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

COM in plain C, Part 6

By **Jeff Glatt**

How to write an ActiveX Script Host in C.

Posted : **23 Jul 2006**
Updated : **23 Jul 2006**
Views : **20,281**

28 votes for this Article.
Popularity: 6.92 Rating: **4.78** out of 5
1 2 3 4 5

Download source files - 305 Kb

Contents

Introduction

When creating an application, it's desirable to provide the end user with a "macro language" he can use to write scripts (i.e., text files containing "instructions" or "commands") which control the operation of your application. For example, if you've created an email program, perhaps you'd like to let the end user write some script which can send an email to a particular address. To do that, perhaps your macro language will provide a `SendMail` function that the script calls, passing an email address, and text body, (each double-quoted), and the syntax would look like so:

```
SendMail("somebody@somewhere.com", "hello")
```

With a macro language, the end user can write a script to automatically perform repetitive operations (hence the term "automation" is used to describe a script controlling an application), or perhaps even add new functionality to your application (if your macro language is powerful/flexible enough).

Microsoft decided to add a macro language to many of its products such as Word, Excel, etc. In fact, MS decided to use a simplified variation of Visual Basic as the basis for the macro language. So, MS put that simplified version of the Visual Basic interpreter (without features such as being able to compile a script into an executable, and other advanced features) into a DLL. Then, MS put particular COM objects inside that DLL which Word or Excel could obtain/use to tell the interpreter to run a VB script, and interact with the application's own functions. For example, one of the special COM objects is an `IActiveScript` object. MS called their new, simplified VB-interpreter-in-a-DLL (with a COM interface) **VBScript**. And the DLL, with that specific set of COM objects, is referred to as an **ActiveX Script Engine**.

Then, MS got to thinking that it would be nice to give the end user his choice of macro language. For example, some end users may want to write their scripts using a Java-like language instead of VBScript. So, MS also created another DLL containing an interpreter that implemented a simplified variation of Java. This interpreter was called **JavaScript**. The JavaScript DLL contained the same set of COM objects as the VBScript DLL. MS devised these COM objects so that they were mostly "language neutral". In other words, Excel could give the JavaScript DLL a JavaScript file to run in the exact same way that Excel could give the VBScript DLL a VBScript file to run. So now, the end user had his choice of two ActiveX Script Engines to use. Later, other third parties packaged their interpreters into a DLL, with these same set of COM objects, and now you can find ActiveX Script Engines in a variety of other languages such as Python, Perl, etc. And any of them can be used with any application that supports any ActiveX Script Engine.

An application that utilizes an ActiveX Script Engine is referred to as an **ActiveX Script Host**.

In order for an application to interact with the engine, the application (EXE) has to have its own, special COM object inside of it known as an `IActiveScriptSite` object. The application calls one of the engine's COM object functions to give it a pointer to the application's `IActiveScriptSite` COM object. Then, the engine and application can communicate, and coordinate the running of a script via their COM objects' functions.

This article will detail how to write an ActiveX Script Host -- that is, how to write an application (EXE) that can load one of these ActiveX Script Engines (DLL), and call the engine's COM objects to run some script (text file) containing instructions in that engine's language. In our particular example, we'll use the VBScript engine, and so our example script file will contain VBScript instructions. But we could easily use any other engine, and write our script using that engine's language instead.

> **In conclusion**, an ActiveX Script Engine is an interpreter that contains some standard COM objects defined by Microsoft. It can be used by any application (i.e., executable) that knows how to utilize those COM objects. Such an application is called an ActiveX Script Host. A properly written host should be able to use any engine interchangeably.

Choosing/opening an engine

Every ActiveX script engine must have its own unique GUID. So, if you know what particular engine you wish to use, you can pass that engine's GUID to the function `CoCreateInstance` to open that script engine and get its `IActiveScript` object (just like in the very first chapter, we wrote an application that passed our `IExample` object's GUID to `CoCreateInstance` and got an `IExample` object). You should be able to find the engine's GUID in some include file shipped with the engine's "development kit".

An ActiveX engine can also associate itself with files whose names end in a particular extension, just like an application can set a file association. The engine's installer will have setup a registry key with the associated file extension. For example, the VBScript engine associates itself with files whose names end in *.vbs*. Your application can look up the file association in the registry, and get the engine's GUID that way. (Then, once you have the GUID, you can call `CoCreateInstance`).

Here is a function that is passed a file extension, for which you wish to get the associated engine's GUID, and a buffer big enough to retrieve that GUID. The function looks up the appropriate registry keys to find that engine's GUID, and copies it to the buffer:

```c
HRESULT getEngineGuid(LPCTSTR extension, GUID *guidBuffer)
{
   wchar_t   buffer[100];
   HKEY      hk;
   DWORD     size;
   HKEY      subKey;
   DWORD     type;

   // See if this file extension is associated

   // with an ActiveX script engine

   if (!RegOpenKeyEx(HKEY_CLASSES_ROOT, extension, 0,
       KEY_QUERY_VALUE|KEY_READ, &hk))
   {
      type = REG_SZ;
      size = sizeof(buffer);
      size = RegQueryValueEx(hk, 0, 0, &type,
             (LPBYTE)&buffer[0], &size);
      RegCloseKey(hk);
      if (!size)
      {
         // The engine set an association.

         // We got the Language string in buffer[]. Now

         // we can use it to look up the engine's GUID


         // Open HKEY_CLASSES_ROOT\{LanguageName}

again:   size = sizeof(buffer);
         if (!RegOpenKeyEx(HKEY_CLASSES_ROOT, (LPCTSTR)&buffer[0], 0,
                           KEY_QUERY_VALUE|KEY_READ, &hk))
         {
            // Read the GUID (in string format)

            // into buffer[] by querying the value of CLSID

            if (!RegOpenKeyEx(hk, "CLSID", 0,
                KEY_QUERY_VALUE|KEY_READ, &subKey))
            {
               size = RegQueryValueExW(subKey, 0, 0, &type,
                      (LPBYTE)&buffer[0], &size);
               RegCloseKey(subKey);
            }
            else if (extension)
            {
               // If an error, see if we have a "ScriptEngine"

               // key under here that contains

               // the real language name

               if (!RegOpenKeyEx(hk, "ScriptEngine", 0,
                   KEY_QUERY_VALUE|KEY_READ, &subKey))
               {
                  size = RegQueryValueEx(subKey, 0, 0, &type,
                         (LPBYTE)&buffer[0], &size);
                  RegCloseKey(subKey);
                  if (!size)
                  {
                     RegCloseKey(hk);
                     extension = 0;
                     goto again;
                  }
               }
            }
         }

         RegCloseKey(hk);

         if (!size)
         {
            // Convert the GUID string to a GUID

            // and put it in caller's guidBuffer

            if ((size = CLSIDFromString(&buffer[0], guidBuffer)))
```

So, to look up VBScript's GUID, we can call getEngineGuid, passing the associated file extension of ".vbs" as so:

```
GUID  guidBuffer;

// Find the script engine to use for files that end with a .VBS extension.

// NOTE: Microsoft's VBscript engine sets up an association in the

// registry for this extension.

getEngineGuid(".vbs", &guidBuffer);
```

Now, to load/open the VBScript engine, and get its IActiveScript object (into our variable we'll name activeScript), we can call CoCreateInstance. Note that the IActiveScript object's GUID is defined by Microsoft, using the name IID_IActiveScript, in an include file called *activscp.h* which can be found in the Platform SDK.

```
#include <window.h>

#include <objbase.h>

#include <activscp.h>


IActiveScript  *activeScript;

CoCreateInstance(&guidBuffer, 0, CLSCTX_ALL,
                 &IID_IActiveScript,
                 (void **)&activeScript);
```

We will also need to get another of the engine's COM objects, called an IActiveScriptParse object. This is a sub-object of the IActiveScript object, so we can pass the IActiveScriptParse's GUID to the IActiveScript's QueryInterface function. Microsoft has defined IActiveScriptParse's GUID with the name IID_IActiveScriptParse. Here, we get this object into our variable named activeScriptParse:

```
IActiveScriptParse  *activeScriptParse;

activeScript->lpVtbl->QueryInterface(activeScript,
             &IID_IActiveScriptParse,
             (void **)&activeScriptParse);
```

> **In conclusion**, each ActiveX Script Engine has its own unique GUID. A host can open an engine (and get the engine's IActiveScript and IActiveScriptParse objects) in the same way that any other COM component is accessed -- by passing that unique GUID to CoCreateInstance. Furthermore, an engine may be associated with a particular file extension, so an engine's GUID can be "looked up" by querying the file extension's registry key.

Our IActiveScriptSite object

We need to supply our own COM object called an IActiveScriptSite object. Microsoft has already defined its GUID and VTable (i.e., an IActiveScriptSiteVtbl struct) for us. All we need to do is write the functions for it. Of course, an IActiveScriptSite VTable starts with the QueryInterface, AddRef, and Release functions. It contains 8 more functions called GetLCID, GetItemInfo, GetDocVersionString, OnScriptTerminate, OnStateChange, OnScriptError, OnEnterScript, and OnLeaveScript. Most of these functions are called by the engine when it wants to notify us of something. For example, our OnEnterScript function is called whenever some function in the script is called. Our OnScriptError is called when/if there's an error in the script itself. Other functions are meant for us to provide information to the engine. For example, the engine calls our GetLCID to ask us what language LCID to use for any dialogs the engine may display.

For now, most of our IActiveScriptSite functions can be stub routines that do nothing except return S_OK.

We'll also provide another sub-object of our IActiveScriptSite. This sub-object is referred to as

IActiveScriptSiteWindow. This sub-object is used by the engine to interact with any application windows we may have open. This is an optional object. We don't need to provide it, but if our application opens its own windows, then this is a useful object to provide.

Because we'll need an IActiveScriptSiteWindow sub-object, we'll define a MyRealIActiveScriptSite struct to wrap both our IActiveScriptSite and IActiveScriptSiteWindow, as so:

```
typedef struct {
   // The IActiveScriptSite must be the base object.

   IActiveScriptSite        site;
   IActiveScriptSiteWindow  siteWnd;
   // Our IActiveScriptSiteWindow sub-object

   // for this IActiveScriptSite.

} MyRealIActiveScriptSite;
```

For our purposes, we're going to need only one IActiveScriptSite (and its IActiveScriptSiteWindow), so the easiest thing is to just declare it globally, and also declare the VTables globally:

```
// Our IActiveScriptSite VTable.

IActiveScriptSiteVtbl SiteTable = {
 QueryInterface,
 AddRef,
 Release,
 GetLCID,
 GetItemInfo,
 GetDocVersionString,
 OnScriptTerminate,
 OnStateChange,
 OnScriptError,
 OnEnterScript,
 OnLeaveScript};

// IActiveScriptSiteWindow VTable.

IActiveScriptSiteWindowVtbl SiteWindowTable = {
 siteWnd_QueryInterface,
 siteWnd_AddRef,
 siteWnd_Release,
 GetSiteWindow,
 EnableModeless};

// Here's our IActiveScript and its IActiveScriptSite sub-object, wrapped

// in our MyRealIActiveScriptSite struct.

MyRealIActiveScriptSite  MyActiveScriptSite;
```

And of course, we need to initialize its VTable pointers at the start of our program:

```
// Initialize the lpVtbl members of our IActiveScriptSite and

// IActiveScriptSiteWindow sub-objects

MyActiveScriptSite.site.lpVtbl = &SiteTable;
MyActiveScriptSite.siteWnd.lpVtbl = &SiteWindowTable;
```

In the directory *ScriptHost*, is an example of a simple ActiveX Script Host. The source file, *IActiveScriptSite.c*, contains the VTables and functions for our IActiveScriptSite and IActiveScriptSiteWindow objects (which are wrapped up in our own MyRealIActiveScriptSite struct). As mentioned in this example, most of the functions are simply stub routines that do nothing. The only non-trivial function is OnScriptError. The engine calls our OnScriptError function if there is a syntax error in the script (i.e., the script itself is written/formatted incorrectly), or there's a runtime error in the script (for example, the engine runs out of memory while executing the script).

The engine passes one of its own COM objects called an IActiveScriptError. This object has functions we can call to get information about the error, such as the line number in the script where the error occurred, and a text message that describes the error. (Note: The line number is referenced from 0, so the first line in the script is line number 0.)

All we do is call some IActiveScriptError functions to get some information, reformat it, and display it to the user in a message box.

```
STDMETHODIMP OnScriptError(MyRealIActiveScriptSite *this,
             IActiveScriptError *scriptError)
{
   ULONG        lineNumber;
   BSTR         desc;
   EXCEPINFO    ei;
   OLECHAR      wszOutput[1024];

   // Call GetSourcePosition() to retrieve the line # where

   // the error occurred in the script

   scriptError->lpVtbl->GetSourcePosition(scriptError, 0, &lineNumber, 0);

   // Call GetSourceLineText() to retrieve the line in the script that

   // has an error.

   desc = 0;
   scriptError->lpVtbl->GetSourceLineText(scriptError, &desc);

   // Call GetExceptionInfo() to fill in our EXCEPINFO struct with more

   // information.

   ZeroMemory(&ei, sizeof(EXCEPINFO));
   scriptError->lpVtbl->GetExceptionInfo(scriptError, &ei);

   // Format the message we'll display to the user

   wsprintfW(&wszOutput[0], L"%s\nLine %u: %s\n%s", ei.bstrSource,
      lineNumber + 1, ei.bstrDescription, desc ? desc : "");

   // Free what we got from the IActiveScriptError functions

   SysFreeString(desc);
   SysFreeString(ei.bstrSource);
   SysFreeString(ei.bstrDescription);
   SysFreeString(ei.bstrHelpFile);

   // Display the message

   MessageBoxW(0, &wszOutput[0], "Error",
            MB_SETFOREGROUND|MB_OK|MB_ICONEXCLAMATION);

   return(S_OK);
}
```

Note that the IActiveScriptError object is good only for the lifetime of our OnScriptError function. In other words, when our OnScriptError returns, that particular IActiveScriptError object disappears (unless we explicitly AddRef it). Also note that the IActiveScriptError's functions return copies of any information we request, so we have to eventually free any BSTRs it returns to us.

> **In conclusion**, a Script Host must provide a standard COM object called an IActiveScriptSite. It may also optionally provide an IActiveScriptSiteWindow, which is a sub-object of the IActiveScriptSite. In a minimal implementation, the functions can simply be stub functions that do nothing. But, the OnScriptError function is typically used to inform the user of any error that occurs in the script.

An example VBScript

Let's run the following VBScript, which simply displays a message box with the text "Hello world":

```
MsgBox "Hello world"
```

To make it easy, we'll simply embed this script as a string right inside of our executable, as global data declared like so:

```
wchar_t VBscript[] = L"MsgBox \"Hello world\"";
```

There is one important thing to note. I've declared this string a wide (UNICODE) data type, and initialized it as so (i.e., the datatype `wchar_t` indicates wide characters, and the **L** qualifier on the string also indicates as much). All script engine functions expect wide character strings. So, when we give the VBScript engine our script to run, it must be in UNICODE format, even if our executable itself isn't internally using UNICODE.

Initializing the engine

Before we can run our script, we first have to open the engine and get its `IActiveScript` object (via `CoCreateInstance`) and its `IActiveScriptParse` sub-object, as shown earlier.

When an engine is first open, it is in the *unitialized state*. Before we can give the engine any scripts to run, we must initialize the engine (once only). This merely involves a call to the engine `IActiveScriptParse`'s `Init` function.

Furthermore, we need to give the engine a pointer to our `IActiveScriptSite` object. Again, we need to do this once only. This merely involves a call to the engine `IActiveScript`'s `SetScriptSite` function, passing a pointer to our `IActiveScriptSite` (which is embedded at the start of our `MyRealIActiveScriptSite`, so a simple cast does the trick).

Here then, are the two calls we need to do once only after the engine is opened:

```
// Let the engine do any initialization it needs to do.

activeScriptParse->lpVtbl->InitNew(activeScriptParse);

// Give the engine our IActiveScriptSite object.

activeScript->lpVtbl->SetScriptSite(activeScript,
     (IActiveScriptSite *)&MyActiveScriptSite);
```

After the above two calls, the engine will automatically switch to the *initialized state*. It is now ready for us to add scripts to the engine.

Note: The engine's `SetScriptSite` function may call our `IActiveScriptSite`'s `QueryInterface` to ask us to return several sub-objects. For example, perhaps, we'll be asked to return a pointer to our `IActiveScriptSiteWindow` sub-object. When we call `SetScriptSite`, we should be prepared to provide any requested sub-objects. If we need to do any pre-initialization of our own COM objects, we should do that **before** calling `SetScriptSite`.

> **In conclusion**, before running any scripts, the host must call an engine's `Init` and `SetScriptSite` functions, to initialize the engine, and to give a pointer to the host's `IActiveScriptSite` object, respectively. This should be done once only, after the engine is opened.

Adding a script to the engine

In order to run a script, we first need to give that script to the engine. We do this by passing a memory buffer, containing the script, to the engine `IActiveScriptParse`'s `ParseScriptText` function. Remember that the script must be in wide-character format. It must also be null-terminated. Since our VBScript is already in a memory buffer (i.e., it's a global variable inside of our executable, declared as `wchar_t` and null-terminated), all we need to do is pass the address of that global, as so:

```
activeScriptParse->lpVtbl->ParseScriptText(activeScriptParse,
                    &VBscript[0], 0, 0, 0, 0, 0, 0, 0, 0);
```

ParseScriptText takes a lot of other arguments, but for our purposes here, we can leave them all set to 0.

So, what happens when we call ParseScriptText? First of all, the engine checks the syntax of the script to ensure that it is a correctly written script. Here, the VB engine makes sure that our script contains legal VB instructions. If there's a syntax error, the engine's ParseScriptText will call our IActiveScriptSite's OnHandleError. The engine will not internally add the script, and (after our OnHandleError function returns) ParseScriptText will return an error (non-zero) value.

If the script is syntactically correct, then the engine makes its own copy of our script, perhaps reformatting it into some internal structures of its own choosing, in preparation of running the script. But the engine does not run the script at this point, because the engine is still in its initialized state. An engine will not run any script we add to the engine until we put the engine into either its *start* or *connected* states.

If all goes well, ParseScriptText returns 0 for success. The engine now has its own internally formatted version of our script, prepared for running. (At this point, if we had allocated the buffer containing the script, we could free it now if desired.)

> **In conclusion**, in order for a script to be run, the host must first pass a memory buffer containing the script (formatted in wide characters, and null-terminated) to the engine's ParseScriptText function. This causes the engine to make its own copy of the script in preparation of running the script. But the script does not run while the engine is still in its initialized state.

Running scripts

To run our VBScript, we simply need to switch the engine to either its *start* or *connected* state. We'll later discuss the differences between the two states, but for now, we'll simply switch to the connected state. We change the engine's state by calling its IActiveScript's SetScriptState, passing the desired state, which here is the constant SCRIPTSTATE_CONNECTED (defined in MS' *activscp.h* include file).

```
activeScript->lpVtbl->SetScriptState(activeScript, SCRIPTSTATE_CONNECTED);
```

As soon as we make this call, the engine starts executing any *immediate instructions* in the script. What is an immediate instruction? That depends upon the language. In VBScript, immediate instructions are any instructions at the start of the script which are not enclosed in some subroutine/function. Since our example script contains one instruction which happens to fit that description, that instruction is immediately executed. We should see a message box pop up with the string "Hello World".

SetScriptState will not return until all of those immediate instructions are done. In this case, it won't return until we dismiss that message box. And since that's the last immediate instruction in our VBScript, SetScriptState returns. At this point, we don't have any further use for the script nor the engine, so we can close down the engine.

Closing an engine

To close an engine, we simply call its IActiveScript's Close function, as so:

```
activeScript->lpVtbl->Close(activeScript);
```

This should cause the engine to stop any running scripts, free any internal resources that are no longer needed, and switch to the *closed* state. It should call our IActiveScriptSite's Release function, and free anything that it obtained from us (such as freeing the copy it made of our script).

After Close returns, we can then call the engine IActiveScriptParse's and IActiveScript's Release functions, like so:

```
activeScript->lpVtbl->Release(activeScript);
activeScript->lpVtbl->Release(activeScript);
```

We're now done with the engine.

Loading a script

Of course, our script language wouldn't be much use to an end user if he couldn't write his own scripts to run. So, instead of hard-coding a VBScript inside of our executable, let's present a file dialog to the end user so he can pick out a VBScript on disk. Then, we'll load the script into a memory buffer, making sure that the script is in wide character format and null-terminated, and pass that memory buffer to `ParseScriptText`.

I won't bother discussing how to present a file dialog to the end user to get his choice of filename.

After the user chooses the filename, we'll pass it to a function called `loadUnicodeScript`, which returns a memory buffer containing the script, formatted in wide characters and null-terminated.

```
OLECHAR * loadUnicodeScript(LPCTSTR fn)
{
   OLECHAR  *script;
   HANDLE   hfile;

   // Assume an error

   script = 0;

   // Open the file

   if ((hfile = CreateFile(fn, GENERIC_READ, FILE_SHARE_READ, 0,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0)) != INVALID_HANDLE_VALUE)
   {
      DWORD  filesize;
      char   *psz;

      // Get a buffer to read in the file, with room to nul-terminate

      filesize = GetFileSize(hfile, 0);
      if ((psz = (char *)GlobalAlloc(GMEM_FIXED, filesize + 1)))
      {
         DWORD  read;

         // Read in the file

         ReadFile(hfile, psz, filesize, &read, 0);

         // Get a buffer to convert to UNICODE, plus an extra wchar_t

         // to nul-terminate it

         if ((script = (OLECHAR *)GlobalAlloc(GMEM_FIXED, (filesize + 1)
             * sizeof(OLECHAR))))
         {
            // Convert to UNICODE and nul-terminate

            MultiByteToWideChar(CP_ACP, 0, psz, filesize, script, filesize + 1);
            script[filesize] = 0;
         }
         else
            display_sys_error(0);

         GlobalFree(psz);
      }
      else
          display_sys_error(0);

      CloseHandle(hfile);
   }
   else
      display_sys_error(0);

   return(script);
}

void display_sys_error(DWORD err)
{
   TCHAR   buffer[160];

   if (!err) err = GetLastError();
   buffer[0] = 0;
   FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0, err,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), &buffer[0], 160, 0);
   MessageBox(0, &buffer[0], "Error", MB_OK);
}
```

Note: `loadUnicodeScript` assumes that the file on disk is not Unicode. If there's a chance that you may be loading a disk file that is already in Unicode format, then you shouldn't convert it again. In this case, `loadUnicodeScript` will need to be modified to check any "signature" in the file. Consult other documentation for more information about different text file encoding.

We can make minor changes to our code to run the script. We simply call `loadUnicodeScript` to load the disk script into a memory buffer, and pass this buffer to `ParseScriptText`. Afterwards, we can free the buffer, and change the engine's state to *connected*, to run the script:

```
LPOLESTR   str;

// Load the script from disk.

str = loadUnicodeScript(fn);

// Have the script engine parse it and internally prepare it to be run.

hr = activeScriptParse->lpVtbl->ParseScriptText(activeScriptParse, str,
    0, 0, 0, 0, 0, 0, 0, 0);

// We no longer need the loaded script.

GlobalFree(str);

// Execute the script's immediate instructions.

activeScript->lpVtbl->SetScriptState(activeScript, SCRIPTSTATE_CONNECTED);
```

Enumerating installed engines

When the user picks out a script to run, we don't want to assume that it will be a VBScript. Perhaps it will be a Jscript, or a script associated with a Python engine, etc.

What we'll do is take the filename he picked, isolate the file extension on the name, and pass that extension to `getEngineGuid`. That will give us the GUID for the appropriate engine we need to open.

But what happens if the filename has no extension, or it has an extension that isn't associated with any installed script engine? In this case, we'll need to present the end user with a list of installed ActiveX Script Engines, and let him manually choose the engine he wants. Then, we'll get the GUID of that chosen engine, and open it.

Microsoft's OLE functions provide COM objects we can obtain and use to get a list of installed engines and their GUIDs. The COM object we need to obtain is known as an `ICatInformation`, and in particular, we want the `ICatInformation` object that lists script engines. We can obtain one of these objects by calling `CoCreateInstance`. We can then call its `EnumClassesOfCategories` to get a sub-object whose `Next` function enumerates the script engine GUIDs. We further can call `ProgIDFromCLSID` to get each engine's name (as registered by the engine's installer).

In the directory *ScriptHost2* is an example of a windowed (GUI) C application that presents a window with a "Run script" menu item. When the user selects this menu item, a file dialog is presented to get the name of the script to run. After picking the script name, the application isolates the extension, and tries to lookup the engine GUID associated with this extension. If no such engine is found, then the application presents a dialog box listing the installed engines so the user can pick the desired engine to use.

The source file *ChooseEngine.c* contains the code for presenting the list of installed engines and getting the GUID of the chosen engine.

Running a script in another thread

There is one problem with our GUI app. The script runs in the same thread as our user interface. The disadvantage of this is that, if the script were to do some never-ending loop, we'd be stuck in that call to `SetScriptState` forever, with no way for the user to abort the script. Indeed, the user interface is totally tied up while the script is running, so the user can't even move our application window.

For this reason, it is best to launch a separate thread to run a script. But there's one big caveat. Most of the engine's COM functions can be called only by the thread that calls `SetScriptSite`. So, we need to let our "script thread" do most of the setup/initialization and cleanup involved in running a script. Another caveat is that our `IActiveScriptSite` functions will be called within our script thread, so if we have any data that is accessed by both our `IActiveScriptSite` functions and UI thread functions, we'll need some sort of synchronization, such as a critical section around any access to that data.

In the directory *ScriptHost3* is a modified version of **ScriptHost2** which runs the script in a secondary thread.

Essentially, what we've done is turn our function runScript into the entry point of a second thread. Not too much alteration is necessary because runScript already does all the initialization and cleanup that would need to be done by the script thread. Most of the alteration concerns thread initialization and cleanup. First of all, the Windows operating system specifies that a thread can be passed only a single parameter (of our own choosing). But our runScript takes two arguments -- a filename, and a GUID. We need to define a new, single struct that wraps both of these. We'll call it a MYARGS struct, and define it as so:

```
typedef struct {
   IActiveScript  *EngineActiveScript;
   HANDLE         ThreadHandle;
   TCHAR          Filename[MAX_PATH];
   GUID           Guid;
} MYARGS;
```

Then, we'll pass runScript a pointer to our MYARGS.

MYARGS has two extra members. ThreadHandle stores a handle to the script thread. We'll also let the script thread store the engine's IActiveScript object pointer in our MYARGS. This is so that the main thread can also gain access to it later.

Since we're going to launch only one script at a time, we'll declare a global MYARGS:

```
MYARGS MyArgs;
```

Our main thread initializes its ThreadHandle member to 0 at the start of our app. We use this member to determine whether the script thread is running. When ThreadHandle is 0, the script thread isn't running. When not 0, it is the handle to the running script thread.

runScript needs to call CoInitialize once the thread starts. Each thread is responsible for initializing COM for itself. And of course, runScript must call CoUninitialize when done. Furthermore, we're going to change our main thread's call from CoInitialize to CoInitializeEx and pass the value COINIT_MULTITHREADED. This ensures that, if our main thread calls any IActiveScript function, then the engine won't block our main thread and force the function to be executed in our script thread. This is very important when we want our main thread to abort our script thread via InterruptScriptThread. We don't want to trust the script thread to abort itself, which it wouldn't be able to do if it was "hung".

Note: In order for the compiler to recognize CoInitializeEx, you must #define the symbol _WIN32_WINNT to be 0x0400 (or greater), and this must be done before you #include objbase.h.

When our main (UI) thread handles the IDM_FILE_RUNSCRIPT message, it fills in MYARG's filename and GUID fields with the name of the script to run and the GUID of the engine to use. Then, our main thread creates/launches the script thread with a call to CreateThread, passing our MYARGS, as so:

```
MyArgs.ThreadHandle = CreateThread(0, 0, runScript, &MyArgs, 0, &wParam);
```

Note: If your script thread, or your IActiveScriptSite functions, call any C language functions, then use beginthread instead. And check your C/C++ "Code Generation" settings to be sure you use a multi-threaded C library. In my example code, I do not call any C library functions that are sensitive to multi-threading, so I can use CreateThread.

Note that we save the handle of the thread in MYARGS's ThreadHandle. If the script thread launched OK, this is now non-zero. When our script thread terminates, it will reset ThreadHandle to 0.

There are two more matters to discuss related to what to do if the script thread has a problem running the script, and also what to do if our main thread needs to abort the script thread.

To make it easier for our main thread to cleanly abort any script, our script thread (and our IActiveScriptSite functions) should avoid doing anything that would cause the thread to "pause" or "wait for something". One example would be calling MessageBox. MessageBox causes a thread to wait until the user dismisses the message box. Another potential problem could be calling SendMessage. This waits for a window procedure to fully process the message and return. And if the window procedure thread does something that causes it to pause or wait, then the thread calling SendMessage also is doomed to pause and wait too.

In runScript, we called our function display_COM_error, which in turn calls MessageBox. This is not good. What we'll do is simply pass off any error message to our UI thread, and let our main thread display any error message box. To do this, we'll use PostMessage. For the message number, we'll use WM_APP (i.e., our own custom message number). For the WPARAM argument, we'll pass the address of the error string. If we pass a 0 for the WPARAM argument, then this means the LPARAM argument is an error number that we should pass to display_sys_error to get an error message to display. For the LPARAM argument, we'll pass an HRESULT error number. If we pass a 0 for the HRESULT, this means that the error string is a wide character string that has been GlobalAlloc()'ed. Our main thread will need to use MessageBoxW to display it, and then must subsequently GlobalFree it.

So for example, in runScript, we change the following error handling from...

```
if ((hr = activeScriptParse->lpVtbl->InitNew(activeScriptParse)))
   display_COM_error("Can't initialize engine : %08X", hr);
```

...to...

```
if ((hr = activeScriptParse->lpVtbl->InitNew(activeScriptParse)))
   PostMessage(MainWindow, WM_APP, (WPARAM)"Can't initialize engine : %08X", hr);
```

We need to modify loadUnicodeScript slightly so that it doesn't call display_sys_error, but instead calls PostMessage to pass off the error message display to the main thread.

There's one more place where our script thread could potentially call MessageBox, and that's in our IActiveScriptSite's OnScriptError. Let's rewrite it so that it GlobalAlloc()s the error message and then PostMessage()s it to the main thread to display. You can peruse the updated code in *IActiveScriptSite.c*.

And we need to add code to our main window procedure to handle WM_APP, as so:

```
case WM_APP:
{
   // Our script thread posts a WM_APP if it needs

   // us to display an error message.

   // wParam = A pointer to the string

   // to display. If 0, then lParam is an error

   // number to be passed to display_sys_error().

   // lParam = The HRESULT. If 0, then wParam

   //          is an allocated WCHAR string which we must

   // free with GlobalFree().

   if (!wParam)
      display_sys_error((DWORD)lParam);
   else if (!lParam)
   {
      MessageBoxW(hwnd, (const WCHAR *)wParam,
                  "Error", MB_OK|MB_ICONEXCLAMATION);
      GlobalFree((void *)wParam);
   }
   else
      display_COM_error((LPCTSTR)wParam, (HRESULT)lParam);
   return(0);
}
```

Note: You may wish to use RegisterWindowMessage to get your own custom message number, rather than using WM_APP. But for our purposes, WM_APP suffices.

There's just one thing left -- how to abort the script from the main thread. Let's do this in our WM_CLOSE processing, so if the user tries to close our window while a script is running, we'll force the script to abort. The engine IActiveScript's InterruptScriptThread function is one of the few functions that can be called by any thread. We pass the value SCRIPTTHREADID_ALL, which simply means to abort all running scripts we've given to the engine (i.e., if we had created numerous threads, each simultaneously running its own VBScript, this would cause the VB engine to abort all of those script threads). Alternately, if we wanted to abort only a particular script thread, we could pass that thread's ID.

```
case WM_CLOSE:
{
   // Is a script running?

   if (MyArgs.ThreadHandle)
   {
      // Abort the script by calling InterruptScriptThread.

      MyArgs.EngineActiveScript->lpVtbl->InterruptScriptThread(
          MyArgs.EngineActiveScript, SCRIPTTHREADID_ALL, 0, 0);
   ...
```

When InterruptScriptThread returns, that doesn't mean that the thread has terminated. It simply means that the engine has marked the running script for termination. We still have to "wait" for the thread to terminate. We'll do that by testing when ThreadHandle is 0. (Remember that the script thread zeroes it upon termination.) But there's one other problem. If the script thread is somehow "sleeping" or waiting for something itself, for example, in a call to MessageBox, then the engine will never get a chance to terminate it. We've been careful to avoid calling such functions ourselves, but note that VBScript has a msgbox function it too can call.

To get around this problem, we'll increment a count, and Sleep() in between increments. When the count "times out", we'll assume the script is locked up, and brute-force terminate it ourselves by calling TerminateThread.

```
wParam = 0;
while (MyArgs.ThreadHandle && ++wParam < 25) Sleep(100);
if (MyArgs.ThreadHandle) TerminateThread(MyArgs.ThreadHandle, 0);
```

> **In conclusion**, a script should be run in a separate thread than the main UI. The script thread must CoInitialize itself. Most of the engine's COM functions can be called only from the script thread. Our IActiveScriptSite's functions are also called within the script thread. The script thread should avoid doing anything that makes it "wait" or "pause". The UI thread may force the script to abort via InterruptScriptThread, but may also need to do a "time out" to brute-force terminate the script thread, if necessary.

Conclusion

This chapter demonstrates how to use an ActiveX Script Engine to run a script. But while it's useful to be able to simply run a script, we haven't yet seen how that script can directly interact with the functions in our app, and exchange data. For this, we'll need to add another COM object to our app. This will be the focus of the next chapter.

License

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

9 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/com_in_c6.aspx** to post and view comments on this article, or click **here** to get a print view with messages.