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

COM in plain C, Part 2

By Jeff Glatt

Posted : **20 Apr 2006**
Updated : **20 Apr 2006**
Views : **50,560**

How to write a COM component in C that can be used by script languages such as VBscript, Visual BASIC, jscript, etc.

- Download source files - 93.5 Kb

Contents

Introduction

In the first part of this series, we created a COM component in plain C, packaged it into a DLL, and learned how to register it so that it could be used by any C/C++ application. We also created a C/C++ include file (*IExample.h*) that contained our GUIDs and the definition of our object's VTable (functions). By including this *.H* file, a C/C++ application knows what the names of our object VTable's members are (to call our functions), what arguments to pass, what the functions return, and what the GUIDs of our object and its VTable are.

Whereas this is all the support a C/C++ app needs, we must add further support (to our `IExample` DLL) to accommodate interpreted "scripting" languages such as Visual Basic, VBScript, Jscript, Python, etc. The focus of this article is how to add such support. After we add this support, we'll write a VBScript example to use our updated `IExample` component.

Why script languages can't use our DLL as is?

Unless a particular language can read our C/C++ include file and extract the information in it, a Visual Basic, Python, etc. application simply does not know what functions are in our object, what arguments those functions are passed, and what they return, and what our GUIDs are. Most languages do not support extracting information from a C/C++ include file. We need some way of specifying all the information contained within *IExample.h* in a language-neutral format, preferably a compact "binary" format, unlike *IExample.h*. As we'll see later, we do this by creating something called a *type library*.

To make it easier and more standardized for a script engine to peruse the contents of our "type library", we'll add some functions, called `IDispatch` functions, to our object. The script engine gets the type library information only via these `IDispatch` functions. Only our object directly accesses its own type library. By not having the script engine directly access our type library, this means that if Microsoft develops later versions of a type library, these messy version details will be hidden from the script engine.

There's another matter with data types. Consider a string in ANSI C. A C string is a series of 8-bit bytes ending with a 0 byte. But that isn't how strings are stored in Pascal. A Pascal string starts with a byte that tells how

many more bytes follow. In other words, a Pascal string starts with a length byte, and then the rest of the bytes. There is no terminating 0 byte. And what about UNICODE versus ANSI? With UNICODE, every character in the string is actually 2 bytes (i.e., a `short`). Visual Basic strings are UNICODE, unlike ANSI C.

We need to deal with the problem of resolving these data type issues. We need to come up with a set of "generic data types" that work for any language that uses our object, and use only those data types for arguments to our object's functions (and any data our functions return).

Automation data types

In order to support any language (as well as extensions in each language such as UNICODE), Microsoft has defined a generic set of data types that a COM object should use. Microsoft calls these "Automation data types". (Later, we'll look at where the word Automation comes from.) One such automation data type is a `BSTR`. If a COM object is passed a string, then the argument can take the form of a `BSTR`. What is a `BSTR`? It's a pointer to a specially formatted string. Every character is two bytes (i.e., a `short`), and it is prefaced with an unsigned long that tells how many more shorts follow, plus it ends with a 0 short. This accommodates the "string" data type of almost every language/extension. But it also means that, a C/C++ app, or our C object itself, will sometimes need to reformat a C string to a null-terminated UNICODE string prefaced with a length.

Just for the sake of illustration, let's take the following ANSI C string:

```
char MyString[] = "Some text";
```

First, let's break up the string into separate chars:

```
char MyString[] = {'S','o','m','e',' ','t','e','x','t', 0};
```

Since C strings are null-terminated, notice that we include that last, 0 byte in the array.

Now, we need to convert the string to UNICODE. This means that every char becomes a short. We'll just redefine the array as `short`:

```
short MyString[] = {'S','o','m','e',' ','t','e','x','t', 0};
```

Now, we need to preface the array with the number of bytes in the array. Note that we don't count the terminating zero as `short`. So, we have a length of 9 `short`s * 2 bytes per `short` (note the Intel Little Endian byte order):

```
short MyString[] = {18, 0, 'S','o','m','e',' ','t','e','x','t', 0};
```

A `BSTR` is defined as a pointer to this specially formatted UNICODE string. Actually, it is a pointer to the third `short` in the UNICODE string, because the length is supposed to **preface** the actual string. So, here we declare a variable of type `BSTR`, and set it to point to our string above:

```
BSTR  strptr;

strptr = &MyString[2];
```

We can now pass `strptr` to a COM object that expects a `BSTR` as an argument. And if one of our object's functions expects a `BSTR` argument, `strptr` is what is going to be passed.

But before you think that you have to manually reformat all of your strings as above, that's not necessary. Fortunately, there is an operating system function called `MultiByteToWideChar` to convert an ANSI C string to UNICODE, and functions like `SysAllocString` and `SysAllocStringLen` which will allocate a buffer to copy the UNICODE string, prefaced with an unsigned long holding the size of that buffer. So to take our original `char` string and make a `BSTR` pointer to it (error checking omitted), we can do:

```
DWORD len;
BSTR  strptr;
char  MyString[] = "Some text";

// Get the length (in wide chars) of the UNICODE buffer we'll need to

// convert MyString to UNICODE.

len = MultiByteToWideChar(CP_ACP, 0, MyString, -1, 0, 0);

// Allocate a UNICODE buffer of the needed length. SysAllocStringLen will

// also allocate room for a terminating 0 short, and the unsigned long count.

// SysAllocStringLen will fill in the unsigned long with the value of

// "len * sizeof(wchar_t)" and then return a pointer to the third short in

// the buffer it allocates. All that is left for us to do is simply stuff a

// UNICODE version of our C string into this buffer.

strptr = SysAllocStringLen(0, len);

// Convert MyString to UNICODE in the buffer allocated by SysAllocStringLen.

MultiByteToWideChar(CP_ACP, 0, MyString, -1, strptr, len);

// strptr is now a pointer (BSTR) to an Automation string datatype.
```

Note: Somebody must later free that buffer allocated by `SysAllocStringLen`, by passing `strptr` to `SysFreeString`. Typically, that somebody is whomever called `SysAllocStringLen` to allocate it.

Note that since the `BSTR` points past the unsigned long size, and the string is null-terminated anyway, you can treat a `BSTR` as if it is merely a wide char (`wchar_t`) data type. For example, you can pass it to `lstrlenW` to get the length in wide chars. But also note that, it's entirely possible that the string could have embedded 0s in it. If you write a function that expects to be passed a human-readable text string, then it's a good assumption that the `BSTR` won't be pointing to something with embedded zeroes in it. On the other hand, if the `BSTR` is pointing to binary data, then this may not be a good assumption. In this case, you can use `SysStringLen` to determine the true length in wide chars (or `SysStringByteLen` if you need the length in bytes).

There are other "Automation data types" too, such as a `long` (i.e., a 32-bit value). So our object's functions are not restricted to being passed only pointers to Automation strings.

In fact, some languages support the concept of a function being passed an argument which could be a choice of data types. Let's say, we have a `Print()` function. And, let's say, that this `Print` function takes one argument, which can be either a string pointer (`BSTR`), or a `long`, or maybe a variety of other Automation data types, and it will print whatever it is passed regardless. For example, if passed a `BSTR`, it will print the characters of that string. If passed a `long`, it will first do something akin to calling:

```
sprintf(myBuffer, "%ld", thePassedLong)
```

...and then print out the resulting string in `myBuffer`.

This `Print` function needs some way to know whether it is being passed a string or a `long`. If we want to put this `Print` function into our object, we can have the application pass us another Automation data type called a `VARIANT`. A `VARIANT` is simply a struct that wraps another Automation data type. (This is the one type of "struct" that all languages supporting COM must internally understand and be able to create, simply for the purpose of passing arguments to an object's function, or receiving back data.)

A `VARIANT` has two members. The first member, `vt`, is set to some value that tells what other kind of data type your `VARIANT` is wrapping. The other member is a union, which is where you store the other data type.

So, if we wrap a `BSTR` into a `VARIANT` struct, then we set the `VARIANT`'s `vt` member to `VT_BSTR`, and store our `BSTR` (i.e., pointer to that specially formatted UNICODE string) in the `VARIANT`'s `bstrVal` member. If we

wrap a long, then we set the VARIANT's vt member to VT_I4, and store our long in the VARIANT's lVal member (which, due to the union, is really the same member as bstrVal).

> **In conclusion**, in order to support script languages, our object's functions must be written to be passed only a certain set of data types, called automation data types. Likewise, any data we return must be in the form of an automation data type.

If you look at the definition of our SetString function, you'll notice one problem. We defined it to be passed a char pointer as so:

```
static HRESULT STDMETHODCALLTYPE SetString(IExample *this, char *str)
```

This is an ANSI C string, which is not an automation data type. We need to replace this with a BSTR argument.

Our GetString function is even more troublesome for script languages. Not only is it defined to take an ANSI C string pointer, but the caller is expected to provide that non-UNICODE buffer, and our function modifies its contents. This is not automation compatible. When returning a BSTR to a script language, **we** must allocate the string and give it to the script engine. The script engine will call SysFreeString to free that string when done with it.

Given these requirements, the best thing to do is to change our IExample object so that its buffer member is a BSTR. When the script engine passes our SetString a BSTR, we'll make a copy of the string, and store this new BSTR in the buffer member. We'll then modify our GetString function to take a pointer to a BSTR (i.e., a handle) provided by the script engine. We'll create another copy of our string, and return it to the script engine (trusting that the script engine will free it).

So that we don't alter our original sources, I copied the contents of the *IExample* directory to a new directory named *IExample2*. I changed *IExample.c* to *IExample2.c*, and *IExample.h* to *IExample2.h*, etc.

In *IExample2.c*, let's rename our MyRealIExample struct to MyRealIExample2 (to differentiate it from our original sources), and change the buffer member data type. (We'll also change the name of the member to "string"). Here is its new definition:

```
typedef struct {
   IExample2Vtbl  *lpVtbl;
   DWORD          count;
   BSTR           string;
} MyRealIExample2;
```

And here are our updated SetString and GetString functions:

```
static HRESULT STDMETHODCALLTYPE SetString(IExample2

*this, BSTR str)
{
   // Make sure that caller passed a string

   if (!str) return(E_POINTER);

   // First, free any old BSTR we allocated

   if (((MyRealIExample2 *)this)->string)
       SysFreeString(((MyRealIExample2 *)this)->string);

   // Make a copy of the caller's string and save this new BSTR

   if (!((MyRealIExample2 *)this)->string =
            SysAllocStringLen(str, SysStringLen(str))))
      return(E_OUTOFMEMORY);

   return(NOERROR);
}

static HRESULT STDMETHODCALLTYPE GetString(IExample2 *this, BSTR *string)
{
   // Make sure that caller passed a handle

   if (!string) return(E_POINTER);

   // Create a copy of our string and return the BSTR in his handle. The

   // caller is responsible for freeing it

   if (!(*string = SysAllocString(((MyRealIExample2 *)this)->string)))
      return(E_OUTOFMEMORY);

   return(NOERROR);
}
```

Also, our `Release` function must free that allocated string right before we free our `MyRealIExample2`:

```
   if (((MyRealIExample2 *)this)->string)
           SysFreeString(((MyRealIExample2 *)this)->string);
   GlobalFree(this);
```

And finally, when our `IClassFactory` creates a `MyRealIExample2`, it should clear the string member:

```
((MyRealIExample2 *)thisobj)->string = 0;
```

Of course, we need to update these function definitions in *IExample2.h*:

```
// Extra functions

STDMETHOD  (SetString)   (THIS_ BSTR) PURE;
STDMETHOD  (GetString)   (THIS_ BSTR *) PURE;
```

We've now updated our object functions to use Automation data types.

The IDispatch functions

Our next step is to add the `IDispatch` functions to our object. There are four `IDispatch` functions, already defined by Microsoft. We must write these four functions, and then add pointers to them in our object's VTable.

Important note: In our VTable, these four pointers must be called `GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, and `Invoke`. They must appear in this order, after the three `IUnknown` functions (i.e., the first three pointers in our VTable), but before the pointers to any of our own extra functions (i.e., `SetString` and `GetString`). So, let's edit *IExample2.h*, and add the definitions for these four `IDispatch` functions. We'll change our object name from `IExample` to `IExample2`. Also, note that in the `DECLARE_INTERFACE_` line of

our macro, we substitute the word IDispatch for IUnknown. This indicates that we're adding the standard IDispatch functions to our object.

```
// IExample2's VTable

#undef  INTERFACE
#define INTERFACE IExample2
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  (SetString)              (THIS_ BSTR) PURE;
   STDMETHOD  (GetString)              (THIS_ BSTR *) PURE;
};
```

Because I'm creating another COM component, I ran *GUIDGEN.EXE* to create new GUIDs for my IExample2 and IExample2VTbl, and pasted them into *IExample2.h*. This allows us to test this new DLL without needing to uninstall our original DLL, nor have the new DLL interfere with the original. Otherwise, if we used the same GUIDs, we'd have two registered COM components (DLLs) with the same GUIDs. Not a good thing.

I also ran *GUIDGEN.EXE* a third time to generate a third GUID. We're going to need this for our type library. I added it to *IExample2.h*, and gave it a C variable name of CLSID_TypeLib.

Now, we need to write those four IDispatch functions. Fortunately, Microsoft's OLE32 DLL has some generic functions we can call that do most of the work for these four functions. The only real work we need to do is to load our type library (which we'll be creating later in this article), and create a COM object called an ITypeInfo to give to a script engine. But even there, Microsoft provides some functions to do a lot of that work, namely LoadRegTypeLib to load the type library, and GetTypeInfoOfGuid to create an ITypeInfo.

Our GetTypeInfoCount function is easy. The caller passes us a handle to a UINT. We fill this in with a 1 if we have a type library, or 0 if we don't. We do:

```
ULONG STDMETHODCALLTYPE GetTypeInfoCount(IExample2 *this, UINT *pCount)
{
   *pCount = 1;
   return(S_OK);
}
```

Our GetTypeInfo function must return (to the script engine) a pointer to an ITypeInfo object that we create from our type library. What does an ITypeInfo object do? It manages the script engine's access to all of the information loaded from our type library. Remember, we said that we're not going to let anyone directly load and access our type library. Instead, we give the script engine an ITypeInfo object, whose functions parse our type library information and return specific information about a function in our IExample2 object. For example, once we give our ITypeInfo to the script engine, that engine can call our ITypeInfo's GetFuncDesc function to determine how many arguments are passed to our GetString function, what each type of argument is, and what GetString returns if anything.

"Oh great. More functions to write", you must be thinking. Yes, we could write all of the necessary functions for an ITypeInfo object, and put them in *IExample2.c*. We could then have IExample2's GetTypeInfo function allocate one of our ITypeInfo objects, initialize it (i.e., store its VTable in it), and return it to the script engine.

But, we're not going to do this. We're going to save ourselves a bit of work. Microsoft has already written some generic ITypeInfo functions (contained in the OLE DLLs), and gives us a function we can call that will allocate an ITypeInfo for us, and fill in its VTable to these generic functions. We can then give this "generic" ITypeInfo to the script engine. All we need to do is load our type library into memory and "give" it to this generic ITypeInfo, and from then on, Microsoft's functions will handle doling out the information to the script engine. In order to take advantage of Microsoft's generic ITypeInfo functions, we must adhere to certain rules. We must make sure our IExample2 object is defined as something called a *dual interface*. Let's not worry right now what that is. But we're going to adhere to this requirement.

Before we dive into our GetTypeInfo function, let me say one thing about a type library. It can contain strings that are meant to be displayed to humans. For example, if we want, we could tag our GetString function with a "text description" that reads: *This function returns an IExample2 object's string*. Then, for example, the Visual BASIC IDE could get our ITypeInfo object, call its GetDocumentation function to retrieve this description for GetString, and display it to a VB programmer who wants to call our IExample's GetString. There are many human languages, so just like the resources for your EXE/DLL can be in different languages, so too does a type library have a "Language ID" (LCID). For this article, we're going to create only an English type library, and ignore any issues concerning loading a particular language type library. Just be aware that a script engine may pass our GetTypeInfo function a LCID other than English, if it desires another language.

GetTypeInfo is defined as so:

```
ULONG STDMETHODCALLTYPE GetTypeInfo(IExample2 *this,
      UINT itinfo, LCID lcid, ITypeInfo **pTypeInfo)
```

The itinfo argument is not used, and should be 0. The LCID specifies what language type library to use. Again, we ignore this since we're going to deal only with English. The final argument is where the script engine expects us to return a pointer to our ITypeInfo object.

We only ever need one ITypeInfo object because its contents do not change. So our GetTypeInfo function will simply make sure that our type library is loaded, and that we have gotten a pointer to Microsoft's generic ITypeInfo. We'll store this pointer in a global variable. We do need to increment the ITypeInfo's reference count (i.e., call its AddRef function) every time the script engine calls our GetTypeInfo. (And of course, we expect the script engine to call ITypeInfo's Release when done with the ITypeInfo). So, here is our GetTypeInfo function, and a helper function (loadMyTypeInfo) we write that does the real work of loading our type library and getting Microsoft's generic ITypeInfo. Note how our helper function calls the OLE API LoadRegTypeLib to do all the work of loading our type library. All we need to do is pass the GUID that I created for our type library (and put in *IExample2.h*, giving it the variable name CLSID_TypeLib).

```
ULONG STDMETHODCALLTYPE GetTypeInfo(IExample2 *this,
      UINT itinfo, LCID lcid, ITypeInfo **pTypeInfo)
{
   HRESULT  hr;

   // Assume an error.

   *pTypeInfo = 0;

   if (itinfo) hr = ResultFromScode(DISP_E_BADINDEX);

   // If our ITypeInfo is already created,

   // just increment its ref count.

   else if (MyTypeInfo)
   {
      MyTypeInfo->lpVtbl->AddRef(MyTypeInfo);
      hr = 0;
   }
   else
   {
      // Load our type library and

      // get Microsoft's generic ITypeInfo object.

      hr = loadMyTypeInfo(this);
   }

   if (!hr) *pTypeInfo = MyTypeInfo;

   return(hr);
}

static HRESULT loadMyTypeInfo(void)
{
   HRESULT   hr;
   LPTYPELIB pTypeLib;

   // Load our type library and get a ptr to its TYPELIB. Note: This does an

   // implicit pTypeLib->lpVtbl->AddRef(pTypeLib).

   if (!(hr = LoadRegTypeLib(&CLSID_TypeLib, 1, 0, 0, &pTypeLib)))
   {
      // Get Microsoft's generic ITypeInfo,

      // giving it our loaded type library. We only

      // need one of these, and we'll store

      // it in a global. Tell Microsoft this is for

      // our IExample2's VTable, by passing that VTable's GUID.

      if (!(hr = pTypeLib->lpVtbl->GetTypeInfoOfGuid(
            pTypeLib, &IID_IExample2, &MyTypeInfo)))
      {
         // We no longer need the ptr

         // to the TYPELIB now that we've given it

         // to Microsoft's generic ITypeInfo.

         // Note: The generic ITypeInfo has done

         // a pTypeLib->lpVtbl->AddRef(pTypeLib),

         // so this TYPELIB ain't going away

         // until the generic ITypeInfo does

         // a pTypeLib->lpVtbl->Release too.

         pTypeLib->lpVtbl->Release(pTypeLib);

         // Since caller wants us to return our ITypeInfo pointer,

         // we need to increment its reference count. Caller is
```

When we later create our type library, we'll need to assign each one of our own functions we add to IExample2 (i.e., GetString and SetString) a numeric value, which Microsoft refers to as a DISPID (Dispatch ID). This can be any value of our choosing, but each function must have a unique DISPID. (Note: Do not use the value 0, or negative values). We'll arbitrarily assign a DISPID of 1 to GetString, and 2 to SetString. Script engines internally use this DISPID for faster calls to our functions, since it's quicker to lookup a function in some table via matching a numeric value versus comparison of a function name string.

Next, we write our GetIDsOfNames function. The purpose of this function is to allow the script engine to pass us the name of one of our own functions we added to IExample2 (i.e., GetString or SetString), and we'll return the DISPID we assigned to that function. For example, if the engine passes us "GetString", then we'll return the DISPID 1.

Because we're going to define our IExample2 as a "dual interface" when we create our type library, we can then call the OLE function DispGetIDsOfNames, which will do all the work that our GetIDsOfNames needs to do (i.e., parsing the info in our type library to look up the DISPID we assigned to a particular function). The only thing we need to do is make sure that we got a ITypeInfo, because that needs to be passed to DispGetIDsOfNames. So, here is our GetIDsOfNames:

```
ULONG STDMETHODCALLTYPE GetIDsOfNames(IExample2 *this,
      REFIID riid, LPOLESTR *rgszNames, UINT cNames,
      LCID lcid, DISPID *rgdispid)
{
   if (!MyTypeInfo)
   {
      HRESULT hr;

      if ((hr = loadMyTypeInfo())) return(hr);
   }

   // Let OLE32.DLL's DispGetIDsOfNames()

   // do all the real work of using our type

   // library to look up the DISPID

   // of the requested function in our object

   return(DispGetIDsOfNames(MyTypeInfo, rgszNames,
                            cNames, rgdispid));
}
```

We have one last IDispatch function to write - Invoke. This is what the script engine calls to (indirectly) call another one of the functions we added to IExample2 (i.e., GetString or SetString). The script engine passes the DISPID for the function it wants to call (i.e., 1 for GetString, or 2 for SetString). The engine also passes an array of VARIANT structs, filled in with the arguments to be passed to GetString or SetString. It also passes a VARIANT we need to fill in with any return value from the function (not the HRESULT return from Invoke, but rather, whatever our type library indicates is the return value from GetString or SetString).

So, what our Invoke function does is first look at the DISPID. If it's 1, we know that we have to call GetString. If it's 2, we know we have to call SetString. So immediately, you should be thinking *case statement*. And it could be a big case statement if you've added lots of functions to IExample2.

Next, we need to pull those arguments out of the VARIANTs and call the correct function, passing it the arguments. For example, if the script is calling SetString, we need to pull the BSTR out of the first VARIANT and pass it to SetString.

Here's a problem. A script engine is allowed to pass a VARIANT that wraps what may not be the exact data type we expect, but one that can be coerced into the correct type. Our SetString expects a BSTR (pointer to a string). But, let's assume the script engine wants to set our string member to the string "10". Now, this is how the engine could setup the VARIANT passed to us:

```
VARIANT myArg;

myArg.vt = VT_BSTR;
myArg.bstrVal = SysAllocString(L"10");
```

OK, we're good to go. It's passing us a BSTR wrapped in that VARIANT.

But the engine could instead pass us a long, as so:

```
VARIANT myArg;

myArg.vt = VT_I4;
myArg.lVal = 10;
```

Now, our Invoke function has to see that this is not VT_BSTR, and do the conversion ourselves before we call SetString (error checking omitted):

```
// Determine which function to call based upon the DISPID

// the caller passed us.

switch (dispid)
{
   // GetString()

   case 1:
      // Here we'd call GetString with the appropriate args.

      break;

   // SetString()

   case 2:
      // Did he pass the BSTR that SetString expects?

      if (params->rgdispidNamedArgs[0]->vt != VT_BSTR)
      {
         // Nope. We've got to try to convert whatever he passed

         // into a BSTR.


         // Did he pass a long?

         if (params->rgdispidNamedArgs[0]->vt == VT_I4)
         {
             // Create a BSTR out of that long.

             wchar_t temp[33];

             wsprintW(temp, L"%d", params->rgdispidNamedArgs[0]->lVal);
             params->rgdispidNamedArgs[0]->bstrVal = SysAllocString(temp);

             // NOTE: The engine will free this BSTR when it calls

             // VariantClear() on params->rgdispidNamedArgs[0].

         }
         else
         {
            // Here we need to check/convert even more possible datatypes!

            // If we exhaust the logical possibilites, return DISP_E_TYPEMISMATCH.

         }
     }

     // Call SetString with the BSTR arg.

     return(this->lpVtbl->SetString(this, params->rgdispidNamedArgs[0]->bStrVal));
}
```

"Oh great! This could turn into major work for me, given there are lots of functions!", you're thinking. Yes, it could. But at least, Microsoft does provide a bunch of OLE APIs to convert a VARIANT from one type to another. So, we can simply do:

```
// SetString()

case 2:
   if ((hr = VariantChangeType(¶ms->rgdispidNamedArgs[0],
       ¶ms->rgdispidNamedArgs[0], 0, VT_BSTR))) return(hr);
   return(this->lpVtbl->SetString(this,
          params->rgdispidNamedArgs[0]->bStrVal));
```

But, here's something even better. Because we're going to define IExample2 as a dual interface, we can call the Microsoft function DispInvoke to do **all** of the work for us. It uses our type library to match that DISPID with the correct function to call, checks/converts the arguments, calls the correct function, and even messages the return value into the VARIANT the engine passed for that purpose. The only thing we need to do, like in our GetIDsOfNames function, is make sure that we have got a ITypeInfo because that needs to be passed to the DispInvoke. So, here's our Invoke:

```
ULONG STDMETHODCALLTYPE Invoke(IExample2 *this,
      DISPID dispid, REFIID riid, LCID lcid,
      WORD wFlags, DISPPARAMS *params, VARIANT *result,
      EXCEPINFO *pexcepinfo, UINT *puArgErr)
{
   // We implement only a "default" interface

   if (!IsEqualIID(riid, &IID_NULL))
      return(DISP_E_UNKNOWNINTERFACE);

   // We need our ITypeInfo (to pass to DispInvoke)

   if (!MyTypeInfo)
   {
      HRESULT hr;

      if ((hr = loadMyTypeInfo())) return(hr);
   }

   // Let OLE32.DLL's DispInvoke() do all

   // the real work of calling the appropriate

   // function in our object, and massaging

   // the passed args into the correct format

   return(DispInvoke(this, MyTypeInfo, dispid, wFlags,
          params, result, pexcepinfo, puArgErr));
}
```

And, that takes care of all the IDispatch functions.

> **In conclusion**, in order to support script languages, our object's VTable must include the four standard IDispatch functions. They must appear immediately after the first three, IUnknown functions (QueryInterface, AddRef, and Release). To make it easier to implement them, we can define our VTable as "dual interface" (in our type library) and then call some Microsoft OLE functions to do most of the real work.

The type library

Our next step is to create our type library. First, we need to create the "source" file that contains the definitions of our object and its VTable, and their GUIDs, much like we created *IExample.h* for the benefit of our C compiler. But, the format of this source file will be different than *IExample.h*, and we will compile it with a special utility called *MIDL.EXE* (which ships with your compiler, or can be found in Microsoft's Windows Platform SDK).

Typically, this source file's name ends with the extension *.IDL*. If so, when we add it to our Visual C++ project,

Visual C++'s IDE will automatically invoke *MIDL.EXE* on it to turn it into a binary file whose name ends with the extension *.TLB*. That TLB file is our type library.

In the *IExample2* directory, you will find the file *IExample2_extra.idl* which is the source for our type library. The syntax has some similarities with C++. A line preceded with `//` is a comment. And, C style opening and closing braces `{` and `}` are used to contain a particular "structure" or entity.

The IDL begins with the following lines:

```
[uuid(E1124082-5FCD-4a66-82A6-755E4D45A9FC), version(1.0),

helpstring("IExample2 COM server")]
library IExample2
{
```

The first line is enclosed in brackets. It contains three particular bits of info, each separated by a comma. The first bit of info is the GUID for our type library. Note that it is the same GUID as specified in *IExample2.h* for `CLSID_TypeLib` (i.e., it's that extra GUID I generated to use for our type library).

Next, is the version number for our type library (which I've chosen as 1.0).

Last, is any help description (for our type library) that we'd like some object browser to display to a programmer (such as Visual BASIC's object browser may do). Here, I identity what my COM component is/does. For example, if you were creating a COM component that reads UPC bar codes, perhaps your help string would be "Acme Brand UPC bar code reader".

The next line must start with a `library` keyword. After this, you can give some name to the type library. I just chose `IExample2`, but it could be anything of your choice. Note that this is not related to the actual filename of the TLB file.

Finally, we have an opening brace. Everything between this and the final closing brace in our IDL source will be part of our type library. This is where we put the definitions of our `IExample2` object and its VTable.

The next line:

```
importlib("STDOLE2.TLB");
```

is like a `#include` statement in a *.H* file. It includes the contents of Microsoft's *STDOLE2.TLB* file (shipped with your compiler, or in the SDK) which defines basic COM objects such as `IUnknown` and `IDispatch`. We need to reference this since our `IExample2` VTable has the `IDispatch` functions (and therefore the `IUnknown` functions too) appearing at the start of it.

What follows is the start of our `IExample2`'s VTable definition. It looks like so:

```
[uuid(B6127C55-AC5F-4ba0-AFF6-7220C95EEF4D), dual,
      oleautomation, hidden, nonextensible]
interface IExample2Vtbl : IDispatch
{
```

The first line is between brackets, and it contains the GUID for our `IExample2`'s VTable (which you'll note is the same GUID identified as `IID_IExample2` in *IExample2.h*; it must be so).

This line also contains several keywords that describe our VTable. The most important keyword is "`dual`". This is how we indicate that `IExample2` is a "dual interface". What this means is that, in addition to allowing some C/C++ application to directly call our `SetString` and `GetString` functions via `IExample2`'s VTable (like you saw in the demo C application with the first article), that same VTable also has the four `IDispatch` functions in it which a script engine such as VBScript can use to (indirectly) call `SetString` and `GetString` via `Invoke`. That's why it is called "`dual`". This same VTable works for both C/C++ apps that want to call our `SetString` and `GetString` functions directly, as well as a script engine that needs to call them via our `Invoke` function.

The `oleautomation` keyword means that we are using only Automation data types for arguments passed to the `SetString` and `GetString` functions, and also for any returned values.

Note: The `dual` keyword automatically implies `oleautomation`. So, we technically don't need to specify `oleautomation`. But, I do anyway. If you identify a VTable with the `oleautomation` keyword, and then you try to use some non-Automation data type as an argument (or a return value) in one of your functions, the *MIDL.EXE* will give an error. Note that older versions of *MIDL.EXE* may flag newer Automation data types as an error, so make sure you use the *MIDL.EXE* in the latest Platform SDK.

The `hidden` keyword simply means that we don't want our VTable itself to be listed in some object browser. We want to show only our `IExample2` object, which we'll define presently.

The `nonextensible` keyword means that we won't be doing something unexpected like adding extra functions (which aren't defined in our type library) to `IExample2`'s VTable when we actually allocate and give an `IExample2` object to someone.

The next line must start with `interface` to indicate that we're defining a VTable. This is followed by whatever name we decide to give to this VTable in our IDL source. We could call it `IExample2Vtbl`, or whatever. I went with "`IExample2Vtbl`". We also specify the `IDispatch` type since our `IExample2` VTable does, in fact, start with an `IDispatch` (and of course, an `IUnknown` before that, since `IDispatch` implies `IUnknown` too).

Note: If our VTable did not have the four `IDispatch` functions, then we would define it as the `IUnknown` type instead.

Finally, we have an opening C brace. After this is where we list all of the functions in our VTable. It looks like this:

```
[helpstring("Sets the test string.")]
[id(1)] HRESULT SetString([in] BSTR str);
[helpstring("Gets the test string.")]
[id(2)] HRESULT GetString([out, retval] BSTR *strptr);
```

First of all, you'll note that we don't define the `IUnknown` (`QueryInterface`, `AddRef`, and `Release`) nor the `IDispatch` (`GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, and `Invoke`) functions. We don't have to, because we specified `IDispatch` as our VTable type, so these automatically get added by the *MIDL.EXE* (which knows about those functions because we imported *STDOLE.TLB*).

So, we list only those functions we added to the end of `IExample2`'s VTable (i.e., `SetString` and `GetString`). These **must** be listed in the same order as they appear in *IExample2.h*'s definition of our VTable.

We've specified a help string before each function. Again, this is just text that an object browser would display to a programmer.

For each function, we first specify the DISPID that we choose to give it. Remember that we arbitrarily decided to choose 1 for `SetString` and 2 for `GetString`.

The rest of the definition for each function looks the same as in *IExample2.h*, except for one thing. Before each argument, we have to specify whether the argument is something that is initialized to a particular value by the script engine before it is passed to us (i.e., an `[in]` argument), or is an un-initialized argument that the engine expects us to fill in with some value (i.e., an `[out]` argument). Or maybe, it's even supposed to be an argument that the script engine initializes to some value, and then also expects us to modify it with a new value (i.e., an `[in, out]` argument). Or, maybe it can be an argument that is passed to us, which the engine expects our `Invoke` function to store in that special `VARIANT` the engine passes for the purpose of returning the value (i.e., an `[out, retval]` argument).

`SetString` expects to be passed a `BSTR` that the engine has set to a particular value. So, we identify that as `[in]`.

Since `GetString` expects to be passed a `BSTR` pointer that we will fill in with a copy of a string we allocated, we should specify this as an `[out]` argument. Unfortunately, for reasons I won't get into here, Microsoft doesn't allow simply `[out]`. We must either use `[in, out]` or `[out, retval]`. The difference between the two is whether we want our allocated `BSTR` to be returned to the script as the return value of its call to `GetString` (i.e., `[out, retval]`), or whether the script will pass the name of some variable to `GetString` to be filled in (i.e., `[in, out]`). In other words, it depends upon how we want the script to call our `GetString` function. We have two choices, and here are the choices illustrated using VBScript:

```
newString = GetString()  ' [out, retval]

GetString(newString)     ' [in, out]
```

The former looks more natural to me, so we'll go ahead and define the argument passed to `GetString` as `[out, retval]`.

Note: Only one argument passed to your function can be marked as `[out, retval]`. The remaining arguments must be either `[in]` or `[in, out]`.

We end the definition of our VTable with a closing brace.

Now, we get to the definition of our `IExample2` object. It looks like so:

```
[uuid(520F4CFD-61C6-4eed-8004-C26D514D3D19),
      helpstring("IExample2 object."), appobject]
coclass IExample2
{
   [default] interface IExample2Vtbl;
}
```

The first line specifies the GUID for our `IExample2` object. Again, note that it is the same GUID defined in *IExample2.h* as `CLSID_IExample2`, and it must be so.

We specify a help string that lets an object browser know what this particular object is/does.

The `appobject` keyword indicates that our `IExample2` object can be gotten via `CoGetClassObject`.

The next line must start with `coclass`, followed by what we decide to call our object in this IDL source.

Within the braces, we put a line that indicates that our object begins with the VTable that we identified as "`IExample2Vtbl`" in our IDL file. This line contains the `interface` keyword, followed by the name of our VTable. We preface it with the `default` keyword (enclosed in brackets) to indicate that this is the VTable a script engine should assume is the one that it will get in its `IExample2` object. (We haven't talked about this yet, but it's possible for an object to have pointers to more than one VTable inside it. This is known as "multiple interfaces". But trying to get such an object to work with script engines is a major hassle, so we'll avoid it altogether. Even though we have only one VTable in `IExample2`, we still need to mark it as the default one.)

We finish up with a couple closing braces, and our IDL file is done. You can compile it with *MIDL.EXE* (or add the *IExample2_extra.idl* to your Visual C++ project and let the IDE run *MIDL.EXE* on it). If there are no problems, you should end up with a binary file named *IExample2.tlb*. That's our type library.

It's possible to embed that *.TLB* file in the resources of our DLL, and have it loaded from there. But I like to keep the *.TLB* separate from the DLL. Why? Because C/C++ applications don't need the *.TLB*, nor do compiled Visual BASIC apps. The *.TLB* is needed only for interpreted Visual BASIC, VBScript, and other such interpreted languages. (And you do need to distribute the type library for those programmers.)

> **In conclusion**, an IDL file contains the "source code" for our type library. We put the definitions of our objects and their VTables in this file, much like we do in a C include file, but in a slightly different format. It is compiled with *MIDL.EXE* to create a *.TLB* file which is our actual, binary type library. This *.TLB* file can either be embedded in our DLL's resources, or shipped along with our DLL, to allow script engines to use our component.

Registering the type library

We're done with our updated component. Of course, before anyone can use it, we need to install/register it. In the first article, you'll recall, we wrote a utility to register our component. We need to make a very simple addition to that code, because our type library (*.TLB* file) needs to be registered too. So, we'll do that in the same installation utility. Microsoft provides an API to do all of the work of registering a type library: `RegisterTypeLib`.

But before we look at that, there's another matter to discuss. Visual BASIC programmers don't like to deal with complicated things like "big numbers". A GUID looks too scary to them. So, Microsoft decided that COM

developers could associate a "Product ID" (ProdID) with their component. And instead of using our `IExample2` GUID to create our object, a Visual BASIC programmer will use our ProdID instead. So what's a ProdID? It's just some unique string you choose to identify your component. Typically, you'll take your main object name (here, we have `IExample2`), and append some interface name to it, and maybe even a version number, each separated by a dot. I'm just going to arbitrarily choose **IExample2.object** as my ProdID. In order to associate this ProdID with our `IExample2`, we need to set a couple extra registry keys. We'll do that in our installation utility as well. All we do is create a registry key named "IExample2.object" (i.e., our chosen ProdID). Then, we create a subkey named "CLSID" under that, and set its default value to our `IExample2` GUID. The only tricky part is that the GUID has to be formatted into a human-readable string.

In the directory *RegIExample2*, there is an updated installation utility which creates the extra ProdID registry keys, and calls RegisterTypeLib to register our type library. The additions are trivial, so you can peruse the commented code to study the details.

After running this installation utility, a script engine can now use our component.

Of course, our uninstall utility needs to be updated to delete the extra registry keys. And, Microsoft provides **UnRegisterTypeLib** to undo what RegisterTypeLib does. Peruse the updated code in the directory *UnregIExample2*.

> **In conclusion**, the type library (i.e., the TLB file) must be registered, just like our `IExample` object itself. This is easily done with RegisterTypeLib. Furthermore, we should choose a ProdID, which is just a non-scary string that a VB programmer can use in lieu of our object's GUID, and set up some registry keys to associate this ProdID with our GUID.

A VBScript example

Let's take a trivial VBScript example of using `IExample2`. We'll simply call `SetString` to set `IExample2`'s string member to "Hello World". Then, we'll call `GetString` to retrieve that string (into another VBScript variable) and display it (so we can be sure it was set/retrieved properly).

```
set myObj = CreateObject("IExample2.object")
myObj.SetString("Hello world")
copy = myObj.GetString()
MsgBox copy, 0, "GetString return"
```

First, the VBScript gets one of our `IExample2` objects. The script calls the VB engine's `CreateObject`. Note that our `IExample2` ProdID has been passed. `CreateObject` first calls `CLSIDFromProgID` to look up `IExample2`'s GUID from that ProdID. Then, `CreateObject` calls `CoCreateInstance` to get a pointer to an `IExample2` that our DLL allocates. The script stores this in the variable `myObj`.

Next, the script calls our `IExample2`'s `SetString`, passing the string "Hello World". The script engine does this by calling `IExample2`'s `GetIDsOfNames` to get the DISPID for `SetString`, then calls `IExample2`'s `Invoke`, passing a `VARIANT` wrapping a `BSTR` of "Hello World".

Next, the script calls our `IExample2`'s `GetString`. Again, the script engine calls `GetIDsOfNames` and `Invoke`. The script stores the string we (i.e., `Invoke`) return in the variable `copy`.

Finally, the script displays what `GetString` returned.

But don't run this script yet. We're going to change something about our component first.

A Property

Consider the following C code:

```
typedef struct {
   char *  string;
} IExample;

IExample example;

example.string = "Hello World";
```

Wouldn't it be nice if VBScript could do a similar thing to set our `IExample2`'s string member? For example, instead of calling `SetString`, the VBScript could do this:

```
myObj.string = "Hello World"
```

But although it looks like the script is directly accessing our `IExample2`'s string member, this would internally translate into a call to our `SetString`, passing "Hello World".

And so too, the script could do this to retrieve the value of `IExample2`'s string member (and assign it to the `copy` variable):

```
copy = myObj.string
```

And again, this would internally translate to a call to our `GetString`, and the return value is assigned to `copy`.

Can we do that? Yes, we can. But we need to change the definitions of `SetString` and `GetString`, not in our C code, nor in our *.H* file, but rather, in our *.IDL* file. Since `SetString` sets the value of a member, we need to mark it as a "property set" function. And since `GetString` retrieves the value of a member, we need to mark it as a "property get" function. In the IDL file, we also rename `GetString` and `SetString` to both be what member name we want the VBScript to use. Here, we'll choose "string". So, here is how we change the definitions in our IDL file:

```
[id(1), propput] HRESULT string([in] BSTR);
[id(1), propget] HRESULT string([out, retval] BSTR *);
```

All we've done is add "`propput`" to `SetString` (and change the function name to "string"). We also added "`propget`" to `GetString`, and changed its function name to "string" as well.

"Wait a minute. How can you have two functions with the same name?", you may ask. We don't. The real function names are still `SetString` and `GetString`. After all, these appear in *IExample2.h*'s VTable definition in the same place that they appear in our IDL VTable definition. It's only that our type library considers the names to both be "string", and it knows that they are two different functions because one sets the value (`propput`) and the other retrieves the value (`propget`).

But in order to declare a function as `propget`, it must take exactly two arguments. First, of course, is a pointer to our object. The second argument must be declared as `[in]`, and it should be the value to set the member. For example, since `IExample2`'s string member needs to be set to a `BSTR` (pointer), we define the argument as a `BSTR`. If the string member had been defined as a `long`, we would have defined the argument as "`long`". As long as we stick to automation data types, we're OK.

And in order to declare a function as `propput`, it must take exactly two arguments. First, of course, is a pointer to our object. The second argument must be declared as `[out, retval]`, and it should be a pointer to where we return the member's value. For example, since `IExample2`'s string member is a `BSTR` (pointer), we define the argument as a `BSTR *` (which is actually a handle since `BSTR` itself is a pointer). If the string member had been defined as a `long`, we would have defined the argument as "`long *`".

I've changed the above two lines in *IExample2_extra.idl*, and renamed it to *IExample2.idl*. Run *MIDL.EXE* on it to produce a new *.TLB* file.

Now, we can rewrite our VBScript example as so:

```
set myObj = CreateObject("IExample2.object")
myObj.string = "Hello World"
MsgBox myObj.string, 0, "GetString return"
```

> **In conclusion**, if you wish to allow a script to set the value of some data member of your object, add a function to set the value, and mark it as `propput` in the IDL file. In the IDL file, the function name will simply be the member name. The second argument passed to the function should be marked `[in]` and be the desired value. If you wish to allow a script to retrieve the value of some data member of your object, add a function to retrieve the value, and mark it as `propget` in the IDL file. In the IDL file, the function name will simply be the member name. The second argument passed to the function should be marked `[out, retval]` and be a pointer to where you return the value.

Note: You do not need to supply both a `propget` and a `propset` for a given member. You can supply one or the other. For example, if we wanted to make the string member a read-only value, then we would eliminate our `SetString` function, and remove that `propput` line from our IDL file (as well as remove `SetString` from *IExample2.h*'s VTable definition).

An updated example C application

Because we changed the data types passed to `SetString` and `GetString`, we must update our C/C++ clients to pass the correct data types. This is fairly trivial, and simply involves using `SysAllocString` where we need to get a `BSTR`, or `SysFreeString` where we need to free one. The updated C example is in the directory *IExampleApp2*.

A COM component in an EXE

Instead of packaging our COM component into a DLL, we could instead package it into its own EXE. The advantage of this is that the EXE runs in its own process space, and therefore, any crash of the application doesn't crash the COM component (or vice versa, necessarily). The disadvantage is that the OS must do "data marshalling" to allow data to be passed between process boundaries, so things can run more slowly.

The great news is that very little in our DLL needs to be changed. In the directory *IExampleExe*, you'll find a project to build an EXE out of the code we used to build `IExample2`. I've changed the name of the files *IExample2.c* to *IExampleExe.c*, *IExample2.h* to *IExampleExe.h*, and *IExample2.idl* to *IExampleExe.idl*. I've also searched and replaced all instances of `IExample2` with `IExampleExe`. So now, our object is referred to as `IExampleExe`. And of course, I replaced the GUIDs in *IExampleExe.h* and *IExampleExe.idl* with newly generated ones so that this EXE component won't conflict with our DLL components. Since we're creating an EXE, we no longer need a DEF file, so that has been removed from the project.

In *IExampleExe.c*, we have to replace the `DllMain` function with a `WinMain` function. Our `WinMain` must do the initialization of our `IClassFactory` and global variables, just like `DllMain` did. But our `WinMain` must also call the OLE APIs `CoInitialize` to initialize COM, and then call `CoRegisterClassObject` to add our EXE to COM's running task list. What's that? It's simply an internal list COM maintains of all the currently running EXEs which are COM components. If our EXE is not in this list, then no application can use our COM component. And, our EXE is not in that list until it calls `CoRegisterClassObject`. Since it is our `WinMain` that calls `CoRegisterClassObject`, it stands to reason that no app can use our COM component until our EXE is started.

After our EXE is added to the running task list, it simply does a message loop until it receives a `WM_QUIT` message.

While our EXE is in memory, spinning around that message loop, an application can call `CoCreateInstance` (or `CoGetClassObject`/`CreateInstance`) to get one of our `IExampleExe` objects and call its functions. To call those functions, the application need do nothing more than the IExample2App demo did. The OS handles all of the needed details to make it happen.

When our EXE is ready to terminate, it calls `CoRevokeClassObject` to remove itself from COM's running task list, and then calls `CoUninitialize` to free up the COM resources.

That's the extent of the differences (although I did add a little code in the `Release` functions of our

`IExampleExe` object and `IClassFactory` object to check if we want to unload the DLL when all apps have `Release`d our objects).

When registering a COM component in an EXE, versus a DLL, instead of creating a key named "`InprocServer32`", we create a key named "`LocalServer32`". In the directory *RegIExampleExe*, is an example of registering `IExampleExe`. (No uninstall utility has been provided. It is a trivial exercise left up to you to accomplish).

To test out this new EXE component, you can simply modify *IExample2App.c* to reference the GUIDs for `IExampleExe`, by changing it to `#include IExampleExe.h`:

```
#include "../IExampleExe/IExampleExe.h"
```

Remember that you must install/register our COM component once, and then run our component (*IExample.exe*) to get it into COM's task list **before** you run the modified *IExample2App.exe* test program.

What's next?

In the next chapter, we'll look at the `IDispatch` functions from the point of view of a script engine. How does a script engine translate some code in its own language to a call to our object's `IDispatch` functions?

License

About the Author

Jeff Glatt

Location: United States

Discussions and Feedback

49 messages have been posted for this article. Visit **http://www.codeproject.com/KB/COM/com_in_c2.aspx** to post and view comments on this article, or click **here** to get a print view with messages.