Inspect Dynamic Objects

Inspect and manipulate dynamic objects efficiently using the TypeLib Information object library.

by Matthew Curland

WHAT YOU NEED
Visual Basic 6

It is not uncommon to encounter objects you didn't have a typelib definition for when you were coding against them. These objects, which I'll refer to as dynamic objects, present a problem. You can't call members of these objects using the VB. (dot) operator because this requires that VB know the name of the method when the code is compiled. In this article, I'll show you how to call members that are unknown at compile time, how to call them in the most efficient fashion, and how to ask an object what its properties and methods are (see Figure 1). Note that this article relies heavily on the Invoke* functions in the TypeLib Information (TLI) object library (TypeLib Information is in the References dialog; tlbinf32.dll is on disk), which ships with VB5 and VB6. I'll use some functionality available only in the VB6 DLL.

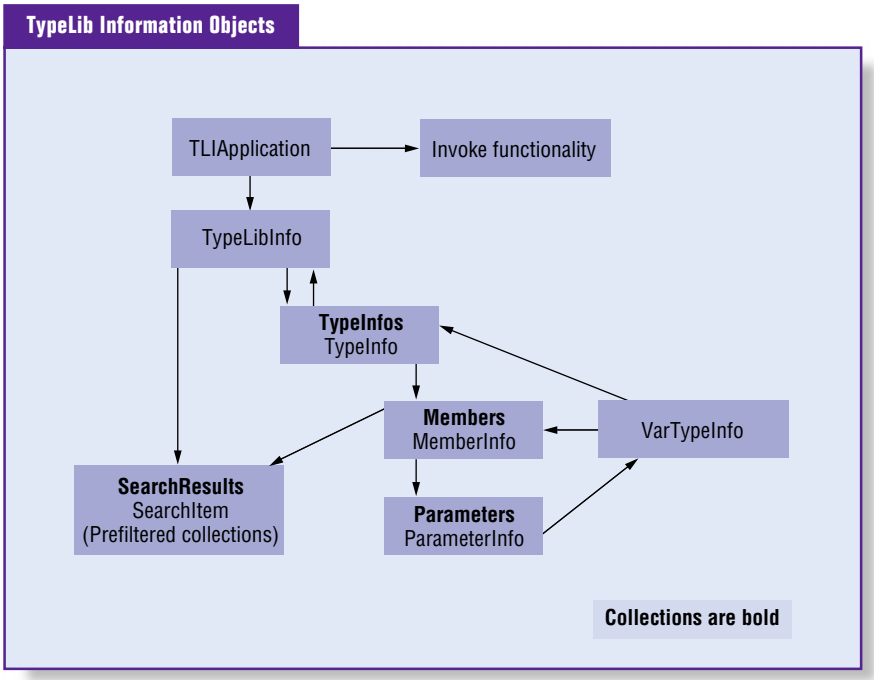

Figure 1 Sometimes you encounter objects you didn't have a typelib definition for when you were coding against them. This figure shows the object model for the TypeLib Information (TLI) object library, which gives you the ability to call members that are unknown at compile time. This article shows you how to call these members in the most efficient fashion and how to ask an object what its properties and methods are. Besides calling dynamic objects, you can use TLI to create a VB-style Object Browser or extract lower-level information from any type library.

Calling methods on an object dynamically requires that the object support the IDispatch interface. Retrieving a reference to an IDispatch interface is the same as assigning a reference to a variable declared "As Object." I strongly discourage the use of As Object or As Variant variables to hold object references for general VB programming—this practice prevents the compiler from optimizing your code for the most efficient function calls against your objects. However, if you don't know the members of an object at compile time, you also don't know the type of the object (and vice versa). You must store your mystery object in an As Object variable because setting it to any other type is likely to fail.

You make a late-bound call in VB by assigning a reference to an As Object variable and using a dot to call the method of a name specified at compile time. This forces a runtime call to IDispatch::GetIDsOfNames to translate the name into a numeric ID, and then a call to IDispatch::Invoke with the numeric ID. IDispatch::Invoke then calls the correct method on the object.

You can use the TLI.InvokeHook function to duplicate this functionality

without VB's dot. TLI.InvokeHook takes the object, member name, or ID; an InvokeKind; and a list of parameters (passed in reverse order, as with IDispatch::Invoke itself), and calls GetIDsOfNames and Invoke. This accomplishes essentially the same task as the native VB dot operator making a late-bound call.

InvokeKind is a required parameter of InvokeHook, so it's important to understand what it is. There are four InvokeKinds: INVOKE_FUNC, INVOKE_PROPERTYGET, INVOKE_PROPERTYPUT, and INVOKE_PROPERTYPUTREF. These values correspond directly to VB's Function/Sub, Property Get, Property Let, and Property Set procedures. To assign a value to a property, you call *obj.prop = value*, which calls a Property Let function if the object is implemented in a VB class. The corresponding InvokeHook call is *InvokeHook obj, "prop," INVOKE_PROPERTYPUT, value*.

You can also specify multiple InvokeKinds by Or'ing the values together. In fact, *INVOKE_FUNC Or INVOKE_PROPERTYGET* is a commonly used value because late-bound calls to functions and Property Get procedures are indistinguishable based solely on the syntax of the calling code. You simply pass both values to cover both possibilities.

So far, the functionality I've described in TLI.InvokeHook is also available in VB6's new CallByName function, except the parameters are passed forward to CallByName. However, you can't duplicate the functionality described in the rest of this article with the CallByName function.

Implement Early ID Calling

InvokeHook can take either a name or a numeric ID as the second parameter. If a name is passed, the name must first be converted through a call to IDispatch::GetIDsOfNames into a numeric ID before the call to IDispatch::Invoke. If you know the ID, you can optimize your code by passing the ID instead of the name. This is particularly important if your object resides in a separate process or runs over DCOM because you can essentially double the speed of your calling code. If you make more than one call to a given function, use InvokeHook with an ID instead of a name.

You can get an ID in three different ways. First, you can get it from a predefined list of known IDs. For example, zero is the default property or method. This list corresponds to the list of values available in the Tools | Procedure Attributes | Properties | Advanced | Procedure ID setting in VB or the DISPID_* constants in the oaidl.h and olectl.h files. You can also get the ID using the TLI.InvokeID method, which takes two parameters (obj and name), calls IDispatch::GetIDsOfNames, and returns the member ID. Finally, you can get a member ID using object inspection, which I'll discuss later.

Because you use InvokeHook to call functions unknown at compile time, you must be able to call a function with an arbitrary number of parameters. However, the last parameter in InvokeHook (and CallByName) is a ParamArray, which requires you to know the number of parameters at compile time. TLI also features an InvokeHookArray function that enables you to work around this problem. This function takes an array of Variants rather than a ParamArray as the last parameter. Both functions run exactly the same code internally.

The main disadvantage of the InvokeHookArray function: You lose implicit support for passing ByRef arguments. If you use InvokeHook, VB takes care of all ByRef arguments for you. For example, if you pass a String variable to a ByRef String parameter in InvokeHook, your variable will be updated. When you assign the string to an array, however, you lose the link with your variable. Get around this limitation by using CopyMemory and VarPtr to fill your array with Variants that have the VT_BYREF bit set, duplicating the work that VB does for you with a ParamArray call. VT_BYREF and other VT_* constants are defined for you in TLI. In general, you Or together VT_BYREF with the VarType of the variable. However, if the variable is an object type, use VT_DISPATCH directly because VarType gives the type of the object's default member. This snippet shows how to pass a Long argument ByRef:

```
Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" _
   (pDest As Any, pSrc As Any, _
   ByVal ByteLen As Long)
Dim Count As LongDim Args() As
Variant
ReDim Args(0)
Args(0) = VarPtr(Count)
'VarType(Long) = VT_I4
CopyMemory Args(0), CInt(VT_I4 Or _
   VT_BYREF), 2
```

RESOURCES

For help on TypeLib Information, download TlbInf32.chm from http://msdn.microsoft.com/vbasic/download.

You need to consider two additional cases if you call a method that has optional parameters. If the optional parameter lies at the end of the parameter list, simply omit it. If you want to skip an optional parameter, however, pass a Missing value in the appropriate slot in your array of Variants. You can retrieve this value by calling the GetMissing function with no parameters. Although VB's IsMissing function applies only to Variant type optional parameters (typed optional parameters are never officially Missing), you should pass the Missing value to skip both Variant and typed optional parameters:

```
Function GetMissing(Optional Var As Variant)
   GetMissing = Var
End Function
```

Inspect Objects at Run Time

The only reason for using InvokeHook is that you don't know the names of the members you need to call at compile time. If you knew the member name, you'd simply compile it into your code instead of calling it through a dynamic object. This means your program must get the member names dynamically from some other source, such as a file or rows in a database. However, any mechanism you use to get the names simply retrieves information almost always available from the object itself anyway. This information is always available from VB-created objects. In other words, the most direct mechanism is simply to ask the object for its list of properties and methods and do away with other sources of information, which can easily get out of sync with the object's current implementation.

OLE Automation provides two standard mechanisms for retrieving type information from running objects: IDispatch::GetTypeInfo and IProvideClassInfo::GetClassInfo. The first gets information about the running object's currently referenced interface; the second retrieves information about the class itself. In this context, a class is a type that can appear after a New keyword in VB or that you can instantiate by passing a ProgID to CreateObject or GetObject. Although a class can implement multiple interfaces, you generally deal with an object's default interface in late-bound scenarios. The information retrieved through GetTypeInfo is equivalent to the type information about the default interface of the class retrieved with GetClassInfo.

TLI supports querying a running object for both pieces of information. GetClassInfo corresponds to TLI.ClassInfoFromObject, and GetTypeInfo corresponds to TLI.InterfaceInfoFromObject. Although the method of retrieving this information is standard, not all objects support these calls, so error trapping is essential. For example, a VB-created class with instancing set to PublicNotCreatable always supports InterfaceInfoFromObject, but supports ClassInfoFromObject only if you implement additional interfaces or if the class has events. Note that VB intrinsic objects or private classes support these calls in the IDE, but not in a compiled executable file.

TLI objects are VB-accessible wrappers on typelib information that include routines to generate useful collections and VB-friendly information from the underlying typelib data. ClassInfoFromObject and InterfaceInfoFromObject both return a TypeInfo object, which is the description of a single type in a type library. A TypeInfo object's Members collection is an unfiltered collection of the object's properties and methods. All the restricted IUnknown and IDispatch methods are included in the collection, and you get an item in the collection for each InvokeKind supported by a property (GET, PUT, PUTREF).

Calling the GetFilteredMembers method off the Members collection reduces this data to a set you can browse easily (download Listing 1 from the free, Registered Level of The Development Exchange; see the Code Online box for details). VB6 allows you to pass user-defined types in public methods and properties of public classes. It also lets you store and pass structures in Variants that are defined publicly in public classes. TLI contains the TypeInfoFromRecordVariant method and RecordField property, which let TLI support runtime examination of such a stored structure in a Variant using the same objects that it uses with class instances. Because UDTs don't contain functions, RecordField is significantly easier to use than InvokeHook—you can determine the InvokeKind syntactically, without having to disambiguate method and Property Get calls (download Listing 2 from DevX).

TypeInfo objects obtained from objects that have publicly accessible type libraries are rarely more than pointers to an on-disk typelib. The typelib is generally registered when the component is installed on the local machine, so you can usually retrieve a reference to the typelib local to the calling thread. You will often perform object inspections against objects running in a different thread or process, or over DCOM. You get much better performance with object inspection if you load the library locally. If you make extensive use of the TypeInfo retrieved from an object running in a different process, all internal TLI calls are also made cross-process. You can use the TypeLibInfoFromRegistry method to create your local TypeLibInfo object if you use the TypeInfo object's Parent and TypeInfoNumber properties. This allows you to use TypeLibInfo.TypeInfos.IndexedItem to retrieve the equivalent TypeInfo object:

```
Function BestClassInfo(ByVal Object As _
   Object) As TypeInfo
Set BestClassInfo = TLI.ClassInfoFromObject(Object)
On Error GoTo NotAvailable
With BestClassInfo.Parent
With TLI.TypeLibInfoFromRegistry _
   (.GUID, .MajorVersion, .MinorVersion, .LCID)
Set BestClassInfo = .Me.TypeInfos.IndexedItem( _
   BestClassInfo.TypeInfoNumber)
End With
End With
Exit Function
NotAvailable:
Err.Clear
End Function
```

The new CallByName functionality provides a partial mechanism for manipulating dynamic objects, but you need a more complete solution to inspect these objects and call them with optimal efficiency. The TypeLib Information objects let you inspect your objects and call them in a flexible and performance-aware fashion. For help on TypeLib Information, download TlbInf32.chm from http://msdn.microsoft.com/vbasic/download. VBPJ

About the Author

Matthew Curland is a developer on Microsoft's Visual Basic team. He worked on VB's IntelliSense features and the VS 6.0 Object Browser (which uses TypeLib Information objects for type library browsing), and is currently working for the VB wizards team. Reach him at MattCur@microsoft.com.

CODE ONLINE

You can find all the code published in this issue of **VBPJ** on The Development Exchange (DevX) at http://www.vbpj.com. For details, please see "Get Extra Code in DevX's Premier Club" in Letters to the Editor.

Inspect Dynamic Objects
Locator+ Codes
Listings for the entire issue: VBPJ1198
Listings for this article only (subscriber Premier Level): BB1198