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

COM in plain C, part 8
By **Jeff Glatt**

Posted   : **6 Jan 2007**
Updated : **29 Mar 2007**
Views   : **21,736**

Miscellaneous Script Host details

🏅 **Prize winner in Competition "MFC/C++ Dec 2006"**

14 votes for this Article.

Popularity: 5.52 Rating: **4.82** out of 5   1 2 3 4 5

Contents

Introduction

In the preceding chapters, we learned how to create an ActiveX Script Host. Although those chapters covered the most important aspects of writing a script host, there are some additional, more esoteric features that your script host may want to utilize. This chapter will detail some of those more esoteric features.

Persistent script code

In our previous Script Host examples, we had a `runScript` function that opened the script engine, ran a script, and then closed down the engine. With this approach, the script got loaded/parsed, run, and then unloaded (after it was done running).

But there may be times when you'd like to add some scripts to the engine, and then leave them added to the engine even when those scripts are not running. Perhaps you'd like these scripts to be callable by any other script you run with that same engine's IActiveScript. In essence, maybe you'd like to treat these scripts like a collection of "ram-based macros". ActiveX Script Engines make this possible. But we have to alter our approach in two ways:

1. When we add a script as a "macro", we need to specify the SCRIPTTEXT_ISPERSISTENT flag to `ParseScriptText`. This tells the engine that the script is to remain internally parsed/loaded inside of the script engine, even after ParseScriptText returns.
2. We can't Release the engine's IActiveScript object until we're all done using our macros. At this point, those macros are finally unloaded.

The best approach to take is to add these macros while the engine is in its INITIALIZED state, but before set into its STARTED or CONNECTED states. `ParseScriptText` will not attempt to run the scripts. Instead, the scripts will be parsed for proper syntax, internally added to the script's engine, and remain there as `ParseScriptText` returns immediately. The scripts will remain in the engine even after called by another script, or run via any other mechanism, until such time as we finally release the engine's `IActiveScript` object.

In the directory **ScriptHost7**, you'll find an example to illustrate this point. We'll be adding a VB script to the engine, and specifying the `SCRIPTTEXT_ISPERSISTENT` flag. To keep things simple, this VB script will be embedded as global data in our EXE as so:

```
wchar_t VBmacro[] = L"Sub HelloWorld\r\nMsgBox \"Hello world\"\r\nEnd Sub";
```

All we have above is a VB subroutine named HelloWorld. It simply displays a message box.

Next, we'll embed a second VB script. This VB script will simply call the first script's HelloWorld subroutine:

```
wchar_t VBscript[] = L"HelloWorld";
```

When we add this second script to the engine, we won't specify the SCRIPTTEXT_ISPERSISTENT flag.

To make persistent scripts work, we need our `runScript` thread to keep the VB engine's IActiveScript around until our program terminates. To achieve this, we'll start this thread at the beginning of our program (instead of starting the thread only when it's time to run a script). This thread will remain alive until our program ends. Initially, the thread will call `CoCreateInstance` to get the VB engine's IActiveScript, then call its QueryInterface to get the engine's IActiveScriptParse, then call `InitNew` to initialize the engine, and finally call `SetScriptSite` to give the engine our IActiveScriptSite. This initialization is the same as our preceding host examples.

Then runScript will call `ParseScriptText` to add our "VB macro" to the engine. This is almost exactly like preceding example, except we specify SCRIPTTEXT_ISPERSISTENT:

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

After adding this script, the runScript thread now waits for the main thread to tell it to go ahead and run the second script (which will call this script we just loaded). The thread waits on an event signal we've created.

The main window has a "Run Script" button. When the user clicks upon this, the main thread sets this event signal:

```
// Let the script thread know that we want it to run a script

SetEvent(NotifySignal[0]);
```

runThread wakes up and calls `ParseScriptText` to add the second script, and then sets the VB engine state to SCRIPTSTATE_CONNECTED via a call to `SetScriptState`. This will run the second script, which will call the HelloWorld sub in the macro script to pop up a message box. After the user dismisses this box, then the first script ends and `SetScriptState` returns.

At this point, we do not Release the IActiveScriptParse and IActiveScript, and close the engine. Instead, we simply call `SetScriptState` again to set the VB engine's state to SCRIPTSTATE_INITIALIZED. This causes the second script to be unloaded. But the macro script is not unloaded, because it is persistent.

runThread goes back to sleep, waiting for the user to click the "Run Script" button again. In this event, runThread repeats the process of adding/running the second script. But note that there is no need to re-add the macro script. It is still loaded in the engine.

Here's the "script loop" in runThread:

```
for (;;)
{
    // Wait for main thread to signal us to run a script.

    WaitForSingleObject(NotifySignal, INFINITE);

    // Have the script engine parse our second script and add it to

    // the internal list of scripts to run. NOTE: We do NOT specify

    // SCRIPTTEXT_ISPERSISTENT so this script will be unloaded

    // when the engine goes back to INITIALIZED state.

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

    // Run all of the scripts that we added to the engine.

    EngineActiveScript->lpVtbl->SetScriptState(EngineActiveScript,
        SCRIPTSTATE_CONNECTED);

    // The above script has ended after SetScriptState returns. Now

    // let's set the engine state back to initialized to unload this

    // script. VBmacro[] remains still loaded.

    EngineActiveScript->lpVtbl->SetScriptState(EngineActiveScript,
        SCRIPTSTATE_INITIALIZED);
}
```

Script code and "Named Items"

In the above example, we added the macro script to the same "named item" as the second script. (Note: We didn't specify any particular named item, so the engine used its default "global item"). But it's possible to load scripts under different "named items".

In the preceding chapter, you'll recall that we made it possible for a script to call our own C functions by creating a named item (via the engine IActiveScript's AddNamedItem). The script used this item's name to call our C functions.

But that's not the only use of a named item. We can also group scripts under named items we create, and that's what we'll examine now.

Consider that we have two C source files named *File1.c* and *File2.c*. Here is the contents of *File1.c*:

```
// File1.c

static void MyFunction(void)
{
    printf("File1.c");
}

static void File1(void)
{
    MyFunction();
}
```

Here is the contents of *File2.c*:

```
// File2.c

static void MyFunction(const char *ptr)
{
    printf(ptr);
}

static void File2(void)
{
    MyFunction("File1.c");
}
```

There are a few things to note above.

1. Due to the static keywords, `MyFunction` in *File1.c* is not the same as `MyFunction` in *File2.c*. We can compile and link these two source files together and not have any problem (ie, there is no name conflict).
2. Due to the static keywords, the functions in *File1.c* cannot call the functions in *File2.c*, and vice versa.

When we create a named item (into which we'll load script code), think of it as creating a C source file. To create a named item, we call the engine IActiveScipt's `AddNamedItem`. Let's assume we have a script engine that implements the C language. First of all, we need to call AddNamedItem twice. The first time, we create a named item with the name *File1.c*. The second time we'll create a named item with the name *File2.c*. This creates the two "source files" in the engine. Then we'll call `ParseScriptText` to add the contents of *File1.c* to the *File1.c* named item. To do this, we must pass the name of the item as the third arg to ParseScriptText. Then, we'll add the contents of *File2.c* to the *File2.c* named item. Here is how we do this:

```
// Here's the contents of File1.c

wchar_t File1[] = L"static void MyFunction(void)\r\n\
{\r\n\
    printf(\"File1.c\");\r\n\
}\r\n\r\n\
static void File1(void)\r\n\
{\r\n\
    MyFunction();\r\n\
}";

// Here's the contents of File2.c

wchar_t File1[] = L"static void MyFunction(const char *ptr)\r\n\
{\r\n\
    printf(ptr);\r\n\
}\r\n\r\n\
static void File2(void)\r\n\
{\r\n\
    MyFunction(\"File1.c\");\r\n\
}";

// Create the File1.c named item. Error-checking omitted!

EngineActiveScript->lpVtbl->AddNamedItem(EngineActiveScript, "File1.c", 0);

// Create the File2.c named item.

EngineActiveScript->lpVtbl->AddNamedItem(EngineActiveScript, "File2.c", 0);

// Add the File1.c contents to the File1.c named object

activeScriptParse->lpVtbl->ParseScriptText(activeScriptParse, &File1[0],
    "File1.c", 0, 0, 0, 0, 0, 0);

// Add the File2.c contents to the File2.c named object

activeScriptParse->lpVtbl->ParseScriptText(activeScriptParse, &File2[0],
    "File2.c", 0, 0, 0, 0, 0, 0);
```

Now let's take our VB "macro script" and put it in a named item we'll create. We'll arbitrarily give this item a name of MyMacro. Here's what we do in runScript:

```
// The name of the named item

wchar_t MyMacroObjectName[] = L"MyMacro";

// Create the MyMacro named item

EngineActiveScript->lpVtbl->AddNamedItem(EngineActiveScript,
    &MyMacroObjectName[0],

SCRIPTITEM_ISVISIBLE|SCRIPTITEM_ISPERSISTENT);

// Add the contents of VBmacro to the MyMacro named item

activeScriptParse->lpVtbl->ParseScriptText(activeScriptParse, &VBmacro[0],
    &MyMacroObjectName[0], 0, 0, 0, 0,
     SCRIPTITEM_ISVISIBLE|SCRIPTTEXT_ISPERSISTENT, 0, 0);
```

You'll notice that we pass a few flags to AddNamedItem. We specify SCRIPTITEM_ISPERSISTENT because we don't want the engine to delete this named item (and its contents) when we reset the engine state to INITIALIZED. We also specify SCRIPTITEM_ISVISIBLE because we want this named item to be accessible from the default global item (which is where the second script gets added). Specifying SCRIPTITEM_ISVISIBLE is tantamount to removing the static keyword on our functions in that C language engine example. It allows a named item's functions to be callable by another named item's functions. Without SCRIPTITEM_ISVISIBLE, a named item's functions may call themselves, but not be called by any other named item's functions.

We have to modify the second VB script. Now it needs to reference the named item when it calls the HelloWorld subroutine. In VBscript, this is accomplished by using the name as if it's an object:

```
wchar_t VBscript[] = L"MyMacro.HelloWorld";
```

There is one more thing to do. When we call AddNamedItem to create "MyMacro", the engine is going to call our IActiveScriptSite's GetItemInfo, passing the name "MyMacro". We need to obtain and return an IDispatch pointer for this named item. Where do we get that? We get it by calling the engine IActiveScript's GetScriptDispatch, passing the item name. Here then is our IActiveScriptSite's GetItemInfo:

```
STDMETHODIMP GetItemInfo(MyRealIActiveScriptSite *this, LPCOLESTR
    objectName, DWORD dwReturnMask, IUnknown **objPtr, ITypeInfo **typeInfo)
{
    if (dwReturnMask & SCRIPTINFO_IUNKNOWN) *objPtr = 0;
    if (dwReturnMask & SCRIPTINFO_ITYPEINFO) *typeInfo = 0;

    // We assume that the named item the engine is asking for is our

    // "MyMacro" named item we created. We need to return the

    // IDispatch for this named item. Where do we get it? From the engine.

    // Specifically, we call the engine IActiveScript's GetScriptDispatch(),

    // passing objectName (which should be "MyMacro").

    if (dwReturnMask & SCRIPTINFO_IUNKNOWN)
        return(EngineActiveScript->lpVtbl->GetScriptDispatch(
            EngineActiveScript, objectName, objPtr));
    return(E_FAIL);
}
```

With the above minor modifications, we have now used a named item for our macro script. What is the benefit of this? First of all, our second script can now have a HelloWorld subroutine inside it, and this will not conflict with MyMacro's HelloWorld. So, we've now eliminated the possibility of any sub/function name conflicts between our macro script and the second script. Furthermore, if we had more macro scripts, we could put each one in its own named item. In this way, each macro script can have similarly named subs/functions, and again there is no conflict between any of the macro scripts. The VB engine knows which sub/function is being called because the object name used in the call indicates which named item holds the sub/function.

In conclusion, using named items is a way to prevent possible sub/function name conflicts in script code you

add to a particular engine.

Call a particular function in a script

In the above example, we called the engine's `GetScriptDispatch` to retrieve an IDispatch to a particular named item. We simply returned this to the engine.

But we could also use that IDispatch ourselves to directly call a particular VB sub/function (in a particular named item). To call the sub/function, we call that IDispatch's `GetIDsOfNames` and `Invoke` functions. This will be much like what we did in **IExampleApp3** when we used an IDispatch's Invoke to call some function in a COM object. You may wish to peruse that example again to refresh your memory. Let's say that we wish to directly call the HelloWorld sub in the MyMacros named object. First we need to get the IDispatch for that named object, which we do by calling the engine IActiveScript's `GetScriptDispatch`. Then we call that IDispatch's `GetIDsOfNames` to get the `DISPID` (ie, unique number) that the engine has assigned to the function we want to call. Finally, we use this DISPID with the IDispatch's `Invoke` to directly call the function.

```c
// NOTE: Error-checking omitted!

IDispatch
*objPtr;
DISPID      dispid;
OLECHAR
*funcName;
DISPPARAMS dspp;
VARIANT    ret;

// Get the IDispatch for "MyMacro" named item

EngineActiveScript->lpVtbl->GetScriptDispatch(EngineActiveScript,

"MyMacro", &objPtr);

// Now get the DISPID for the "HelloWorld" sub

funcName = (OLECHAR
*)L"HelloWorld";
objPtr->lpVtbl->GetIDsOfNames(objPtr, &IID_NULL,
&funcName, 1,
    LOCALE_USER_DEFAULT, &dispid);

// Call HelloWorld.

// Since HelloWorld has no args passed to it, we don't have to do

// any grotesque initialization of DISPPARAMS.

ZeroMemory(&dspp,
sizeof(DISPPARAMS));
VariantInit(&ret);
objPtr->lpVtbl->Invoke(objPtr,
dispid, &IID_NULL, LOCALE_USER_DEFAULT,
   DISPATCH_METHOD,
&dspp, &ret, 0, 0);
VariantClear(&ret);

// Release the IDispatch now that we made the call

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

In the directory **ScriptHost8** is an example that adds the following VB script (containing a main subroutine) to the VB engine:

```c
wchar_t VBscript[] = L"Sub main\r\nMsgBox \"Hello world\"\r\nEnd Sub";
```

Then we directly call this main routine. One thing you'll notice is that we don't create/specify any particular named item when we call `ParseScriptText`. This causes the script code to be added to the default "global named item". And so we need to fetch the IDispatch for this global named item. How do we do that? We pass a 0 to GetScriptDispatch for the name. This is a special value that tells GetScriptDispatch to return the global named item's IDispatch.

Query/Set a variable in a script

To query or set the value of a particular variable (in a particular named item), we do almost exactly the same as above. The only difference is that, in the call to Invoke, we specify the `DISPATCH_PROPERTYGET` flag if querying the value, or the `DISPATCH_PROPERTYPUT` if setting the value. Here then is an example of setting a variable named "MyVariable" within the "MyMacro" named item:

```
// NOTE: Error-checking omitted!

IDispatch  *objPtr;
DISPID     dispid, dispPropPut;
OLECHAR    *varName;
DISPPARAMS dspp;
VARIANT    arg;

// Get the IDispatch for "MyMacro" named item

EngineActiveScript->lpVtbl->GetScriptDispatch(EngineActiveScript,
    "MyMacro", &objPtr);

// Now get the DISPID for the "MyVariable" variable (ie, property)

varName = (OLECHAR *)L"MyVariable";
objPtr->lpVtbl->GetIDsOfNames(objPtr, &IID_NULL, &varName, 1,
    LOCALE_USER_DEFAULT, &dispid);

// Set the value to 10.

VariantInit(&arg);
ZeroMemory(&dspp, sizeof(DISPPARAMS));
dspp.cArgs = dspp.cNamedArgs = 1;
dispPropPut = DISPID_PROPERTYPUT;
dspp.rgdispidNamedArgs = &dispPropPut;
dspp.rgvarg = &arg;
arg.vt = VT_I4;
arg.lVal = 10;
objPtr->lpVtbl->Invoke(objPtr, dispid, &IID_NULL, LOCALE_USER_DEFAULT,
   DISPATCH_PROPERTYPUT, &dspp, 0, 0, 0);
VariantClear(&arg);

// Release the IDispatch now that we made the call

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

In the directory **ScriptHost9** is an example where we set the value of MyVariable and then call the main sub which displays this variable.

Interfacing different languages

It's possible to have a script written in one language call a script written in another language. For example, let's assume we have the following VBScript function that displays a message box:

```
Sub SayHello
    MsgBox "Hello World"
End Sub
```

And let's assume we have the following JScript function that calls the above VBScript function:

```
function main()
{
    SayHello();
}
```

First of all, because we're going to be using scripts in two different languages, JScript and VBScript, then we need to call `CoCreateInstance` twice; once to get the JScript engine's IActiveScript, and once to get the VBScript engine's IActiveScript. Of course, we'll store those two pointers in two separate variables (`JActiveScript` and `VBActiveScript` respectively).

We also need to get the IActiveScriptParse for each engine. And we need to call each engine's `SetScriptSite` to give it our IActivesScriptSite. (We could use a separate IActivesScriptSite for each engine, but our purposes, we'll use the same one for both engines since we won't be simultaneously running scripts in both languages. Rather, the VB engine is going to be running a script only when the JScript engine calls into that VBScript function).

In other words, `runScript` has to do all of the initialization that we must do to use a script engine, but do it once for each engine.

Then, we have to call the JScript engine's `ParseScriptText` to add the above JScript code to the JScript engine, and call the VBScript engine's `ParseScriptText` to add the above VBScript code to the VBScript engine. We'll add the code to the respective engine's global named item.

To facilitate the JScript calling a VBScript, we need to create a named object in the JScript engine that we'll associate with the VBScript engine. We do this before we add the scripts to their engines. Let's arbitrarily give this item a name of "VB".

```
JActiveScript->lpVtbl->AddNamedItem(JActiveScript, L"VB",
    SCRIPTITEM_GLOBALMEMBERS|SCRIPTITEM_ISVISIBLE);
```

Let's examine what happens when the JScript engine runs our JScript code above. The engine looks through all its JScript functions that we've loaded into the JScript engine. It sees that there is no JScript function named "SayHello". Since we've added some named item to the JScript engine (with the `ISVISIBLE` flag), the engine says to itself, "Hmmm. Maybe this SayHello function is inside of this named item. I need to get an IDispatch for this named item so I can call its GetIDsOfNames function, asking for a SayHello `DISPID`. If that IDispatch successfully gives me that DISPID, then I can call that IDispatch's Invoke to call that SayHello function."

And how does an engine get an IDispatch for a named item? By now, you should know that it calls our IActiveScriptSite's `GetItemInfo`. In this case, the JScript engine is going to pass an item name of "VB". Here's where it may seem a little confusing. When our GetItemInfo detects that it is being asked for that particular item, we're going to call the VBScript engine's `GetScriptDispatch` to get the VBscript engine's global named item. And that's what we're going to return to the JScript engine.

Yes, you read that correctly. When the JScript engine asks for the "VB" named item's IDispatch, we're actually going to return the VBScript engine's global named item IDispatch. Why? Because our VBScript SayHello function was added to the global named item, not some item named "VB". In other words, we're using the "VB" item as a "placeholder" in the JScript engine. The JScript engine doesn't need to know that it's "VB" item is essentially going to be the VBScript engine's global named item.

So, the JScript engine will call `GetIDsOfNames` of the VB engine's global named item IDispatch. And sure enough, the VB engine will return the `DISPID` for its SayHello function. When JScript calls that IDispatch's Invoke, it ends up calling into the VB engine to have the VB engine run the VB SayHello function.

Here then is our IActiveScriptSite's `GetItemInfo`:

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

    hr = E_FAIL;

    if (dwReturnMask & SCRIPTINFO_ITYPEINFO) *typeInfo = 0;

    if (dwReturnMask & SCRIPTINFO_IUNKNOWN)
    {
        *objPtr = 0;

        // If the engine is asking for our "VB" named item we created,

        // then we know this is the JScript engine calling. We need to

        // return the IDispatch for VBScript's "global named item".

        if (!lstrcmpW(objectName, L"VB"))
        {
            hr = VBActiveScript->lpVtbl->GetScriptDispatch(VBActiveScript,
                0, objPtr);
        }
    }

    return(hr);
}
```

In the directory **ScriptHost10** is an example of JScript calling VBScript.

Incidentally, you may be wondering about the SCRIPTITEM_GLOBALMEMBERS flag. You may recall when we previously dealt with a named item, the script had to reference that item's name as if it was an object name, for example:

```
VB.SayHello()
```

When you create an item with the SCRIPTITEM_GLOBALMEMBERS flag, then specifying the object name is optional. For example, either the above call works, or this also works:

```
SayHello()
```

So what we're doing here is making it possible for JScript to call SayHello as if it were just another "local" JScript function. In other words, it's more or less a notional shortcut that hides the messy details of named items.

But there is a price to be paid for this convenience. There may now be name conflicts between any items with SCRIPTITEM_GLOBALMEMBERS flag, as well as the global named item.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

11 messages have been posted for this article. Visit
http://www.codeproject.com/KB/COM/com_in_c8.aspx to post and view comments on this article, or
click **here** to get a print view with messages.