C, C++, VC6, Windows, Visual Studio, COM, Dev

Posted : **3 Aug 2006**
Updated : **8 Aug 2006**
Views : **18,652**

COM in plain C, Part 7

By **Jeff Glatt**

An ActiveX Script Host with custom COM objects. This allows a script to call C functions in your app.

12 votes for this Article.
Popularity: 4.84 Rating: **4.49** out of 5
1 2 3 4 5

Download source files - 380 Kb

Contents

Introduction

In the previous chapter, we learned how to run a script from our application. But in order for that script to be able to call C functions in our application, and ultimately, exchange data with our application, we need to add another, custom COM object to our application. Our C functions (that are callable by the script) will be wrapped in this COM object. As you'll recall from Chapter 2, in order for a script to call functions in our COM object, we will need to add the `IDispatch` standard functions to it (as well as the `IUnknown` standard functions).

In a nutshell, what we're going to do is create our own custom COM object, just like we did in Chapter 2. We're going to define it as containing the `IUnknown` functions, followed by the `IDispatch` functions, followed by whatever extra functions that we'd like the script to be able to call. And, we'll also write an *.IDL* file in order to create a type library (so that the script can discover what the names of our extra functions are, what arguments they are passed, and what they return). All we're doing is creating our own custom, script-accessible object, exactly like we did in Chapter 2. But instead of putting this object in its own DLL, it will be part of our executable.

Declaring our custom COM object

We can choose to call our custom COM object anything we'd like. Let's arbitrarily call it an `IApp`.

Let's provide three functions that a script may call, named `Output`, `SetTitle`, and `GetTitle`. The `Output` function will display a line of text in our main window's EDIT control. We'll use another custom window message (`WM_APP` + 1) with `PostMessage` to pass off this line to our main window procedure to display -- just like we did in our `IActiveScriptSite`'s `OnHandleError` function. The `SetTitle` function will change the text in our main window's title bar. And, the `GetTitle` function will retrieve the title bar text. We'll put these three functions in our `IApp` object, after the `IUnknown` and `IDispatch` functions.

In the directory *ScriptHost4* is a version of our Script Host that adds this `IApp` object. The source file named *AppObject.c* contains most of the new code.

We'll need a unique GUID for our `IApp` object, so I've run *GUIDGEN.EXE* to create one, and given it the name `CLSID_IApp`. We'll also need a GUID for `IApp`'s VTable, so I created another GUID and gave it the name `IDD_IApp`. I've put these two GUIDs in the source files named *Guids.c* and *Guids.h*.

As you'll recall from Chapter 2, we use a special macro to declare our IApp object's VTable (and the object itself). Here's the declaration:

```
#undef  INTERFACE
#define INTERFACE IApp
DECLARE_INTERFACE_ (INTERFACE, IDispatch)
{
   // IUnknown functions

   STDMETHOD  (QueryInterface)(THIS_ REFIID, void **) PURE;
   STDMETHOD_ (ULONG, AddRef)(THIS) PURE;
   STDMETHOD_ (ULONG, Release)(THIS) PURE;
   // IDispatch functions

   STDMETHOD_ (ULONG, GetTypeInfoCount)(THIS_ UINT *) PURE;
   STDMETHOD_ (ULONG, GetTypeInfo)(THIS_ UINT, LCID, ITypeInfo **) PURE;
   STDMETHOD_ (ULONG, GetIDsOfNames)(THIS_ REFIID, LPOLESTR *, UINT,
               LCID, DISPID *) PURE;
   STDMETHOD_ (ULONG, Invoke)(THIS_ DISPID, REFIID, LCID, WORD,
               DISPPARAMS *, VARIANT *, EXCEPINFO *, UINT *) PURE;
   // Extra functions

   STDMETHOD  (Output)(THIS_ BSTR) PURE;
   STDMETHOD  (SetTitle)(THIS_ BSTR) PURE;
   STDMETHOD  (GetTitle)(THIS_ BSTR *) PURE;
};
```

Remember that the above macro also automatically defines our IApp as containing one member, lpVtbl, which is a pointer to the above VTable. But as is so often the case, we need to add extra, private members to our object, so we'll also define a MyRealIApp struct, as so:

```
typedef struct {
  IApp                       iApp;
  IProvideMultipleClassInfo classInfo;
} MyRealIApp;
```

Our MyRealIApp wraps our IApp, as well as a standard COM sub-object known as an IProvideMultipleClassInfo, which we'll examine later.

We need to declare the global VTable for our IApp, and also our IProvideMultipleClassInfo sub-object's VTable:

```
// Our IApp VTable.

IAppVtbl IAppTable = {
  QueryInterface,
  AddRef,
  Release,
  GetTypeInfoCount,
  GetTypeInfo,
  GetIDsOfNames,
  Invoke,
  Output,
  SetTitle,
  GetTitle};

// Our IProvideMultipleClassInfo VTable.

IProvideMultipleClassInfoVtbl IProvideMultipleClassInfoTable = {
  QueryInterface_CInfo,
  AddRef_CInfo,
  Release_CInfo,
  GetClassInfo_CInfo,
  GetGUID_CInfo,
  GetMultiTypeInfoCount_CInfo,
  GetInfoOfIndex_CInfo};
```

For our purposes, we'll need only one IApp object. All running scripts will share this one object (so it's important to employ some synchronization in our IApp functions if they access any global data and we have multiple script threads running). The easiest thing to do then is to declare our IApp object as a global, and call a function at the start of our program to initialize it.

We'll also be loading two ITypeInfos -- one for IApp's VTable, and one for our IApp object itself. And, we'll need to store those ITypeInfos somewhere. Because we need only one of each, I'll declare two global variables for those.

```
// For our purposes, we need only one IApp object, so

// we'll declare it global

MyRealIApp   MyIApp;

// The ITypeInfo for our IApp object. We need only 1 so we make it global

ITypeInfo   *IAppObjectTypeInfo;

// The ITypeInfo for our IApp VTable. We need only one of these

ITypeInfo   *IAppVTableTypeInfo;

void initMyRealIAppObject(void)
{
   // Initialize the sub-object VTable pointers

   MyIApp.iApp.lpVtbl = &IAppTable;
   MyIApp.classInfo.lpVtbl = &IProvideMultipleClassInfoTable;

   // Haven't yet loaded the ITypeInfos for IApp's VTable, nor IApp itself

   IAppObjectTypeInfo = IAppVTableTypeInfo = 0;
}
```

Our IDL file and type library

We need to create an *.IDL* file so *MIDL.EXE* can compile a type library for us. Our IDL file defines our IApp VTable, and the IApp object itself. This is essentially the same thing that we did in Chapter 2 when we defined our IExample2 custom COM object. You'll find a source file named *ScriptHost.idl* to peruse.

Note that IApp's VTable is declared as a dual interface. This will allow our IDispatch functions to use some standard COM calls to do most of the work of those functions. Again, this is the same thing as what we did with IExample2's IDispatch functions.

Also notice that our SetTitle and GetTitle functions are declared as propput and propget, and reference the same DISPID number. So, these two functions work to set and fetch the value of a variable named Title as far as the script is concerned. Again, this should be familiar to you from Chapter 2.

One thing we're going to do differently than IExample2 concerns the actual type library that *MIDL.EXE* will create for us (i.e., *ScriptHost.tlb*). With IExample2, we simply left the type library as a separate file. With our application, we are instead going to embed that type library file in our EXE's resources and use the COM function LoadTypeLib to extract/load it from our resources.

I've created an *.RC* file for our application's resources. In *ScriptHost.rc*, you'll see the following statement:

```
1 TYPELIB MOVEABLE PURE "debug/ScriptHost.tlb"
```

This takes the type library file (*ScriptHost.tlb*, compiled by *MIDL.EXE* and placed in the *Debug* directory), and embeds it into our EXE's resources, giving it a resource ID number of 1. The type of the resource is TYPELIB, which indicates that this resource is a type library file. We do not need to ship our type library file (*ScriptHost.tlb*) as a separate file because now it is embedded right inside of our EXE.

Here's a function that takes a GUID and creates an ITypeInfo for whatever that GUID refers to. The function extracts the info from our embedded type library resource. For example, to get an ITypeInfo for our IApp VTable, we simply pass the GUID for IApp's VTable. We also pass a handle where the ITypeInfo is returned.

```
HRESULT getITypeInfoFromExe(const GUID *guid, ITypeInfo **iTypeInfo)
{
   wchar_t    fileName[MAX_PATH];
   ITypeLib   *typeLib;
   HRESULT    hr;

   // Assume an error

   *iTypeInfo = 0;

   // Load the type library from our EXE's resources

   GetModuleFileNameW(0, &fileName[0], MAX_PATH);
   if (!(hr = LoadTypeLib(&fileName[0], &typeLib)))
   {
      // Let Microsoft's GetTypeInfoOfGuid() create a generic ITypeInfo

      // for the requested item (whose GUID is passed)

      hr = typeLib->lpVtbl->GetTypeInfoOfGuid(typeLib, guid, iTypeInfo);

      // We no longer need the type library

      typeLib->lpVtbl->Release(typeLib);
   }

   return(hr);
}
```

Register our COM object with the engine

To register our COM object with the engine, we must call the engine `IActiveScript`'s `AddNamedItem` function once. We must also decide upon a string name for this COM object, which the script will use to call our COM functions. This name should be a legal variable name in any of the language engines we use. A good generic approach is to use all alphabetic characters for the string name (without spaces), which is supported by almost every language as a variable name.

Let's arbitrarily decide upon a string name of "application". We need to register our COM object before we actually run the script, but after we initialize the engine. So we'll add the call to `AddNamedItem` in `runScript` after we call `SetScriptSite`, but before we load the script and pass it off to `ParseScriptText`. Here is the line we add:

```
args->EngineActiveScript->lpVtbl->AddNamedItem(args->EngineActiveScript,
    "application", SCRIPTITEM_ISVISIBLE|SCRIPTITEM_NOCODE)
```

The second argument is our string name, which is "application" here. The third argument is some flags. The `SCRIPTITEM_ISVISIBLE` flag means that the script can call our object's functions. (Without this flag, the script will not be able to call any of our functions. There may be a reason why you'd want this, but not here.) `SCRIPTITEM_NOCODE` means that the object's functions are C code inside our executable. It is possible to actually add an object to a script where the object's functions are contained in another script you add to the engine. In that case, you would not use `SCRIPTITEM_NOCODE`, but here we want it because our object's functions are indeed C functions in our executable.

As soon as we call `AddNamedItem`, the script engine will call our `IActiveScriptSite`'s `GetItemInfo` function to retrieve a pointer to our custom COM object and/or its `ITypeInfo`. So now, we need to replace the former stub code with useful instructions. Here it is:

```
STDMETHODIMP GetItemInfo(MyRealIActiveScriptSite *this, LPCOLESTR
   objectName, DWORD dwReturnMask, IUnknown **objPtr, ITypeInfo **typeInfo)
{
   HRESULT   hr;

   // Assume failure

   hr = E_FAIL;
   if (dwReturnMask & SCRIPTINFO_IUNKNOWN) *objPtr = 0;
   if (dwReturnMask & SCRIPTINFO_ITYPEINFO) *typeInfo = 0;

   // Does the engine want our IApp object (has the string name "application")?

   if (!lstrcmpiW(objectName, "application"))
   {
      // Does the engine want a pointer to our IApp object returned?

      if (dwReturnMask & SCRIPTINFO_IUNKNOWN)
      {
         // Give the engine a pointer to our IApp object. Engine will call

         // its AddRef() function, and later Release() it when done

         *objPtr = getAppObject();
      }

      // Does the engine want the ITypeInfo for our IApp object returned?

      if (dwReturnMask & SCRIPTINFO_ITYPEINFO)
      {
         // Make sure we have an ITypeInfo for our IApp object. (The script

         // engine needs to figure out what args are passed to our IApp's

         // extra functions, and what those functions return. And for that,

         // it needs IApp's ITypeInfo). The engine will call its AddRef()

         // function, and later Release() it when done

         if ((hr = getAppObjectITypeInfo(typeInfo))) goto bad;
      }

      hr = S_OK;
   }
bad:
   return(hr);
}
```

Notice that the engine will pass the string name we gave to the object (i.e., "application"), so we do a string compare to ensure that the engine is indeed asking for our `IApp` object. The engine will also pass some handles where it wants us to return a pointer to the object and/or its `ITypeInfo`. If the engine wants a pointer to our object, it passes the `SCRIPTINFO_IUNKNOWN` flag. If it wants our object's `ITypeInfo`, then it passes the `SCRIPTINFO_ITYPEINFO` flag. Note that both flags can be passed simultaneously. We simply call our `getAppObject` and/or `getAppObjectITypeInfo` functions (in *AppObject.c*) to fill in the engine's handles.

How a script calls our COM object functions

When we call `AddNamedItem`, our COM object is added to the engine, and exposed to the script as if it were an object that the script itself created. For example, if a VBScript wants to call our COM object's `Output` function, it will do so as follows:

```
application.Output("Some text")
```

In VBScript, a dot follows an object name. Notice that the script references our `IApp` object using our string name of "application". The VBScript does not need to call `CreateObject`. Because our application has registered our object via `AddNamedItem`, our object is automatically available to the script just by using our chosen string name.

If the script wants to set our Title property (i.e., call our `SetTitle` function), it can do so as follows:

```
application.Title = "Some text"
```

If the script wants to retrieve the value of our `Title` property (i.e., call our `GetTitle` function), it can do so as follows:

```
title = application.Title
```

There is an example VBScript named *script.vbs* (in the *ScriptHost4* directory) which calls our `IApp` functions.

Our IProvideMultipleClassInfo object

Our `IProvideMultipleClassInfo` object functions (in *AppObject.c*) are called by the engine when it needs to get GUIDs and/or `ITypeInfo` objects related to our `IApp` object. The engine may ask for the GUID of our `IApp` [default] VTable, or it may ask for the GUID of any [default, source] VTable our `IApp` object has (if we had one such VTable), or may ask for our `IApp`'s `ITypeInfo` object (i.e., not one of its VTables' `ITypeInfo`s).

Application and document objects

Suppose we have a text editor application. The app is a "multiple document" app, meaning that the user can edit several different text files, each open in its own MDI window.

In this case, our `IApp` object would be more or less an "application object"; that is, it would contain functions to control the overall operation of our app. For example, it may have a function to show or hide the toolbar or status bar. We'd need only one `IApp` object.

We'd define a second custom object, which we'll arbitrarily call an `IDocument` object. This object would have functions that control one text editor window and its contents. For example, perhaps, we'd have a function to insert some text at the current cursor position, and another function to move the cursor to a certain position in the document. Every time we create/load another text file, we'd create another `IDocument` object just for this new document. So, if the user has several editor windows open, we'd have several `IDocument` objects (one per window).

Because we're going to have several instances of the same object, we would not call the engine's `AddNamedItem` to register our `IDocument` object. (Indeed, if the user has not yet opened any documents, we may not have any instance of an `IDocument` object.) So, how may the script gain access to a particular `IDocument` object (assuming the script wishes to alter the contents of that document)? Typically, our `IApp` object will have some function the script may call, which will return an `IDocument` object. Maybe, the script will pass the "name" of the document it desires. Or maybe, our `IApp` will have a function that simply returns the currently active document's `IDocument`. (It's up to you how you want to do this).

Typically, you'd have a linked list of some "document" structures which the `IApp` could search through to find/retrieve the appropriate `IDocument` object. For example, maybe, we would define a `GetDocument` function for our `IApp`. `GetDocument` would be passed the `BSTR` name of the desired `IDocument` to fetch. Our IDL file definition may look like this:

```
[id(3)] HRESULT GetDocument([in] BSTR name, [out, retval] IDispatch **document);
```

Of course, in order for a script to call our `IDocument`'s functions, our `IDocument` must have the standard `IDispatch` functions in its VTable. In that case, it can (and should, as far as the script engine is concerned) masquerade as an `IDispatch`. So that's what we indicate that `GetDocument` returns.

So for example, a VBScript may fetch a document named *test.txt*, like so:

```
SET doc = application.GetDocument("test.txt")
```

In the directory *ScriptHost5* is a rudimentary text file viewer. It's a simple MDI app that can open numerous text files for viewing; each file in its own window. We have an IApp object with one extra function called CreateDocument. This is passed the BSTR filename of a text file to load. It creates a new MDI child window, and loads/displays that text file in the window. (Actually, we simply create a blank document.) CreateDocument also creates a new IDocument object for that window, and returns it to the script.

Our IDocument object contains a function called WriteText. This is passed a BSTR of text, and this text replaces any other text in the window.

There is a VBScript file named *test.vbs* in the *ScriptHost5* directory. It simply calls our IApp's CreateDocument twice, to create two IDocument objects, naming them "Document 1" and "Document 2". It calls the WriteText function of each IDocument. For "Document 1", it sets the text "This is document 1.". For "Document 2", it sets the text "This is document 2.".

Note that the ScriptHost5 is hardwired to load and run *text.vbs*, just for the sake of simplicity.

From this example, you should notice how our IApp object controls the overall operation of our viewer, whereas the IDocument controls each individual document. Note the additions we made to *ScriptHost.idl* to define our two custom objects (and note that both have dual VTables, so we use standard COM functions to do most of the IDispatch work). Also note that we need another GUID for IDocument's VTable in our IDL file. But we don't define the IDocument object itself in our IDL file. We don't need to, since the script doesn't ever create one of these via CoCreateInstance, but rather, indirectly fetches one that our app itself creates.

A C++ example host

In the directory *ScriptHost6* is a C++ version of ScriptHost5. You'll notice in *AppObject.h* that we declare our MyRealIApp and MyRealIDocument as classes based upon our IApp and IDocument. The COM functions are then made members of their respective classes. Notice that, whereas in the C example our MyRealApp's functions took a pointer to the MyRealApp as the first argument, this is omitted in the C++ version. That's because it becomes the hidden "this" pointer. And, we do not need to declare and initialize the pointers to any VTables. The C++ compiler does all that for us.

Furthermore, whenever we call a COM function, we omit the ->lpVtbl, and do not pass a pointer to the object as the first argument.

Also, in *IActiveScriptSite.h*, we declare our MyRealIActiveScriptSite as a class based upon both an IActiveScriptSite and an IActiveScriptSiteWindow. Notice that we no longer need a separate QueryInterface, AddRef, and Release for each sub-object. We declare only the base object's QueryInterface, AddRef, and Release, and then the C++ compiler automatically generates those functions, with proper delegation, for other sub-objects.

License

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

13 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/com_in_c7.aspx** to post and view comments on this article, or click **here** to get a print view with messages.