If you were in Details mode when looking at Control Panel, Explorer assumes you want to look at the file directory in details mode also. Explorer maintains a little data structure called State Information that's passed from the soon-to-disappear view to the soon-to-appear view. In the example mentioned above, the State Information would indicate Details mode.

When your view window is created, you are given this State Information as a parameter in your IShellView::CreateViewWindow method. Likewise, your IShellView::GetCurrentInfo method must supply this State Information when it is called by Explorer.

For consistency, Explorer passes a pointer to the IShellView interface used for the previously active view as a parameter to your IShellView::CreateViewWindow before calling the previously active view's DestroyViewWindow. This allows your view object to transfer appropriate state information from the previous view object.

The IShellBrowser::GetViewStateStream method gets a pointer to a stream to store your state and attribute settings when your IShellView::SaveViewState method is called. This gives you a place to store the current view settings so you can restore them in your next session.

**Figure 9** is a table that lists the files in the Cab File Viewer, the interface(s) implemented in the file, highlighting those covered here. The files are excerpted in **Figure 10**. I'm not going to walk through the code, but it's a good idea to review it and use it as a reference for adding features to your own name space extension. One of the best places to look for information regarding shell interfaces is the SHLOBJ.H file itself.

### Debugging

Debugging shell extensions can be tricky and name space extensions are no exception. My favorite method is to use the Just-in-Time debugging capabilities of Visual C++¨ 4.*x*. I code a DebugBreak into my extension at a strategic location and let Windows 95 invoke the debugger automatically for me when the DebugBreak statement is executed. The nice thing about the DebugBreak API is that you don't have to exit Explorer and restart it within the debugger. The extension is running under the same circumstances it will ultimately be running in when it's complete. At the point where I hit the DebugBreak call, I'll be looking at the assembly language. If I step out of that function and close the assembly window I'll be looking at my source code, which Visual C++ automatically locates and loads. The other nice thing about using DebugBreak is that you can program the DebugBreak statement to be hit under a certain set of circumstances, for example, if a flag changes state or a counter drops below zero or whatever.

Now that you're having fun debugging your extension, here are a couple of common implementation errors to watch for. First, all accesses to nonconstant global variables of DLLs must be serialized by a critical section or a mutex. Otherwise, you risk having your global data manipulated unexpectedly by other threads. Second, objects returned from IShellView::CreateViewObject or IShellView::GetUIObjectOf(IShellView, IDropTarget, IContextMenu, and so on) must be newly allocated each time they're called. Implementing an IShellView interface in the same object that implements IShellFolder (using C++ multiple inheritance) is a very common mistake. Having a pointer to your IShellView in the IShellFolder object is another bad idea, since one IShellFolder object can have multiple dynamically changing views.

### Beyond the Basics: Name Space Giblets

So far I've discussed only a basic implementation of a namespace extension that can be viewed from Explorer. Your extension can implement much more: drag and drop, printing, copying, toolbars, and so on. The following can be used to enhance the functionality of your name space extension.

IExtractIcon can be implemented to provide custom "on-the-fly" icons for different data types within your name space. To support drag and drop, your view must support the standard OLE IDropSource and/or IDropTarget interfaces, depending on whether you want to support dragging, dropping, or both. IContextMenu interface can be implemented to provide context menus for your items.

### Considerations for Windows NT 4.0

Almost any extension written for Windows 95 should work on Windows NT 4.0 (with the same compatibility considerations as apps) with only an additional registry key, and if you are writing a shell extension for Windows 95 you should take the extra time to test your application on the Windows NT 4.0 beta currently available.

I cannot stress enough the importance of testing your extension carefully on both Windows 95 and Windows NT 4.0. Although the systems strive for compatibility, they are separate operating systems and there can be differences. While most differences encountered are minor, they are much easier to fix before you release your code than after.

Make sure your code checks the platform it's running on and uses only features that are available on that platform. Although almost all Win32 API are supported on both, the Win32 SDK does contain some that are targeted at a specific platform. (The Win32 SDK documentation can tell you which APIs are platform-specific.)

The Microsoft Knowledgebase Article Q92395 contains information on determining the platform you are running on. At the very least give the user a reasonable message saying that this platform is not supported, such as "Tough luck, Bozo!"

For the vast majority of shell extensions, the only new requirement for Windows NT is that it has an "approved" list of shell extensions in the registry and you must add yourself to this list before your shell extension will run.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\
ShellExtensions\Approved\{CLSID}="Name of Extension"
```

The only other thing to be aware of is that Windows NT also provides UNICODE[a] support for shell extensions, although it is compatible with the Windows 95 ANSI implementations.

### Conclusion

One of the more tedious tasks of developing name space extensions is that you have to implement a fair number of small methods that barely change from implementation to implementation to be fully consistent with Windows and the internal name space. In fact, many are implemented by the shell itself. It would be nice if a custom name space implementation could aggregate the existing shell interfaces, allowing developers to concentrate on customizing a particular area, rather than having to totally redevelop entire interfaces. Hopefully this will change in a future version.

Finally, remember that question: "Just because you can do it, should you?" I bring this up not to discourage you from using the name space extension mechanism, but rather to provoke you into thinking carefully about what you are trying to accomplish and whether or not it's the best solution. A name space extension is a complex piece of code and may be overkill in many cases. Sometimes just registering an application and its data file type in the registry is what you really want.

*The author would like to thank Dhananjay Mahajan, Chris Guzak, Satoshi Nakajima, Mary Kirtland, and Francis Hogle for their help.*

*From the July 1996 issue of Microsoft Systems Journal.*