The Complete Idiot's Guide to Writing Shell Extensions - Part I

Michael Dunn, 15 Mar 2006


4.92 (271 votes)

Rate this: vote 1vote 2vote 3vote 4vote 5

A step-by-step tutorial on writing shell extensions

- Download demo project - 46K

Contents

README.TXT

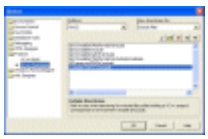
This is the stuff I want you to read first, before proceeding on or posting messages to this article's discussion board.

This series of articles was originally written for VC 6 users. Now that VC 8 is out, I felt it was about time to update the articles to cover VC 7.1. (Also, the automatic 6-to-7 conversion done by VC 7.1 doesn't always go smoothly so VC 7.1 users could get stuck when trying to use the demo source code.) So as I go through and update this series, the articles will be updated to reflect new VC 7.1 features, and I'll have VC 7.1 projects in the source downloads.

Important note for VC 2005 users: The Express edition of VC 2005 does **not** come with ATL or MFC. Since the articles in this series use ATL, and some use MFC, you won't be able to use the Express editions with the articles' sample code.

If you are using VC 6, you should get an updated Platform SDK. You can use the web install version, or download the CAB files or an ISO image and run the setup locally. Be sure you use the utility to add the SDK include and lib directories to the VC search path. You can find this in the *Visual Studio Registration* folder in the Platform SDK program group. It's a good idea to get the latest Platform SDK even if you're using VC 7 or 8 so you have the latest headers and libs.

Important note for VC 7 users: You **must** make a change to the default include path if you haven't updated your Platform SDK. Make sure that $(VCInstallDir)PlatformSDK\include is first in the list, above ($VCInstallDir)include, as shown here:



Since I haven't used VC 8 yet, I don't know if the sample code will compile on 8. Hopefully the 7-to-8 upgrade process will work better than the 6-to-7 process did. Please post on this article's forum if you have any trouble with VC 8.

Introduction to the Series

A shell extension is a COM object that adds some kind of functionality to the Windows shell (Explorer). There are all kinds of extensions out there, but very little easy-to-follow documentation about what they are. (Although I bet the situation has improved during the six years since I originally wrote that!) I highly recommend Dino Esposito's great book *Visual C++ Windows Shell*

Programming (ISBN 1861001843) if you want an in-depth look into lots of aspects of the shell, but for folks who don't have the book, or only care about shell extensions, I've written up this tutorial that will astound and amaze you, or failing that, get you well on your way to understanding how to write your own extensions. This guide assumes you are familiar with the basics of COM and ATL. If you need a refresher on COM basics, check out my Intro to COM article.

Part I contains a general introduction to shell extensions, and a simple context menu extension to whet your appetite for the following parts.

There are two parts in the term "shell extension." *Shell* refers to Explorer, and *extension* refers to code you write that gets run by Explorer when a predetermined event happens (e.g., a right-click on a .DOC file). So a shell extension is *a COM object that adds features to Explorer*.

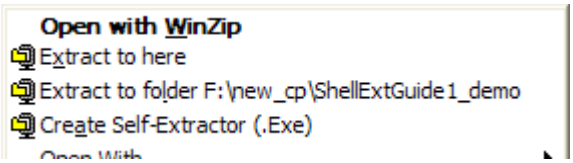
A shell extension is an in-process server that implements some interfaces that handle the communication with Explorer. ATL is the easiest way to get an extension up and running quickly, since without it you'd be stuck writing QueryInterface() and AddRef() code over and over. It is also *much* easier to debug extensions on Windows NT-based OSes, as I will explain later.

There are many types of shell extensions, each type being invoked when different events happen. Here are a few of the more common types, and the situations in which they are invoked:

Type	When it's invoked	What it does
Context menu handler	User right-clicks on a file or folder. In shell versions 4.71+, also invoked on a right-click in the background of a directory window.	Adds items to the context menu.
Property sheet handler	Properties dialog displayed for a file.	Adds pages to the property sheet.
Drag and drop handler	User right-drags items and drops them on a directory window or the desktop.	Adds items to the context menu.
Drop handler	User drags items and drops them on a file.	Any desired action.
QueryInfo handler (shell version 4.71+)	User hovers the mouse over a file or other shell object like My Computer.	Returns a string that Explorer displays in a tooltip.

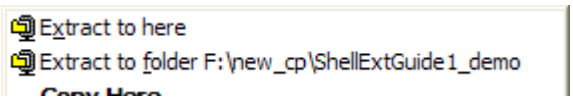
Introduction to Part I

By now you many be wondering what an extension looks like in Explorer. One example is WinZip - it contains many types of extensions, one of them being a context menu handler. Here some commands that WinZip adds to the context menu for compressed files:



WinZip contains the code that adds the menu items, provides fly-by help (text that appears in Explorer's status bar), and carries out the appropriate actions when the user chooses one of the WinZip commands.

WinZip also contains a drag and drop handler. This type is very similar to a context menu extension, but it is invoked when the user drags a file using the right mouse button. Here is what WinZip's drag and drop handler adds to the context menu:



There are many other types (and Microsoft keeps adding more in each version of Windows!). For now, we'll just look at context menu extensions, since they are pretty simple to write and the results are easy to see (instant gratification!).

Before we begin coding, there are some tips that will make the job easier. When you cause a shell extension to be loaded by Explorer, it will stay in memory for a while, making it impossible to rebuild the DLL. To have Explorer unload extensions more often, create this registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\AlwaysUnloadDLL

and set the default value to "1". On 9x, that's the best you can do. On NT, go to this key:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer

and create a DWORD called *DesktopProcess* with a value of 1. This makes the desktop and Taskbar run in one process, and subsequent Explorer windows each run in its own process. This means that you can do your debugging with a single Explorer window, and when you close it, your DLL is automatically unloaded, avoiding any problems with the file being in use. You will need to log off and back on for these changes to take effect.

I will explain how to debug on 9x a little later.

Using AppWizard to Get Started

Let's start simple, and make an extension that just pops up a message box to show that it's working. We'll hook the extension up to .TXT files, so our extension will be called when the user right-clicks a text file.

OK, it's time to get started! What's that? I haven't told you how to use the mysterious shell extension interfaces yet? Don't worry, I'll be explaining as I go along. I find that it's easier to follow along with examples if the concepts are explained, and followed immediately by sample code. I could explain everything first, then get to the code, but I find that harder to absorb. Anyway, fire up VC and we'll get started.

Run the AppWizard and make a new ATL COM program. We'll call it "SimpleExt". Keep all the default settings in the AppWizard, and click *Finish*. We now have an empty ATL project that will build a DLL, but we need to add our shell extension's COM object. In the ClassView tree, right-click the *SimpleExt classes* item, and pick *New ATL Object*. (In VC 7, right-click the item and pick *Add*|*Add Class*.)

In the ATL Object Wizard, the first panel already has *Simple Object* selected, so just click *Next*. On the second panel, enter "SimpleShlExt" in the *Short Name* edit box (the other edit boxes on the panel will be filled in automatically):

By default, the wizard creates a COM object that can be used from C and script-based clients through OLE Automation. Our extension will only be used by Explorer, so we can change some settings to remove the Automation features. Go to the *Attributes* page, and change the *Interface* type to *Custom*, and change the *Aggregation* setting to *No*:

When you click OK, the wizard creates a class called CSimpleShlExt that contains the basic code for implementing a COM object, and adds this class to the project. We will add our code to this class.

The Initialization Interface

When our shell extension is loaded, Explorer calls our QueryInterface() function to get a pointer to an IShellExtInit interface. This interface has only one method, Initialize(), whose prototype is:

```
Hide   Copy Code
HRESULT IShellExtInit::Initialize (
  LPCITEMIDLIST pidlFolder,
  LPDATAOBJECT pDataObj,
  HKEY hProgID )
```

Explorer uses this method to give us various information. pidlFolder is the PIDL of the folder containing the files being acted upon. (A PIDL [**p**ointer to an **ID l**ist] is a data structure that uniquely identifies any object in the shell, whether it's a file system object or not.) pDataObj is an IDataObject interface pointer through which we retrieve the names of the files being acted upon. hProgID is an open HKEY which we can use to access the registry key containing our DLL's registration data. For this simple extension, we'll only need to use the pDataObj parameter.

To add this to our COM object, open the *SimpleShlExt.h* file, and add the lines listed below in bold. Some of the COM-related code generated by the wizard isn't needed, since we're not implementing our own interface, so I've indicated the code that can be removed with strikeout type:

Hide Copy Code

```
#include <shlobj.h>
#include <comdef.h>

class ATL_NO_VTABLE CSimpleShlExt :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CSimpleShlExt, &CLSID_SimpleShlExt>,
  public ISimpleShlExt,
  public IShellExtInit
{
  BEGIN_COM_MAP(CSimpleShlExt)
    COM_INTERFACE_ENTRY(ISimpleShlExt)
    COM_INTERFACE_ENTRY(IShellExtInit)
  END_COM_MAP()
```

The COM_MAP is how ATL implements QueryInterface(). It tells ATL what interfaces other programs can retrieve from our COM objects.

Inside the class declaration, add the prototype for Initialize(). We'll also need a buffer to hold a filename:

Hide Copy Code

```
protected:
  TCHAR m_szFile[MAX_PATH];

public:
  // IShellExtInit
  STDMETHODIMP Initialize(LPCITEMIDLIST, LPDATAOBJECT, HKEY);
```

Then, in the *SimpleShlExt.cpp* file, add the definition of the function:

Hide Copy Code

```
STDMETHODIMP CSimpleShlExt::Initialize (
  LPCITEMIDLIST pidlFolder,
  LPDATAOBJECT pDataObj,
  HKEY hProgID )
```

What we'll do is get the name of the file that was right-clicked, and show that name in a message box. If there is more than one selected file, you could access them all through the pDataObj interface pointer, but since we're keeping this simple, we'll only look at the first filename.

The filename is stored in the same format as the one used when you drag and drop files on a window with the WS_EX_ACCEPTFILES style. That means we get the filenames using the same API: DragQueryFile(). We'll begin the function by getting a handle to the data contained in the IDataObject:

Hide Copy Code

```
HRESULT CSimpleShlExt::Initialize(...)
{
FORMATETC fmt = { CF_HDROP, NULL, DVASPECT_CONTENT,
                -1, TYMED_HGLOBAL };
STGMEDIUM stg = { TYMED_HGLOBAL };
HDROP     hDrop;

  // Look for CF_HDROP data in the data object. If there
  // is no such data, return an error back to Explorer.
  if ( FAILED( pDataObj->GetData ( &fmt, &stg ) ))
    return E_INVALIDARG;
```

```
  // Get a pointer to the actual data.
  hDrop = (HDROP) GlobalLock ( stg.hGlobal );

  // Make sure it worked.
  if ( NULL == hDrop )
    return E_INVALIDARG;
```

Note that it's vitally important to error-check everything, especially pointers. Since our extension runs in Explorer's process space, if our code crashes, we take down Explorer too. On 9x, such a crash might necessitate rebooting the computer.

Now that we have an HDROP handle, we can get the filename we need.

Hide Copy Code

```
  // Sanity check – make sure there is at least one filename.
UINT uNumFiles = DragQueryFile ( hDrop, 0xFFFFFFFF, NULL, 0 );
HRESULT hr = S_OK;

  if ( 0 == uNumFiles )
    {
    GlobalUnlock ( stg.hGlobal );
    ReleaseStgMedium ( &stg );
    return E_INVALIDARG;
    }

  // Get the name of the first file and store it in our
  // member variable m_szFile.
  if ( 0 == DragQueryFile ( hDrop, 0, m_szFile, MAX_PATH ) )
    hr = E_INVALIDARG;

  GlobalUnlock ( stg.hGlobal );
  ReleaseStgMedium ( &stg );

  return hr;
}
```

If we return E_INVALIDARG, Explorer will not call our extension for this right-click event again. If we return S_OK, then Explorer will call QueryInterface() again and get a pointer to another interface: IContextMenu.

The Interface for Interacting with the Context Menu

Once Explorer has initialized our extension, it will call the IContextMenu methods to let us add menu items, provide fly-by help, and carry out the user's selection.

Adding IContextMenu to our shell extension is similar to adding IShellExtInit. Open up *SimpleShlExt.h* and add the lines listed here in bold:

Hide Copy Code

```
class ATL_NO_VTABLE CSimpleShlExt :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CSimpleShlExt, &CLSID_SimpleShlExt>,
    public IShellExtInit,
    public IContextMenu
{
  BEGIN_COM_MAP(CSimpleShlExt)
    COM_INTERFACE_ENTRY(IShellExtInit)
    COM_INTERFACE_ENTRY(IContextMenu)
  END_COM_MAP()
```

And then add the prototypes for the IContextMenu methods:

Hide Copy Code

```
public:
  // IContextMenu
  STDMETHODIMP GetCommandString(UINT, UINT, UINT*, LPSTR, UINT);
  STDMETHODIMP InvokeCommand(LPCMINVOKECOMMANDINFO);
```

```
  STDMETHODIMP QueryContextMenu(HMENU, UINT, UINT, UINT, UINT);
```

Modifying the context menu

IContextMenu has three methods. The first one, QueryContextMenu(), lets us modify the menu. The prototype of QueryContextMenu() is:

```
Hide   Copy Code
HRESULT IContextMenu::QueryContextMenu (
  HMENU hmenu, UINT uMenuIndex, UINT uidFirstCmd,
  UINT uidLastCmd, UINT uFlags );
```

hmenu is a handle to the context menu. uMenuIndex is the position in which we should start adding our items. uidFirstCmd and uidLastCmd are the range of command ID values we can use for our menu items. uFlags indicates why Explorer is calling QueryContextMenu(), and I'll get to this later.

The return value is documented differently depending on who you ask. Dino Esposito's book says it's the number of menu items added by QueryContextMenu(). The MSDN docs from VC 6 says it's the command ID of the last menu item we add, plus 1. The online MSDN says this:

If successful, returns an HRESULT value that has its severity value set to SEVERITY_SUCCESS and its code value set to the offset of the largest command identifier that was assigned, plus one. For example, assume that idCmdFirst is set to 5 and you add three items to the menu with command identifiers of 5, 7, and 8. The return value should be MAKE_HRESULT(SEVERITY_SUCCESS, 0, 8 - 5 + 1). Otherwise, it returns an OLE error value.

I've been following Dino's explanation so far in the code I've written, and it's worked fine. Actually, his method of making the return value is equivalent to the online MSDN method, as long as you start numbering your menu items with uidFirstCmd and increment it by 1 for each item.

Our simple extension will have just one item, so the QueryContextMenu() function is quite simple:

```
Hide   Copy Code
HRESULT CSimpleShlExt::QueryContextMenu (
  HMENU hmenu, UINT uMenuIndex, UINT uidFirstCmd,
  UINT uidLastCmd, UINT uFlags )
{
  // If the flags include CMF_DEFAULTONLY then we shouldn't do anything.
  if ( uFlags & CMF_DEFAULTONLY )
    return MAKE_HRESULT ( SEVERITY_SUCCESS, FACILITY_NULL, 0 );

  InsertMenu ( hmenu, uMenuIndex, MF_BYPOSITION,
               uidFirstCmd, _T("SimpleShlExt Test Item") );

  return MAKE_HRESULT ( SEVERITY_SUCCESS, FACILITY_NULL, 1 );
}
```

The first thing we do is check uFlags. You can look up the full list of flags in MSDN, but for context menu extensions, only one value is important: CMF_DEFAULTONLY. This flag tells namespace extensions to add only the default menu item. Shell extensions should not add any items when this flag is present. That's why we return 0 immediately if the CMF_DEFAULTONLY flag is present. If that flag isn't present, we modify the menu (using the hmenu handle), and return 1 to tell the shell that we added 1 menu item.

Showing fly-by help in the status bar

The next IContextMenu that can be called is GetCommandString(). If the user right-clicks a text file in an Explorer window, or selects a text file and then clicks the *File* menu, the status bar will show fly-by help when our menu item is highlighted. Our GetCommandString() function will return the string that we want Explorer to show.

The prototype for GetCommandString() is:

```
Hide   Copy Code
HRESULT IContextMenu::GetCommandString (
  UINT idCmd, UINT uFlags, UINT* pwReserved,
  LPSTR pszName, UINT cchMax );
```

idCmd is a zero-based counter that indicates which menu item is selected. Since we have just one menu item, idCmd will always be zero. But if we had added, say, 3 menu items, idCmd could be 0, 1, or 2. uFlags is another group of flags that I'll describe later. We can ignore pwReserved. pszName is a pointer to a buffer owned by the shell where we will store the help string to be displayed. cchMax is the size of the buffer. The return value is one of the usual HRESULT constants, such as S_OK or E_FAIL.

GetCommandString() can also be called to retrieve a "verb" for a menu item. A verb is a language-independent string that identifies an action that can be taken on a file. The docs for ShellExecute() have more to say, and the subject of verbs is best suited for another article, but the short version is that verbs can be either listed in the registry (such as "open" and "print"), or created dynamically by context menu extensions. This lets an action implemented in a shell extension be invoked by a call to ShellExecute().

Anyway, the reason I mentioned all that is we have to determine why GetCommandString() is being called. If Explorer wants a fly-by help string, we provide it. If Explorer is asking for a verb, we'll just ignore the request. This is where the uFlags parameter comes into play. If uFlags has the GCS_HELPTEXT bit set, then Explorer is asking for fly-by help. Additionally, if the GCS_UNICODE bit is set, we must return a Unicode string.

The code for our GetCommandString() looks like this:

Hide Shrink Copy Code

```
#include <atlconv.h>  // for ATL string conversion macros

HRESULT CSimpleShlExt::GetCommandString (
  UINT idCmd, UINT uFlags, UINT* pwReserved,
  LPSTR pszName, UINT cchMax )
{
USES_CONVERSION;

  // Check idCmd, it must be 0 since we have only one menu item.
  if ( 0 != idCmd )
    return E_INVALIDARG;

  // If Explorer is asking for a help string, copy our string into the
  // supplied buffer.
  if ( uFlags & GCS_HELPTEXT )
    {
    LPCTSTR szText = _T("This is the simple shell extension's help");

    if ( uFlags & GCS_UNICODE )
      {
      // We need to cast pszName to a Unicode string, and then use the
      // Unicode string copy API.
      lstrcpynW ( (LPWSTR) pszName, T2CW(szText), cchMax );
      }
    else
      {
      // Use the ANSI string copy API to return the help string.
      lstrcpynA ( pszName, T2CA(szText), cchMax );
      }

    return S_OK;
    }

  return E_INVALIDARG;
}
```

Nothing fancy here; I just have the string hard-coded and convert it to the appropriate character set. If you have never used the ATL conversion macros, check out Nish and my article on string wrapper classes; they make life a lot easier when having to pass Unicode strings to COM methods and OLE functions.

One important thing to note is that the lstrcpyn() API guarantees that the destination string will be null-terminated. This is different from the CRT function strncpy(), which does not add a terminating null if the source string's length is greater than or equal to cchMax. I suggest always using lstrcpyn(), so you don't have to insert checks after every strncpy() call to make sure the strings end up null-terminated.

Carrying out the user's selection

The last method in IContextMenu is InvokeCommand(). This method is called if the user clicks on the menu item we added. The prototype for InvokeCommand() is:

```
Hide   Copy Code
HRESULT IContextMenu::InvokeCommand (
  LPCMINVOKECOMMANDINFO pCmdInfo );
```

The CMINVOKECOMMANDINFO struct has a ton of info in it, but for our purposes, we only care about lpVerb and hwnd. lpVerb performs double duty - it can be either the name of the verb that was invoked, or it can be an index telling us which of our menu items was clicked on. hwnd is the handle of the Explorer window where the user invoked our extension; we can use this window as the parent window for any UI that we show.

Since we have just one menu item, we'll check lpVerb, and if it's zero, we know our menu item was clicked. The simplest thing I could think to do is pop up a message box, so that's just what this code does. The message box shows the filename of the selected file, to prove that it's really working.

```
Hide   Copy Code
HRESULT CSimpleShlExt::InvokeCommand (
  LPCMINVOKECOMMANDINFO pCmdInfo )
{
  // If lpVerb really points to a string, ignore this function call and bail out.
  if ( 0 != HIWORD( pCmdInfo->lpVerb ) )
    return E_INVALIDARG;

  // Get the command index - the only valid one is 0.
  switch ( LOWORD( pCmdInfo->lpVerb ) )
    {
    case 0:
      {
      TCHAR szMsg[MAX_PATH + 32];

      wsprintf ( szMsg, _T("The selected file was:\n\n%s"), m_szFile );

      MessageBox ( pCmdInfo->hwnd, szMsg, _T("SimpleShlExt"),
                  MB_ICONINFORMATION );

      return S_OK;
      }
    break;

    default:
      return E_INVALIDARG;
    break;
    }
}
```

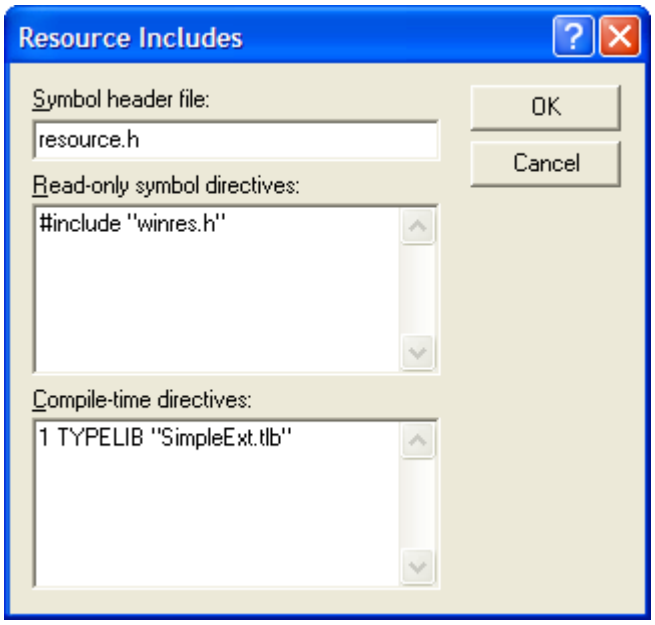
Other code details

There are a couple more tweaks we can make to the wizard-generated code to remove OLE Automation features that we don't need. First, we can remove some registry entries from the *SimpleShlExt.rgs* file (the purpose of this file is explained in the next section):

```
Hide   Copy Code
HKCR
{
  SimpleExt.SimpleShlExt.1 = s 'SimpleShlExt Class'
  {
    CLSID = s '{5E2121EE-0300-11D4-8D3B-444553540000}'
  }
  SimpleExt.SimpleShlExt = s 'SimpleShlExt Class'
  {
    CLSID = s '{5E2121EE-0300-11D4-8D3B-444553540000}'
    CurVer = s 'SimpleExt.SimpleShlExt.1'
  }
  NoRemove CLSID
  {
    ForceRemove {5E2121EE-0300-11D4-8D3B-444553540000} = s 'SimpleShlExt Class'
    {
```

```
    ProgID = s 'SimpleExt.SimpleShlExt.1'
    VersionIndependentProgID = s 'SimpleExt.SimpleShlExt'
    InprocServer32 = s '%MODULE%'
    {
      val ThreadingModel = s 'Apartment'
    }
    'TypeLib' = s '{73738B1C-A43E-47F9-98F0-A07032F2C558}'
  }
 }
}
```

We can also remove the type library from the DLL's resources. Click *View*/*Resource Includes*. In the *Compile-time directives* box, you'll see a line that includes the type library:



Remove that line, then click OK when VC warns about modifying the includes.

In VC 7, the command is in a different location. On the *Resource View* tab, right-click the *SimpleExt.rc* folder, and pick *Resource Includes* on the menu.

Now that we've removed the type library, we need to change two lines of code and tell ATL that it shouldn't do anything with the type library. Open *SimpleExt.cpp*, go to the DllRegisterServer() function, and change the RegisterServer() parameter to FALSE:

```
Hide   Copy Code
STDAPI DllRegisterServer()
{
//...
  return _Module.RegisterServer(TRUE FALSE);
}
```

DllUnregisterServer() needs a similar change:

```
Hide   Copy Code
STDAPI DllUnregisterServer()
{
//...
  return _Module.UnregisterServer(TRUE FALSE);
}
```

Registering the Shell Extension

So now we have all of the COM interfaces implemented. But... how do we get Explorer to use our extension? ATL automatically generates code that registers our DLL as a COM server, but that just lets other apps use our DLL. In order to tell Explorer our extension exists, we need to register it under the key that holds info about text files:

HKEY_CLASSES_ROOT\txtfile

Under that key, a key called ShellEx holds a list of shell extensions that will be invoked on text files. Under ShellEx, the ContextMenuHandlers key holds a list of context menu extensions. Each extension creates a subkey under ContextMenuHandlers and sets the default value of that key to its GUID. So, for our simple extension, we'll create this key:

HKEY_CLASSES_ROOT\txtfile\ShellEx\ContextMenuHandlers\SimpleShlExt

and set the default value to our GUID: "{5E2121EE-0300-11D4-8D3B-444553540000}".

You don't have to write any code to do this, however. If you look at the list of files on the FileView tab, you'll see *SimpleShlExt.rgs*. This is a text file that is parsed by ATL, and tells ATL what registry entries to add when the server is registered, and which ones to delete when the server is unregistered. Here's how we specify the registry entries to add so Explorer knows about our extension:

```
Hide   Copy Code
HKCR
{
 NoRemove txtfile
 {
  NoRemove ShellEx
  {
   NoRemove ContextMenuHandlers
   {
    ForceRemove SimpleShlExt = s '{5E2121EE-0300-11D4-8D3B-444553540000}'
   }
  }
 }
}
```

Each line is a registry key name, with "HKCR" being an abbreviation for HKEY_CLASSES_ROOT. The NoRemove keyword means that the key should not be deleted when the server is unregistered. The last line has another keyword, ForceRemove, which means that if the key exists, it will be deleted before the new key is written. The rest of the line specifies a string (that's what the "s" means) that will be stored in the default value of the SimpleShlExt key.

I need to editorialize a bit here. The key we register our extension under is HKEY_CLASSES_ROOT\txtfile. However, the name "txtfile" isn't a permanent or pre-determined name. If you look in HKEY_CLASSES_ROOT\.txt, the default value of *that* key is where the name is stored. This has a couple of side effects:

- We can't reliably use an RGS script since "txtfile" may not be the correct key name.
- Some other text editor may be installed that associates itself with .TXT files. If it changes the default value of the HKEY_CLASSES_ROOT\.txt key, all existing shell extensions will stop working.

This sure seems like a design flaw to me. I think Microsoft thinks the same way, since recently-created extensions, like the QueryInfo extension, are registered under the key that's named after the file extension.

OK, end of editorial. There's one final registration detail. On NT, it's advisable to put our extension in a list of "approved" extensions. There is a system policy that can be set to prevent extensions from being loaded if they are not on the approved list. The list is stored in:

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

In this key, we create a string value whose name is our GUID. The contents of the string can be anything. The code to do this goes in our DllRegisterServer() and DllUnregisterServer() functions. I won't show the code here, since it's just simple registry access, but you can find the code in the article's sample project.

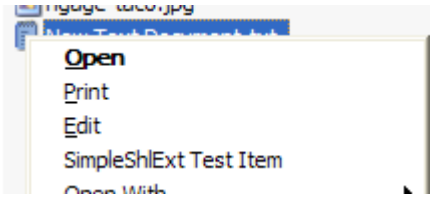
Debugging the shell extension

Eventually, you'll be writing an extension that isn't quite so simple, and you'll have to debug it. Open up your project settings, and on the Debug tab, enter the full path to Explorer in the "Executable for debug session" edit box, for example "C:\windows\explorer.exe". If you're using NT, and you've set the DesktopProcess registry entry described earlier, a new Explorer window will open when you press F5 to start debugging. As long as you do all your work in that window, you'll have no problem rebuilding the DLL later, since when you close that window, your extension will be unloaded.

On Windows 9x, you will have to shut down the shell before running the debugger. Click Start, and then Shut Down. Hold down Ctrl+Alt+Shift and click Cancel. That will shut down Explorer, and you'll see the Taskbar disappear. You can then go back to MSVC and press F5 to start debugging. To stop the debugging session, press Shift+F5 to shut down Explorer. When you're done debugging, you can run Explorer from a command prompt to restart the shell normally.

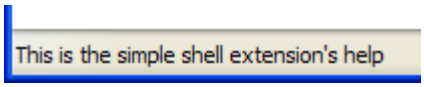
What Does It All Look Like?

Here's what the context menu looks like after we add our item:

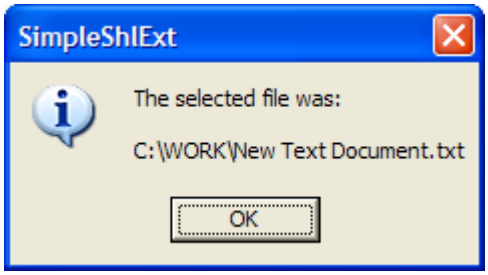


Our menu item is there!

Here's what Explorer's status bar looks like with our fly-by help displayed:



And here's what the message box looks like, showing the name of the file that was selected:



The Complete Idiot's Guide to Writing Shell Extensions - Part II

Michael Dunn, 15 May 2006


4.97 (89 votes)

Rate this: vote 1vote 2vote 3vote 4vote 5

A tutorial on writing a shell extension that operates on multiple files at once.

- Download demo project - 65 K

Contents

Introduction

In Part I of the Guide, I gave an introduction to writing shell extensions, and demonstrated a simple context menu extension that operated on a single file at a time. In Part II, I'll show how to handle multiple files in a single right-click operation. This article's sample extension is a utility that can register and unregister COM servers. It also demonstrates how to use the ATL dialog class CDialogImpl. I will wrap up Part II by explaining some special registry keys that you can use to have your extension invoked on *any* file, not just preselected types.

Part II assumes that you've read Part I so you know the basics of context menu extensions. You should also understand the basics of COM, ATL, and the STL collection classes.

Remember that VC 7 (and probably VC 8) users will need to change some settings before compiling. See the README section in Part I for the details.

Using AppWizard to Get Started

Run the AppWizard and make a new ATL COM program. We'll call it "DllReg". Keep all the default settings in the AppWizard, and click *Finish*. In VC 7, be sure to uncheck the *Attributed* checkbox; we won't be using Attributed ATL in this sample. To add a COM object to the DLL, go to the ClassView tree, right-click the *DllReg classes* item, and pick *New ATL Object*. (In VC 7, right-click the item and pick *Add*|*Add Class*.)

In the ATL Object Wizard, the first panel already has *Simple Object* selected, so just click *Next*. On the second panel, enter "DllRegShlExt" in the *Short Name* edit box (the other edit boxes on the panel will be filled in automatically):

By default, the wizard creates a COM object that can be used from C and script-based clients through OLE Automation. Our extension will only be used by Explorer, so we can change some settings to remove the Automation features. Go to the *Attributes* page, and change the *Interface* type to *Custom*, and change the *Aggregation* setting to *No*:

When you click *OK*, the wizard creates a class called CDLLRegShlExt that contains the basic code for implementing a COM object. We will add our code to this class.

We'll be using the list view control and the STL string and list classes, so add these lines to *stdafx.h* after the existing #include lines that include ATL headers:

Hide Copy Code
```
#include <atlwin.h>
#include <commctrl.h>
#include <string>
#include <list>
typedef std::list< std::basic_string<TCHAR> > string_list;
```

The Initialization Interface

Our IShellExtInit::Initialize() implementation will be quite different from the extension in Part I, for two reasons. First, we will enumerate all of the selected files. Second, we will test the selected files to see if they export registration and unregistration functions. We will consider only those files that export both DllRegisterServer() and DllUnregisterServer(). All other files will be ignored.

We start out just as in Part I, by removing some wizard-generated code and adding the IShellExtInit interface to the C++ class:

Hide Copy Code
```
#include <shlobj.h>
#include <comdef.h>

class ATL_NO_VTABLE CDLLRegShlExt :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CDLLRegShlExt, &CLSID_DllRegShlExt>,
  public IDllRegShlExt,
  public IShellExtInit
{
  BEGIN_COM_MAP(CDLLRegShlExt)
    COM_INTERFACE_ENTRY(IDllRegShlExt)
    COM_INTERFACE_ENTRY(IShellExtInit)
  END_COM_MAP()
```

Our CDLLRegShlExt class will also need a few member variables:

Hide Copy Code
```
protected:
  HBITMAP     m_hRegBmp;
  HBITMAP     m_hUnregBmp;
  string_list m_lsFiles;
```

The CDLLRegShlExt constructor loads two bitmaps for use in the context menu:

Hide Copy Code
```
CDLLRegShlExt::CDLLRegShlExt()
{
  m_hRegBmp = LoadBitmap ( _Module.GetModuleInstance(),
                      MAKEINTRESOURCE(IDB_REGISTERBMP) );

  m_hUnregBmp = LoadBitmap ( _Module.GetModuleInstance(),
                        MAKEINTRESOURCE(IDB_UNREGISTERBMP) );
}
```

Now we're ready to write the Initialize() function. Initialize() will perform these steps:

1. Change the current directory to the directory being viewed in the Explorer window.
2. Enumerate all of the files that were selected.
3. For each DLL and OCX file, try to load it with LoadLibrary().
4. If LoadLibrary() succeeded, see if the file exports DllRegisterServer() and DllUnregisterServer().
5. If both exports are found, add the filename to our list of files we can operate on, m_lsFiles.

Hide Copy Code
```
HRESULT CDLLRegShlExt::Initialize (
  LPCITEMIDLIST pidlFolder, LPDATAOBJECT pDataObj, HKEY hProgID )
{
UINT      uNumFiles;
HDROP     hdrop;
FORMATETC etc = { CF_HDROP, NULL, DVASPECT_CONTENT,
                  -1, TYMED_HGLOBAL };
STGMEDIUM stg = { TYMED_HGLOBAL };
HINSTANCE hinst;
HRESULT (STDAPICALLTYPE* pfn)();
```

Tons of boring local variables! The first step is to get an HDROP from the pDataObj passed in. This is done just like in the Part I extension.

Hide Copy Code
```
// Read the list of folders from the data object.  They're stored in HDROP
// format, so just get the HDROP handle and then use the drag 'n' drop APIs
// on it.
if ( FAILED( pDO->GetData ( &etc, &stg ) ))
    return E_INVALIDARG;

// Get an HDROP handle.
hdrop = (HDROP) GlobalLock ( stg.hGlobal );

if ( NULL == hdrop )
  {
  ReleaseStgMedium ( &stg );
  return E_INVALIDARG;
  }

// Determine how many files are involved in this operation.
uNumFiles = DragQueryFile ( hdrop, 0xFFFFFFFF, NULL, 0 );
```

Next comes a for loop that gets the next filename (using DragQueryFile()) and tries to load it with LoadLibrary(). The real shell extension in the sample project does some directory-changing beforehand, which I have omitted here since it's a bit long.

Hide Copy Code
```
for ( UINT uFile = 0; uFile < uNumFiles; uFile++ )
  {
  // Get the next filename.
  if ( 0 == DragQueryFile ( hdrop, uFile, szFile, MAX_PATH ) )
    continue;

  // Try & load the DLL.
  hinst = LoadLibrary ( szFile );

  if ( NULL == hinst )
    continue;
```

Next, we'll see if the module exports the two required functions.

Hide Copy Code
```
// Get the address of DllRegisterServer();
(FARPROC&) pfn = GetProcAddress ( hinst, "DllRegisterServer" );

// If it wasn't found, skip the file.
if ( NULL == pfn )
```

```
 {
 FreeLibrary ( hinst );
 continue;
 }

// Get the address of DllUnregisterServer();
(FARPROC&) pfn = GetProcAddress ( hinst, "DllUnregisterServer" );

// If it was found, we can operate on the file, so add it to
// our list of files (m_lsFiles).
if ( NULL != pfn )
  m_lsFiles.push_back ( szFile );

FreeLibrary ( hinst );
}   // end for
```

If both exported functions are present in the module, the filename is added to m_lsFiles, which is an STL list collection that holds strings. That list will be used later, when we iterate over all the files and register or unregister them.

The last thing to do in Initialize() is free resources and return the right value back to Explorer.

```
Hide   Copy Code
  // Release resources.
  GlobalUnlock ( stg.hGlobal );
  ReleaseStgMedium ( &stg );

  // If we found any files we can work with, return S_OK.  Otherwise,
  // return E_INVALIDARG so we don't get called again for this right-click
  // operation.
  return (m_lsFiles.size() > 0) ? S_OK : E_INVALIDARG;
}
```

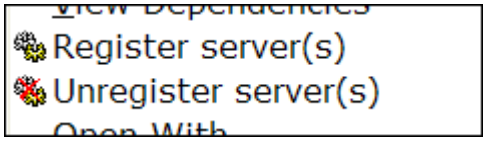
If you take a look at the sample project's code, you'll see that I have to figure out which directory is being viewed by looking at the names of the files. You might wonder why I don't just use the pidlFolder parameter, which is documented as "the item identifier list for the folder that contains the item whose context menu is being displayed." Well, during my testing, this parameter was always NULL, so it's useless.

Adding Our Menu Items

Next up are the IContextMenu methods. As before, we add IContextMenu to the list of interfaces that CDLLRegShlExt implements, by adding the lines in bold:

```
Hide   Copy Code
class ATL_NO_VTABLE CDLLRegShlExt :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CDLLRegShlExt, &CLSID_DllRegShlExt>,
  public IShellExtInit,
  public IContextMenu
{
  BEGIN_COM_MAP(CDLLRegShlExt)
    COM_INTERFACE_ENTRY(IShellExtInit)
    COM_INTERFACE_ENTRY(IContextMenu)
  END_COM_MAP()
```

We'll add two items to the menu, one to register the selected files, and another to unregister them. The items look like this:



Our QueryContextMenu() implementation starts out like in Part I. We check uFlags, and return immediately if the CMF_DEFAULTONLY flag is present.

```
Hide   Copy Code
```

```
HRESULT CDLLRegShlExt::QueryContextMenu (
  HMENU hmenu, UINT uMenuIndex, UINT uidFirstCmd,
  UINT uidLastCmd, UINT uFlags )
{
UINT uCmdID = uidFirstCmd;

  // If the flags include CMF_DEFAULTONLY then we shouldn't do anything.
  if ( uFlags & CMF_DEFAULTONLY )
    return MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_NULL, 0);
```

Next up, we add the "Register servers" menu item. There's something new here: we set a bitmap for the item. This is the same thing that WinZip does to have the little folder-in-a-vice icon appear next to its own menu items.

```
Hide   Copy Code
// Add our register/unregister items.
InsertMenu ( hmenu, uMenuIndex, MF_STRING | MF_BYPOSITION, uCmdID++,
            _T("Register server(s)") );

// Set the bitmap for the register item.
if ( NULL != m_hRegBmp )
  SetMenuItemBitmaps ( hmenu, uMenuIndex, MF_BYPOSITION, m_hRegBmp, NULL );

  uMenuIndex++;
```

The SetMenuItemBitmaps() API is how we show our little gears icon next to the "Register servers" item. Note that uCmdID is incremented, so that the next time we call InsertMenu(), the command ID will be one more than the previous value. At the end of this step, uMenuIndex is incremented so our second item will appear after the first one.

And speaking of the second menu item, we add that next. It's almost identical to the code for the first item.

```
Hide   Copy Code
InsertMenu ( hmenu, uMenuIndex, MF_STRING | MF_BYPOSITION, uCmdID++,
            _T("Unregister server(s)") );

// Set the bitmap for the unregister item.
if ( NULL != m_hUnregBmp )
  SetMenuItemBitmaps ( hmenu, uMenuIndex, MF_BYPOSITION, m_hUnregBmp, NULL );
```

And at the end, we tell Explorer how many items we added.

```
Hide   Copy Code
  return MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_NULL, 2);
}
```

Providing Fly-By Help and a Verb

As before, the GetCommandString() method is called when Explorer needs to show fly-by help or get a verb for one of our commands. This extension is different than the last one in that we have 2 menu items, so we need to examine the uCmdID parameter to tell which item Explorer is calling us about.

```
Hide   Copy Code
#include <atlconv.h>

HRESULT CDLLRegShlExt::GetCommandString (
  UINT uCmdID, UINT uFlags, UINT* puReserved,
  LPSTR szName, UINT cchMax )
{
USES_CONVERSION;
LPCTSTR szPrompt;

  if ( uFlags & GCS_HELPTEXT )
    {
    switch ( uCmdID )
      {
      case 0:
        szPrompt = _T("Register all selected COM servers");
```

```
      break;

      case 1:
        szPrompt = _T("Unregister all selected COM servers");
      break;

      default:
        return E_INVALIDARG;
      break;
      }
```

If uCmdID is 0, then we are being called for our first item (register). If it's 1, then we're being called for the second item (unregister). After we determine the help string, we copy it into the supplied buffer, converting to Unicode first if necessary.

```
Hide   Copy Code
    // Copy the help text into the supplied buffer.  If the shell wants
    // a Unicode string, we need to case szName to an LPCWSTR.
    if ( uFlags & GCS_UNICODE )
      lstrcpynW ( (LPWSTR) szName, T2CW(szPrompt), cchMax );
    else
      lstrcpynA ( szName, T2CA(szPrompt), cchMax );
  }

  return S_OK;
}
```

For this extension, I also wrote code that provides a verb. However, during my testing, Explorer never called GetCommandString() to get a verb. I even wrote a test app that called ShellExecute() on a DLL and tried to use a verb, but that didn't work either. I have omitted the verb-related code here, but you can check it out in the sample project if you're interested.

Carrying Out The User's Selection

When the user clicks one of our menu items, Explorer calls our InvokeCommand() method. InvokeCommand() first checks the high word of lpVerb. If it's non-zero, then it is the name of the verb that was invoked. Since we know verbs aren't working properly (at least on Win 98), we'll bail out. Otherwise, if the low word of lpVerb is 0 or 1, we know one of our two menu items was clicked.

```
Hide   Copy Code
HRESULT CDLLRegShlExt::InvokeCommand ( LPCMINVOKECOMMANDINFO pCmdInfo )
{
  // If lpVerb really points to a string, ignore this function call and bail out.
  if ( 0 != HIWORD( pInfo->lpVerb ))
    return E_INVALIDARG;

  // Check that lpVerb is one of our commands (0 or 1)
  switch ( LOWORD( pInfo->lpVerb ))
    {
    case 0:
    case 1:
      {
      CProgressDlg dlg ( &m_lsFiles, pInfo );

      dlg.DoModal();
      return S_OK;
      }
    break;

    default:
      return E_INVALIDARG;
    break;
    }
}
```

If lpVerb is 0 or 1, we create a progress dialog (which is derived from the ATL class CDialogImpl), and pass it the list of filenames.

All of the real work happens in the CProgressDlg class. Its OnInitDialog() function initializes the list control, and then calls CProgressDlg::DoWork(). DoWork() iterates over the string list that was built in CDLLRegShlExt::Initialize(), and calls the appropriate

function in each file. The basic code is below; it is not complete, since for clarity I've left out the error-checking and the parts that fill the list control. It's just enough to demonstrate how to iterate over the list of filenames and act on each one.

```
Hide   Shrink ▲   Copy Code
void CProgressDlg::DoWork()
{
HRESULT (STDAPICALLTYPE* pfn)();
string_list::const_iterator it;
HINSTANCE hinst;
LPCSTR    pszFnName;
HRESULT   hr;
WORD      wCmd;

  wCmd = LOWORD ( m_pCmdInfo->lpVerb );

  // We only support 2 commands, so check the value passed in lpVerb.
  if ( wCmd > 1 )
    return;

  // Determine which function we'll be calling.  Note that these strings are
  // not enclosed in the _T macro, since GetProcAddress() only takes an
  // ANSI string for the function name.
  pszFnName = wCmd ? "DllUnregisterServer" : "DllRegisterServer";

  for (it = m_pFileList->begin(); it != m_pFileList->end(); it++)
    {
    // Try to load the next file.
    hinst = LoadLibrary ( it->c_str() );

    if ( NULL == hinst )
      continue;

    // Get the address of the register/unregister function.
    (FARPROC&) pfn = GetProcAddress ( hinst, pszFnName );

    // If it wasn't found, go on to the next file.
    if ( NULL == pfn )
      continue;

    // Call the function!
    hr = pfn();

    // Omitted: error handling and checks on the return
    // value of the function we called.
    }  // end for
```

The remainder of DoWork() is cleanup and error handling. You can find the complete code in *ProgressDlg.cpp* in the sample project.

Registering the Shell Extension

The DllReg extension operates on in-proc COM servers, so it should be invoked on DLL and OCX files. As in Part I, we can do this through the RGS script, DllRegShlExt.rgs. Here's the necessary script to register our DLL as a context menu handler for each of those extensions.

```
Hide   Copy Code
HKCR
{
  NoRemove dllfile
  {
    NoRemove shellex
    {
      NoRemove ContextMenuHandlers
      {
        ForceRemove DLLRegSvr = s '{8AB81E72-CB2F-11D3-8D3B-AC2F34F1FA3C}'
      }
    }
  }
```

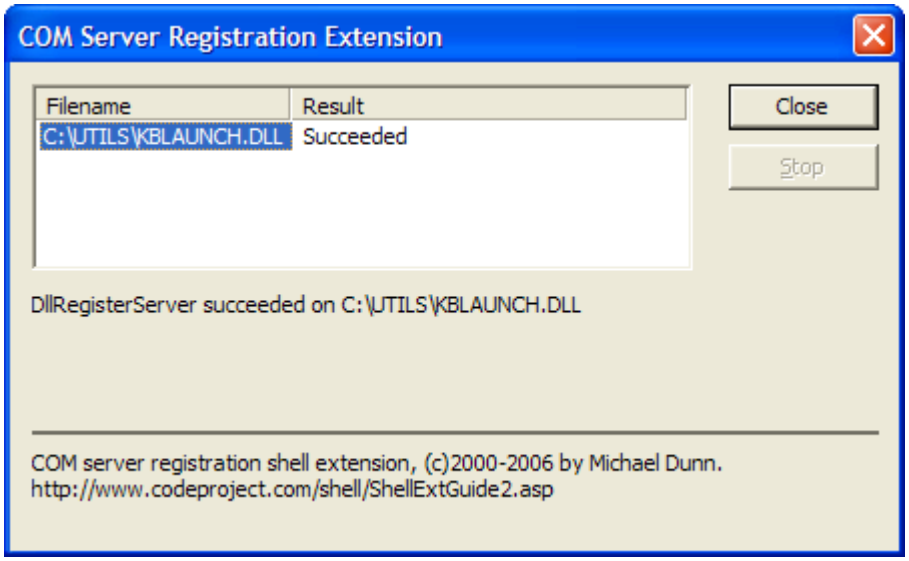
```
 NoRemove ocxfile
 {
  NoRemove shellex
  {
   NoRemove ContextMenuHandlers
   {
    ForceRemove DLLRegSvr = s '{8AB81E72-CB2F-11D3-8D3B-AC2F34F1FA3C}'
   }
  }
 }
}
```

The syntax of the RGS file, and the keywords NoRemove and ForceRemove are explained in Part I, in case you need a refresher on their meaning.

As in our previous extension, on NT-based OSes, we need to add our extension to the list of "approved" extensions. The code to do this is in the DllRegisterServer() and DllUnregisterServer() functions. I won't show the code here, since it's just simple registry access, but you can find the code in the sample project.

The Extension In Action

When you click one of our menu items, the progress dialog is displayed and shows the results of the operations:



The list control shows the name of each file, and whether the function call succeeded or not. When you select a file, a message is shown beneath the list that gives more details, along with a description of the error if the function call failed.

Notice that in the above screen shot, the dialog isn't using the XP theme. As described in the MSDN article "Using Windows XP Visual Styles", there are two things we need to do to make our UI themed. The first is to tell the OS to use common controls version 6 for our DLL, by putting a manifest in the resources. You can copy the necessary manifest XML from the above MSDN article and save it to a file called *dllreg.manifest* in the project's *res* subdirectory. Next, add this line to the resource includes:

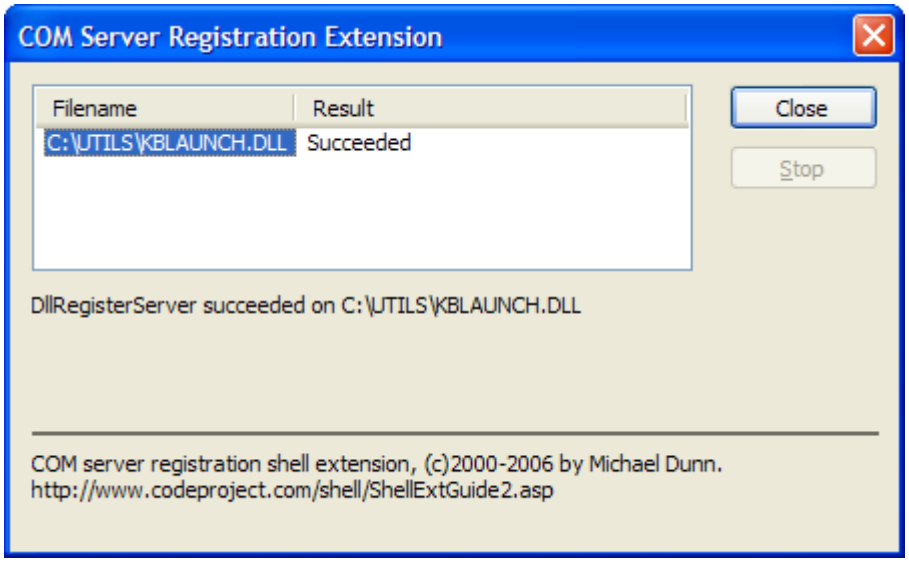
Hide Copy Code
```
ISOLATIONAWARE_MANIFEST_RESOURCE_ID RT_MANIFEST "res\\dllreg.manifest"
```

Then in *stdafx.h*, add this line before all includes:

Hide Copy Code
```
#define ISOLATION_AWARE_ENABLED 1
```

As of May, 2006, the MSDN article says the symbol is called SIDEBYSIDE_COMMONCONTROLS, but in my SDKs, only ISOLATION_AWARE_ENABLED is used. If you have a newer SDK and ISOLATION_AWARE_ENABLED doesn't work for you, try SIDEBYSIDE_COMMONCONTROLS.

After making these changes and rebuilding, the dialog now follows the active theme:



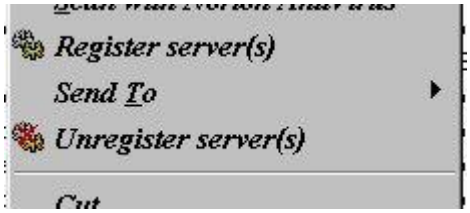
Other Ways to Register The Extension

So far, our extensions have been invoked only for certain file types. It's possible to have the shell call our extension for *any* file by registering as a context menu handler under the HKCR* key:

```
Hide   Copy Code
HKCR
{
  NoRemove *
  {
    NoRemove shellex
    {
      NoRemove ContextMenuHandlers
      {
        ForceRemove DLLRegSvr = s '{8AB81E72-CB2F-11D3-8D3B-AC2F34F1FA3C}'
      }
    }
  }
}
```

The HKCR* key lists shell extensions that are called for all files. Note that the docs say that the extensions are also invoked for any shell object (meaning files, directories, virtual folders, Control Panel items, etc.), but that was not the behavior I saw during my testing. The extension was only invoked for files in the file system.

In shell version 4.71+, there is also a key called HKCR\AllFileSystemObjects. If we register under this key, our extension is invoked for all files and directories in the file system, except root directories. (Extensions that are invoked for root directories are registered under HKCR\Drive.) However, on some versions of Windows, you get some strange behavior when registering under this key. For example, on Windows 98, the DllReg menu items ended up being mixed in with the *Send To* item:



This wasn't a problem on XP.

You can also write a context menu extension that operates on directories. For an example of such an extension, check out my article A Utility to Clean Up Compiler Temp Files.

Finally, in shell version 4.71+, you can have a context menu extension invoked when the user right-clicks the background of an Explorer window that's viewing a directory (including the desktop). To have your extension invoked like this, register it under HKCR\Directory\Background\shellex\ContextMenuHandlers. Using this method, you can add your own menu items to the desktop

context menu, or the menu for any other directory. The parameters passed to IShellExtInit::Initialize() are a bit different, though, so I may cover this topic in a future article.

22

The Complete Idiot's Guide to Writing Shell Extensions - Part V

Michael Dunn, 24 May 2006

4.96 (58 votes)

Rate this: vote 1vote 2vote 3vote 4vote 5

A tutorial on writing a shell extension that adds pages to the properties dialog of files.

- Download demo project - 60K

Contents

Introduction

Here in Part V of the Guide, we'll venture into the world of property sheets. When you bring up the properties for a file system object, Explorer shows a property sheet with a tab labeled *General*. The shell lets us add pages to the property sheet by using a type of shell extension called a **property sheet handler**.

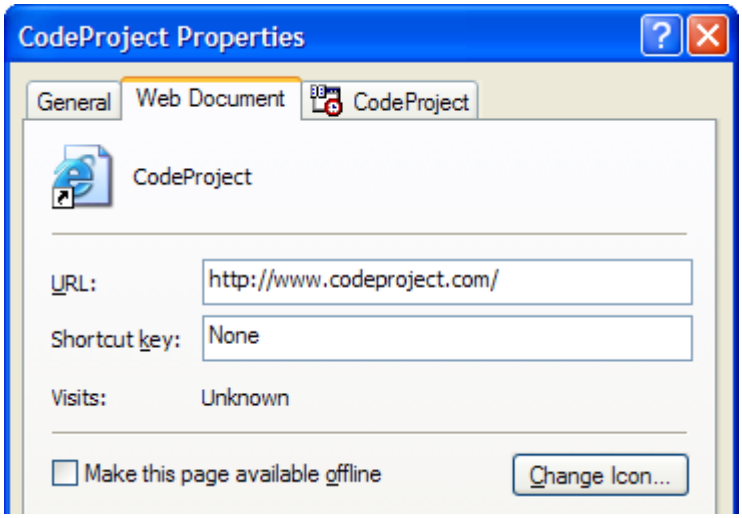
This article assumes that you understand the basics of shell extensions, and are familiar with the STL collection classes. If you need a refresher on STL, you should read Part II, since the same techniques will be used in this article.

Remember that VC 7 (and probably VC 8) users will need to change some settings before compiling. See the README section in Part I for the details.

Everyone is familiar with Explorer's properties dialogs. More specifically, they are **property sheets** that contain one or more pages. Each property sheet has a *General* tab that lists the full path, modified date, and other various stuff. Explorer lets us add our own pages to the property sheets, using a **property sheet handler** extension. A property sheet handler can also add or replace pages in certain Control Panel applets, but that topic will not be covered here. See my article Adding Custom Pages to Control Panel Applets to learn more about extending applets.

This article presents an extension that lets you modify the created, accessed, and modified times for a file right from its properties dialog. I will do all property page handling in straight SDK calls, without MFC or ATL. I haven't tried using an MFC or WTL property page object in an extension; doing so may be tricky because the shell expects to receive a handle to the sheet (an HPROPSHEETPAGE), and MFC hides this detail in the CPropertyPage implementation.

If you bring up the properties for a .URL file (an Internet shortcut), you can see property sheet handlers in action. The *CodeProject* tab is a sneak peek at this article's extension. The *Web Document* tab shows an extension installed by IE.



The Initialization Interface

You should be familiar with the set-up steps now, so I'll skip the instructions for going through the VC wizards. If you're following along in the wizards, make a new ATL COM app called *FileTime*, with a C++ implementation class CFileTimeShlExt.

Since a property sheet handler operates on all selected files at once, it uses IShellExtInit as its initialization interface. We'll need to add IShellExtInit to the list of interfaces that CFileTimeShlExt implements. Again, this should be familiar to you, so I will not repeat the steps here.

The class will also need a list of strings to hold the names of the selected files.

Hide Copy Code
```
typedef list< basic_string<TCHAR> > string_list;

protected:
  string_list m_lsFiles;
```

The Initialize() method will do the same thing as Part II - read in the names of the selected file and store them in the string list. Here's the beginning of the function:

Hide Copy Code
```
STDMETHODIMP CFileTimeShlExt::Initialize (
  LPCITEMIDLIST pidlFolder, LPDATAOBJECT pDataObj,
  HKEY hProgID )
{
TCHAR     szFile[MAX_PATH];
UINT      uNumFiles;
HDROP     hdrop;
FORMATETC etc = { CF_HDROP, NULL, DVASPECT_CONTENT, -1, TYMED_HGLOBAL };
STGMEDIUM stg;
INITCOMMONCONTROLSEX iccex = { sizeof(INITCOMMONCONTROLSEX), ICC_DATE_CLASSES };

  // Init the common controls.
  InitCommonControlsEx ( &iccex );
```

We initialize the common controls because our page will use the date/time picker (DTP) control. Next we do all the mucking about with the IDataObject interface and get an HDROP handle for enumerating the selected files.

Hide Copy Code
```
// Read the list of items from the data object.  They're stored in HDROP
// form, so just get the HDROP handle and then use the drag 'n' drop APIs
// on it.
if ( FAILED( pDataObj->GetData ( &etc, &stg ) ))
  return E_INVALIDARG;

// Get an HDROP handle.
hdrop = (HDROP) GlobalLock ( stg.hGlobal );
```

```
if ( NULL == hdrop )
  {
  ReleaseStgMedium ( &stg );
  return E_INVALIDARG;
  }

// Determine how many files are involved in this operation.
uNumFiles = DragQueryFile ( hdrop, 0xFFFFFFFF, NULL, 0 );
```

Next comes the loop that actually enumerates through the selected files. This extension will only operate on files, not directories, so any directories we come across are ignored.

```
Hide   Copy Code
for ( UINT uFile = 0; uFile < uNumFiles; uFile++ )
  {
  // Get the next filename.
  if ( 0 == DragQueryFile ( hdrop, uFile, szFile, MAX_PATH ) )
    continue;

  // Skip over directories.  We *could* handle directories, since they
  // keep the creation time/date, but I'm just choosing not to do so
  // in this example.
  if ( PathIsDirectory ( szFile ) )
    continue;

  // Add the filename to our list of files to act on.
  m_lsFiles.push_back ( szFile );
  }   // end for

// Release resources.
GlobalUnlock ( stg.hGlobal );
ReleaseStgMedium ( &stg );
```

The code that enumerates the filenames is the same as before, but there's also something new here. A property sheet has a limit on the number of pages it can have, defined as the constant MAXPROPPAGES in *prsht.h*. Each file will get its own page, so if our list has more than MAXPROPPAGES files, it gets truncated so its size is MAXPROPPAGES. (Even though MAXPROPPAGES is currently 100, the property sheet will not display that many tabs. It maxes out at around 34.)

```
Hide   Copy Code
  // Check how many files were selected.  If the number is greater than the
  // maximum number of property pages, truncate our list.
  if ( m_lsFiles.size() > MAXPROPPAGES )
    m_lsFiles.resize ( MAXPROPPAGES );

  // If we found any files we can work with, return S_OK.  Otherwise,
  // return E_FAIL so we don't get called again for this right-click
  // operation.
  return (m_lsFiles.size() > 0) ? S_OK : E_FAIL;
}
```

Adding Property Pages

If Initialize() returns S_OK, Explorer queries for a new interface, IShellPropSheetExt. IShellPropSheetExt is quite simple, with only one method that requires an implementation. To add IShellPropSheetExt to our class, open *FileTimeShlExt.h* and add the lines listed here in bold:

```
Hide   Copy Code
class CFileTimeShlExt :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CFileTimeShlExt, &CLSID_FileTimeShlExt>,
  public IShellExtInit,
  public IShellPropSheetExt
{
  BEGIN_COM_MAP(CFileTimeShlExt)
    COM_INTERFACE_ENTRY(IShellExtInit)
<FONT COLOR="red">    </FONT>COM_INTERFACE_ENTRY(IShellPropSheetExt)
  END_COM_MAP()
```

```
public:
<FONT COLOR="red">  </FONT>// IShellPropSheetExt
  STDMETHODIMP AddPages(LPFNADDPROPSHEETPAGE, LPARAM);
  STDMETHODIMP ReplacePage(UINT, LPFNADDPROPSHEETPAGE, LPARAM)
     { return E_NOTIMPL; }
```

The AddPages() method is the one we'll implement. ReplacePage() is only used by extensions that replace pages in Control Panel applets, so we do not need to implement it here. Explorer calls our AddPages() function to let us add pages to the property sheet that Explorer sets up.

The parameters to AddPages() are a function pointer and an LPARAM, both of which are used only by the shell. lpfnAddPageProc points to a function inside the shell that we call to actually add the pages. lParam is some mysterious value that's important to the shell. We don't mess with it, we just pass it right back to the lpfnAddPageProc function.

Hide Copy Code

```
STDMETHODIMP CFileTimeShlExt::AddPages (
  LPFNADDPROPSHEETPAGE lpfnAddPageProc,
  LPARAM lParam )
{
PROPSHEETPAGE  psp;
HPROPSHEETPAGE hPage;
TCHAR          szPageTitle [MAX_PATH];
string_list::const_iterator it, itEnd;

  for ( it = m_lsFiles.begin(), itEnd = m_lsFiles.end();
        it != itEnd; it++ )
    {
    // 'it' points at the next filename. Allocate a new copy of the string
    // that the page will own.
    LPCTSTR szFile = _tcsdup ( it->c_str() );
```

The first thing we do is make a copy of the filename. The reason for this is explained below.

The next step is to create a string to go in our page's tab. The string will be the filename, without the extension. Additionally, the string will be truncated if it's longer than 24 characters. This is totally arbitrary; I chose 24 because it looked good to me. There should be *some* limit, to prevent the name from running off the end of the tab.

Hide Copy Code

```
// Strip the path and extension from the filename - this will be the
// page title.  The name is truncated at 24 chars so it fits on the tab.
lstrcpyn ( szPageTitle, it->c_str(), MAX_PATH );
PathStripPath ( szPageTitle );
PathRemoveExtension ( szPageTitle );
szPageTitle[24] = '\0';
```

Since we're using straight SDK calls to do the property page, we'll have to get our hands dirty with a PROPSHEETPAGE struct. Here's the setup for the struct:

Hide Copy Code

```
psp.dwSize      = sizeof(PROPSHEETPAGE);
psp.dwFlags     = PSP_USEREFPARENT | PSP_USETITLE |
                  PSP_USEICONID | PSP_USECALLBACK;
psp.hInstance   = _Module.GetResourceInstance();
psp.pszTemplate = MAKEINTRESOURCE(IDD_FILETIME_PROPPAGE);
psp.pszIcon     = MAKEINTRESOURCE(IDI_TAB_ICON);
psp.pszTitle    = szPageTitle;
psp.pfnDlgProc  = PropPageDlgProc;
psp.lParam      = (LPARAM) szFile;
psp.pfnCallback = PropPageCallbackProc;
psp.pcRefParent = (UINT*) &_Module.m_nLockCnt;
```

There are a few important details here that we must pay attention to for the extension to work correctly:

1. The pszIcon member is set to the resource ID of a 16x16 icon, which will be displayed in the tab. Having an icon is optional, of course, but I added an icon to make our page stand out.

2. The pfnDlgProc member is set to the address of the dialog proc of our page.
3. The lParam member is set to szFile, which is a copy of the filename the page is associated with.
4. The pfnCallback member is set to the address of a callback function that gets called when the page is created and destroyed. The role of this function will be explained later.
5. The pcRefParent member is set to the address of a member variable inherited from CComModule. This variable is the lock count of the DLL. The shell increments this count when the property sheet is displayed, to keep our DLL in memory while the sheet is open. The count will be decremented after the sheet is destroyed.

Having set up that struct, we call the API to create the property page.

```
Hide   Copy Code
hPage = CreatePropertySheetPage ( &psp );
```

If that succeeds, we call the shell's callback function which adds the newly-created page to the property sheet. The callback returns a BOOL indicating success or failure. If it fails, we destroy the page.

```
Hide   Copy Code
   if ( NULL != hPage )
     {
     // Call the shell's callback function, so it adds the page to
     // the property sheet.
     if ( !lpfnAddPageProc ( hPage, lParam ) )
       DestroyPropertySheetPage ( hPage );
     }
   }   // end for

  return S_OK;
}
```

A Sticky Situation With Lifetimes of Objects

Time to deliver on my promise to explain about the duplicate string. The duplicate is needed because after AddPages() returns, the shell releases its IShellPropSheetExt interface, which in turn destroys the CFileTimeShlExt object. That means that the property page's dialog proc can't access the m_lsFiles member of CFileTimeShlExt.

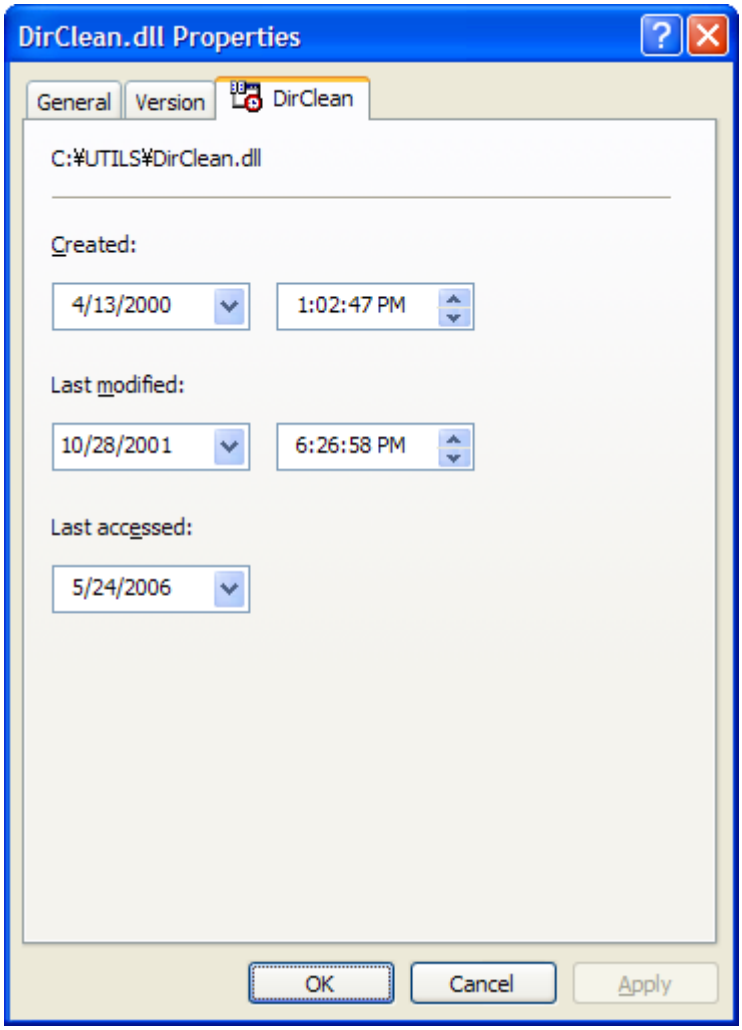
My solution was to make a copy of each filename, and pass a pointer to that copy to the page. The page owns that memory, and is responsible for freeing it. If there is more than one selected file, each page gets a copy of the filename it is associated with. The memory is freed in the PropPageCallbackProc function, shown later. This line in AddPages():

```
Hide   Copy Code
psp.lParam = (LPARAM) szFile;
```

is the important one. It stores the pointer in the PROPSHEETPAGE struct, and makes it available to the page's dialog proc.

The Property Page Callback Functions

Now, on to the property page itself. Here's what the new page looks like. Keep this picture in mind while you're reading over the explanation of how the page works.



Notice there is no last accessed time control. FAT only keeps the last accessed date. Other file systems keep the time, but I have not implemented logic to check the file system. The time will always be stored as 12 midnight if the file system supports the last accessed time field.

The page has two callback functions and two message handlers. These prototypes go at the top of *FileTimeShlExt.cpp*:

Hide Copy Code
```
BOOL CALLBACK PropPageDlgProc ( HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam );
UINT CALLBACK PropPageCallbackProc ( HWND hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp );
BOOL OnInitDialog ( HWND hwnd, LPARAM lParam );
BOOL OnApply ( HWND hwnd, PSHNOTIFY* phdr );
```

The dialog proc is pretty simple. It handles three messages: WM_INITDIALOG, PSN_APPLY, and DTN_DATETIMECHANGE. Here's the WM_INITDIALOG part:

Hide Copy Code
```
BOOL CALLBACK PropPageDlgProc ( HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
BOOL bRet = FALSE;

  switch ( uMsg )
    {
    case WM_INITDIALOG:
      bRet = OnInitDialog ( hwnd, lParam );
    break;
```

OnInitDialog() is explained later. Next up is PSN_APPLY, which is sent if the user clicks the OK or Apply button.

Hide Copy Code
```
case WM_NOTIFY:
  {
```

```
  NMHDR* phdr = (NMHDR*) lParam;

  switch ( phdr->code )
    {
    case PSN_APPLY:
      bRet = OnApply ( hwnd, (PSHNOTIFY*) phdr );
    break;
```

And finally, DTN_DATETIMECHANGE. This one is simple - we just enable the Apply button by sending a message to the property sheet (which is the parent window of our page).

```
Hide   Copy Code
        case DTN_DATETIMECHANGE:
          // If the user changes any of the DTP controls, enable
          // the Apply button.
          SendMessage ( GetParent(hwnd), PSM_CHANGED, (WPARAM) hwnd, 0 );
        break;
        }  // end switch
      }  // end case WM_NOTIFY
    break;
    }  // end switch

  return bRet;
}
```

So far, so good. The other callback function is called when the page is created or destroyed. We only care about the latter case, since it's when we can free the duplicate string that was created back in AddPages(). The ppsp parameter points at the PROPSHEETPAGE struct used to create the page, and the lParam member still points at the duplicate string which must be freed.

```
Hide   Copy Code
UINT CALLBACK PropPageCallbackProc ( HWND hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp )
{
  if ( PSPCB_RELEASE == uMsg )
    free ( (void*) ppsp->lParam );

  return 1;
}
```

The function always returns 1 because when the function is called during the creation of the page, it can prevent the page from being created by returning 0. Returning 1 lets the page be created normally. The return value is ignored when the function is called when the page is destroyed.

The Property Page Message Handlers

A lot of important stuff happens in OnInitDialog(). The lParam parameter again points to the PROPSHEETPAGE struct used to create this page. *Its* lParam member points to that ever-present filename. Since we need to have access to that filename in the OnApply() function, we save the pointer using SetWindowLong().

```
Hide   Copy Code
BOOL OnInitDialog ( HWND hwnd, LPARAM lParam )
{
PROPSHEETPAGE*  ppsp = (PROPSHEETPAGE*) lParam;
LPCTSTR         szFile = (LPCTSTR) ppsp->lParam;
HANDLE          hFind;
WIN32_FIND_DATA rFind;

  // Store the filename in this window's user data area, for later use.
  SetWindowLong ( hwnd, GWL_USERDATA, (LONG) szFile );
```

Next, we get the file's created, modified, and accessed times using FindFirstFile(). If that succeeds, the DTP controls are initialized with the right data.

```
Hide   Copy Code
hFind = FindFirstFile ( szFile, &rFind );

if ( INVALID_HANDLE_VALUE != hFind )
```

```
 {
 // Initialize the DTP controls.
 SetDTPCtrl ( hwnd, IDC_MODIFIED_DATE, IDC_MODIFIED_TIME,
          &rFind.ftLastWriteTime );

 SetDTPCtrl ( hwnd, IDC_ACCESSED_DATE, 0,
          &rFind.ftLastAccessTime );

 SetDTPCtrl ( hwnd, IDC_CREATED_DATE, IDC_CREATED_TIME,
          &rFind.ftCreationTime );

 FindClose ( hFind );
 }
```

SetDTPCtrl() is a utility function that sets the contents of the DTP controls. You can find the code at the end of *FileTimeShlExt.cpp*.

As an added touch, the full path to the file is shown in the static control at the top of the page.

```
Hide   Copy Code
 PathSetDlgItemPath ( hwnd, IDC_FILENAME, szFile );
 return FALSE;
}
```

The OnApply() handler does the opposite - it reads the DTP controls and modifies the file's created, modified, and accessed times. The first step is to retrieve the filename pointer by using GetWindowLong() and open the file for writing.

```
Hide   Copy Code
BOOL OnApply ( HWND hwnd, PSHNOTIFY* phdr )
{
LPCTSTR  szFile = (LPCTSTR) GetWindowLong ( hwnd, GWL_USERDATA );
HANDLE   hFile;
FILETIME ftModified, ftAccessed, ftCreated;

 // Open the file.
 hFile = CreateFile ( szFile, GENERIC_WRITE, FILE_SHARE_READ, NULL,
                  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
```

If we can open the file, we read the DTP controls and write the times back to the file. ReadDTPCtrl() is the counterpart of SetDTPCtrl().

```
Hide   Copy Code
 if ( INVALID_HANDLE_VALUE != hFile )
   {
   // Retrieve the dates/times from the DTP controls.
   ReadDTPCtrl ( hwnd, IDC_MODIFIED_DATE, IDC_MODIFIED_TIME, &ftModified );
   ReadDTPCtrl ( hwnd, IDC_ACCESSED_DATE, 0, &ftAccessed );
   ReadDTPCtrl ( hwnd, IDC_CREATED_DATE, IDC_CREATED_TIME, &ftCreated );

   // Change the file's created, accessed, and last modified times.
   SetFileTime ( hFile, &ftCreated, &ftAccessed, &ftModified );
   CloseHandle ( hFile );
   }
 else
   // <<Error handling omitted>>

 // Return PSNRET_NOERROR to allow the sheet to close if the user clicked OK.
 SetWindowLong ( hwnd, DWL_MSGRESULT, PSNRET_NOERROR );
 return TRUE;
}
```

Registering the Shell Extension

Registering a drag and drop handler is similar to registering a context menu extension. The handler can be invoked for a particular file type, for example all text files. This extension works on *any* file, so we register it under the HKEY_CLASSES_ROOT* key. Here's the RGS script to register the extension:

Hide Copy Code

```
HKCR
{
  NoRemove *
  {
    NoRemove shellex
    {
      NoRemove PropertySheetHandlers
      {
        {3FCEF010-09A4-11D4-8D3B-D12F9D3D8B02}
      }
    }
  }
}
```

You might notice that the extension's GUID is the stored as the name of a registry key here, instead of a string value. The documentation and books I've looked at conflict on the correct naming convention, although during my brief testing, both ways worked. I have decided to go with the way Dino Esposito's book (*Visual C++ Windows Shell Programming*) does it, and put the GUID in the name of the registry key.

As always, on NT-based OSes, we need to add our extension to the list of "approved" extensions. The code to do this is in the DllRegisterServer() and DllUnregisterServer() functions in the sample project.