# Windows 95 Shell Extension Registration, Debugging, and Support

*by Nancy Winnick Cluts*

The Windows 95 shell is extensible. You can access tools that manipulate objects in the shell name space, and you can browse through the file system and networks.

All shell extensions are implemented as Component Object Model (COM) objects and reside in dynamic-link libraries (DLLs). Once you grasp the basics of how to implement COM objects, the additional information you need to understand to implement a shell extension should be fairly simple.

The Windows 95 shell supports seven types of shell extensions (referred to as handlers):

- **Context menu handlers.** Add menu items to the context menu for a particular file object. The context menu is displayed when a user clicks a file object with the alternate (usually the right) mouse button. Context menu handlers implement the **IContextMenu** and **IShellExtInit** interfaces.
- **Drag-and-drop handlers.** Context menu handlers that are accessed when a user drops an object after dragging it to a new location. Drag-and-drop handlers implement the **IContextMenu** and **IShellExtInit** interfaces. The only difference between a context menu handler and a drag-and-drop handler is the way it is registered in the registration database.
- **Icon handlers.** Add instance-specific icons for file objects or add icons for specific file classes. Icon handlers implement the **IExtractIcon** and **IPersistFile** interfaces.
- **Property sheet handlers.** Add pages to the property sheet dialog box the shell displays for a

file object. Property sheet handlers implement the **IShellPropSheetExt** and **IShellExtInit** interfaces.

- **Copy hook handlers.** Prevent a folder or printer object from being copied, moved, deleted, or renamed. Copy hook handlers implement the **ICopyHook** interface.
- **Drop target handler.** Controls the action that occurs when the shell drops objects on other objects. Drop target handlers implement the **IDropTarget** and **IPersistFile** interfaces.
- **Data object handler.** Supplies the object when files are being dragged and dropped or copied and pasted. Data object handlers implement the **IDataObject** and **IPersistFile** interfaces.

## Register now

Like all COM objects, shell extensions must be registered in the registration database, or they won't work. Each extension must register its class ID (CLSID) under HKEY_CLASSES_ROOT\CLSID in the registry. Within this key, the extension adds an **InProcServer32** key that gives the location of the extension's DLL. The first line in the sample below registers the CLSID of a property sheet extension called NancyPropSheet. The second line specifies the location of the DLL containing the extension and the threading model.

```
[HKEY_CLASSES_ROOT\CLSID\{771a9da0-731a-11ce-
    993c-00aa004adb6c}]
 @="NancyPropSheet"
[HKEY_CLASSES_ROOT\CLSID\{771a9da0-731a-11ce-
    993c-00aa004adb6c}\InprocServer32]
 @="c:\\windows\\system\\propext.dll"
 "ThreadingModel"="Apartment"
```

Your shell extension must also be registered under the **shellex** key, which contains the information the shell uses to associate a shell extension with a file type. You can map your shell extension to a particular class of file (based upon the file extension), or you can specify that the shell extension is valid for files of all types. In the property sheet extension sample above, the property sheet was registered specifically for NWCFile as follows:

```
[HKEY_CLASSES_ROOT\.NWC]
 @="NWCFile"
[HKEY_CLASSES_ROOT\NWCFile]
 @="Shell Extension file"
[HKEY_CLASSES_ROOT\NWCFile\shellex\
    PropertySheetHandlers]
 @="NWCPage"
[HKEY_CLASSES_ROOT\NWCFile\shellex\
    PropertySheetHandlers\NWCPage]
 @= "{771a9da0-731a-11ce-993c-00aa004adb6c}"
```

Now say you want all files to reap the benefits of your special property page. To do this, you specify '*' after

HKEY_CLASSES_ROOT as seen in the following example:

```
[HKEY_CLASSES_ROOT\*\shellex\
    PropertySheetHandlers]
 @="NWCPage"
[HKEY_CLASSES_ROOT\*\shellex\
    PropertySheetHandlers\NWCPage]
 @= "{771a9da0-731a-11ce-993c-00aa004adb6c}"
```

## Initial impressions

Besides registration in the registration database, shell extensions share the way in which they are initialized. The shell uses two interfaces to initialize instances of shell extensions: **IShellExtInit** and **IPersistFile**.

The **IShellExtInit** interface is used to initialize context menu handlers, drag-and-drop handlers, and property sheet handlers, while **IPersistFile** is employed to initialize instances of icon handlers, data object handlers, and drop target handlers. (Copy hook handlers don't use the **IShellExtInit** or **IPersistFile** interface.)

After implementing the initialization interface for your shell extension, you need to implement the actual interface for the particular shell extension you are writing. For example, if you are writing a property sheet extension, you will need to implement the **IShellPropSheetExt** interface.

## Debugging shell extensions

When your shell extension is written and registered, what do you do if it doesn't work? It's not like most applications, where you can simply include **DebugBreak** calls in your code or run the Visual C++ development system in debug mode, set some breakpoints, and go.

Because shell extensions are loaded at the startup of the Explorer, you have to find a way to start the Explorer without loading all of its DLLs. Here's how:

1. Go into your project settings and, under the Debug tab, type the path to EXPLORER.EXE in the Executable For Debug Session edit box.
2. Close all applications you are running, and turn off your computer.
3. Restart Windows 95 and Visual C++, loading your shell extension.
4. Click the Start button and choose Shutdown.
5. Now, hold down the CTRL+ALT+SHIFT keys and simultaneously click No. (Sounds a bit like playing Twister, doesn't it?)
6. The desktop will go blank and your heart will start palpitating, but don't worry. If you press ALT+TAB you can get to your instance of Visual C++.

7. At this point you are ready to debug. Set your breakpoints and go.

If you want to be able to force the shell to unload DLLs very quickly so you don't need to exit it to debug your shell extension, you can do this through a setting in the registry. Under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer, add the **AlwaysUnloadDll** key and set its value to a number (I set it to 1 because that's the kind of gal I am). Adding this key sets the time-out value for DLLs to a very small value.

## Shell extensions and Windows NT

You may wonder whether your shell extension will run on the Windows NT operating system. Windows 95 shell extensions can work on Windows NT 3.51. This version gives a user the option of running Windows NT 3.51 with the Windows 95 shell. To do so, however, there is one additional step that your application must take in its setup process and registration.

For the Windows NT shell to recognize and run a shell extension, the handler's CLSID must also be listed under a new registry key (in addition to the registry entries you need to make for Windows 95) that contains a list of the handlers approved for the shell to run. By default, this key's access control permissions allow only someone with administrator privileges to modify the list. The CLSID must be registered under Windows NT at the following location:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved

To register the extension, a "named value" should be added to the **Approved** key. The name of the value must be the string form of the CLSID that can be obtained through the **StringFromCLSID** function.

For example, to register the property sheet extension shown above, you can add the following line to your registration file:

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\
  Windows\CurrentVersion\Shell Extensions\
  Approved\{771a9da0-731a-11ce-993c-
  00aa004adb6c}] @="NancyPropSheet"
```

## Security questions

If you haven't written applications for Windows NT, it may seem odd that your setup application may not be able to write to this key. The ability to write to the key depends on the access privileges of the person installing the application. The setup application should attempt to open the key described above,

requesting the KEY_SET_VALUE permission. If it succeeds, the new CLSID can be added to fully register the corresponding shell extension.

If the request fails due to a security violation, the user installing the application does not have permission to register new shell extensions. In this case, the setup application might warn the user that some application features will not be available unless an administrator turns them on (by installing the application, or by writing the registry keys directly). Or, if the shell extension is crucial to the application's functioning, the setup application might cause the installation to fail completely, notifying the user that the program must be installed by an administrator.

The following sample code demonstrates how an application can register its shell extension under Windows NT:

```
// First, attempt to open the registry key
// where approved extensions are listed.
long err;
HKEY hkApproved;

err = RegOpenKeyEx(
 HKEY_LOCAL_MACHINE,
 "Software\\Microsoft\\Windows\\
  CurrentVersion\\Shell Extensions\\Approved",
  0,
  KEY_SET_VALUE,
  &hkApproved);

if (err == ERROR_ACCESS_DENIED)
{
 // The user does not have permissions to add
 // a new value to this key.
.
.
.
}
else if (err == ERROR_FILE_NOT_FOUND)
{
// The key does not exist. This should only
// happen if setup is running on Windows 95
// instead of Windows NT, or if you are
// installing on an older version of either
// operating system that does not have the
// Win95 UI.
.
.
.
}
else if (err != ERROR_SUCCESS)
{
// Some other problem...
}
else
{
 // Assume that lpstrProgID contains our
 // ProgID string.
 LPSTR lpstrProgID = "My Bogus Class";

 // Assume that clsidExtension contains the
 // CLSID struct. The code below creates a
 // string from this CLSID. If a string
 // version of the CLSID is already handy,
 // skip this code.
```

```
CLSID clsidExtension = {0x11111111, 0x1111,
  0x1111, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
  0x11, 0x11};
HRESULT hr;
LPOLESTR lpolestrCLSID;
CHAR rgchCLSID[40];
CoInitialize(NULL);
hr = StringFromCLSID(clsidExtension,
     &lpolestrCLSID);
// StringFromCLSID returns a Unicode string,
// so convert to ANSI for calling the
// registry. Note that on Windows NT you can
// call the Unicode version of the registry
// API instead.
WideCharToMultiByte(CP_ACP, 0, lpolestrCLSID,
  -1, rgchCLSID, 40, NULL, NULL);
CoTaskMemFree(lpolestrCLSID);
CoUninitialize();

// Now add the new value to the registry.
err = RegSetValueEx(
    hkApproved,
    rgchCLSID,
    0,
    REG_SZ,
    (const BYTE *)lpstrProgID,
    strlen(lpstrProgID));
// Finally, close the key.
err = RegCloseKey(hkApproved);
```

That's all there is to it. For more detailed information about how to implement the different shell extensions, consult the Microsoft Development Library; some sample code is also available in the merged Win32 Software Development Kit.

*Nancy Winnick Cluts, a writer on the Developer Network's DevTech team, retired from a highly successful career as an international super-model to write* Programming the Windows 95 User Interface, *to be published by Microsoft Press in August.*