

NAVIGATING THE LANDSCAPE OF IVI

**Kirk Fertitta
Pacific MindWorks, Inc.
5665 Oberlin Drive, Suite 202
San Diego, CA 92121
(858) 587-8876 x237
kirk@pacificmindworks.com**

Abstract - Since its inception in August of 1998, the IVI Foundation has continued to expand and evolve software standards for instrument drivers. The adoption of IVI has increased markedly during that period, particularly over the past two years, with a number of instrument vendors announcing a growing inventory of instruments shipping with IVI drivers. Additionally, end user companies as well as U.S. government agencies have helped create an expectation that newly developed instruments will be made available with IVI drivers.

With this increased visibility and availability of IVI technology has come an increased confusion over the different types of IVI drivers available, especially when trying to determine which technology is most suitable for a particular application domain or development environment. Specifically, most organizations do not fully understand the important differences between IVI-COM and IVI-C drivers. With more driver standards on the way from the IVI Foundation and with the increased involvement of the newly formed LXI Consortium, a clear understanding of the full IVI technology landscape is crucial.

This paper will begin with a comparison of IVI-COM and IVI-C technologies, from both a driver developer and an end user perspective. It will go on to demonstrate new technology that allows both IVI-COM and IVI-C drivers to be developed and deployed together in a single component. This technology offers compelling advantages in terms of development time and cost, as well as in long term driver maintenance, while simultaneously delivering to end users the advantages of both IVI-COM and IVI-C drivers.

UNDERSTANDING IVI-COM AND IVI-C DRIVERS

Over the years, a great many debate have raged over which of the current IVI driver technologies (IVI-COM or IVI-C) offer the best overall end-user experience. While this paper does not attempt to offer a final resolution to these debates, it is crucial that companies embracing the IVI standards fully understand the benefits of each technology and where they offer the most value. Indeed, it is the author's belief that a single driver that presents both IVI-COM and IVI-C interfaces offers the best solution for both the driver developer and the end user.

The following sections offer a comparison of IVI-COM and IVI-C technologies. The goal is to understand the issues at stake and to explain why we believe the best approach for supporting IVI is using a native IVI-COM driver augmented with an IVI-C wrapper.

Ease of Use in Popular Development Environments

Microsoft Visual Studio

Undoubtedly, IVI-COM drivers offer the best end-user experience in Microsoft environments. Many features of COM itself were designed with Visual Basic users in mind. COM type libraries present the Visual Studio IDE (integrated development environment) with a great deal of descriptive information (metadata) for supporting such features as object browsers, IntelliSense, and context-sensitive help. Visual Basic users are almost completely insulated from the complexities

of COM and C++, while Visual C++ users enjoy the power and flexibility of direct COM programming with a variety of convenience features, such as smart pointer wrapper classes.

The value of type library metadata becomes most evident when using IVI drivers from .NET applications. Visual Studio ships with a command line utility called `tlbimp`. This utility feeds off of COM type libraries to generate .NET wrappers known as *interop assemblies*. These interop assemblies present a .NET interface to COM components, so that .NET applications can access the COM component as if it had been originally authored in .NET.

The overall experience for .NET users accessing IVI-COM drivers is quite good. Moreover, Visual Studio support for automatically generating or accessing interop assemblies is excellent. Interop assemblies can be deployed in a shared location (known as the Global Assembly Cache – or GAC), so that all .NET applications accessing an IVI-COM driver receive identical views of the component. Versioning and side-by-side deployment of interop assemblies is also supported. This is all crucial for the overall IVI user experience, and absolutely mandatory if any modicum of interchangeability is to be preserved.

Moreover, the IVI Foundation has capitalized on these built-in features by defining a very simple specification to ensure consistency amongst IVI-COM driver interop assemblies – *IVI-3.14: Primary Interop Assembly*. The fact that the body of this specification requires barely eight pages to completely codify interop assembly creation and deployment gives testimony to the ease with which IVI-COM drivers can be integrated into .NET applications.

By contrast, IVI-C drivers are simply conventional Win32 DLLs. As such, they do not contain enough metadata to enable the .NET Framework or Visual Studio to integrate them as easily as IVI-COM drivers. The metadata for IVI-C drivers is actually dispersed among several files – the header file, the function panel file (.fp), the attribute information file (.sub), and the DLL itself. None of these can be consumed by Visual Studio to produce something akin to an interop assembly. Rather, each client application must include the function signature along with the requisite .NET interop attributes for every IVI-C driver method it wishes to use. Constructing such function signatures directly within the client

application is extraordinarily tedious and error-prone – and every client must repeat these declarations. While tools for automatically generating a .NET wrapper for IVI-C drivers may be (and likely have been) written, no standard exists to govern how they are created. This invariably leads to a loss of interchangeability as well as the inconsistent user experience IVI has sought to eliminate in the first place.

ANSI-C Environments

As convenient and powerful as IVI-COM driver support is in Microsoft IDEs, it is equally unpalatable in ANSI-C environments. While IVI-COM drivers contain type libraries with all of the requisite metadata for instantiating and using the driver, none of the type library information is typically consumable by ANSI-C environments. Rather, the end user must manually `#include` header files (.h) and interface definition files (.i.c) emitted by the Microsoft IDL compiler (MIDL). These MIDL-generated files are not really intended for end-user consumption and are actually re-generated each time the driver source code is compiled. In fact, the MIDL compiler does not even generate in these files the proper data structures for a C client to access the IVI-COM driver. Rather, the header files emitted by MIDL must be manually “massaged” to provide a working interface for C users; and since the header file is re-generated each time the driver is compiled, this “touch-up” process must be repeated each time the driver is compiled.

End users in ANSI-C environments must also contend directly with COM data types, such as VARIANTS, SAFEARRAYs, and BSTRs. The convenient C++ wrappers available to Visual C++ users are simply not available to users working in straight C. As an example, consider the code required to create a SAFEARRAY and populate it with a few data values.

```
SAFEARRAY* psa = NULL;
SAFEARRAYBOUND rgsabound[1];
double *pData = NULL;

/* Create an array of 3 double elements */
rgsabound[0].lBound = 0;
rgsabound[0].cElements = 3;
psa = ::SafeArrayCreate(VT_R8, 1, rgsabound);
if (psa == NULL)
    return E_OUTOFMEMORY;
```

```

/* Populate the array with data */
::SafeArrayAccessData(psa, (void HUGE_P **
&pData);
for (int i = 0; i < 3; i++)
{
    pData[i] = i * 3.14;
}

```

Receiving a SAFEARRAY of data returned from an IVI-COM driver function call can be even more tedious than the code shown above. By comparison, a user working in C++ with access to the CComSafeArray helper class has a much easier time than the C user. The same task of allocating and initializing an array of doubles can be accomplished with the following simple code in Visual C++:

```

CComSafeArray<double> sa(3);
sa.SetAt(0, 1.5);
sa.SetAt(0, 2.5);
sa.SetAt(0, 3.5);

```

Traversing the hierarchy of an IVI-COM driver is also quite inconvenient in C-based environments. In Visual C++, a property that resides on a lower level interface can be accessed very naturally using smart pointer wrapper classes. Consider the code below that enables FM modulation on a signal generator driver:

```

IIVI_RFSigGenPtr spSigGen(CLSID_MySigGen);

spSigGen->AnalogModulation->FM->Enabled =
    VARIANT_TRUE;

```

The corresponding code an ANSI-C user would have to write to accomplish the same task is somewhat daunting, especially considering that setting a property like the one shown above is done repeatedly in virtually all IVI-COM driver applications. Depending upon exactly how the driver developer fixed the bugs in the MIDL-generated header file for their driver, the client code to access the Enabled property would look something like the following:

```

HRESULT hr;
IIVI_RFSigGen* pSigGen = NULL;

/* Instantiate the driver */
hr = ::CoCreateInstance(&CLSID_MySigGen,
    NULL, CLSCTX_ALL, &IID_IIVI_RFSigGen,
    (void**)&pSigGen);

```

```

if (FAILED(hr)) return hr;

```

```

/* Access the AnalogModulation sub-interface */
IIVI_RFSigGenAnalogModulation* pMod = NULL;
hr = IIVI_RFSigGen_get_AnalogModulation(
    pSigGen, &pMod);
if (FAILED(hr)) return hr;

```

```

/* Access the FM sub-interface */
IIVI_RFSigGenFM* pFM = NULL;
hr = IIVI_RFSigGenAnalogModulation_get_FM(
    pSigGen, &pFM);
if (FAILED(hr)) return hr;

```

```

/* Set the Enabled property on the FM interface */
hr = IIVI_RFSigGenFM_put_Enabled(
    pSigGen, VARIANT_TRUE);

```

```

/* Release the interface pointers */
IIVI_RFSigGen_Release(pSigGen);
IIVI_RFSigGen_Release(pMod);
IIVI_RFSigGen_Release(pFM);

```

Needless to say, the code above makes IVI-COM driver usage in ANSI-C environments most uninviting, particularly considering that the same task can be accomplished with a single line of code in either Visual C++, C#, or Visual Basic. Had the Enabled property been located on a repeated capability, such as a “Channel” or a “Trace”, then the code above would be uglier still.

Not surprisingly, IVI-C drivers provide a much cleaner end-user experience in ANSI-C environments. Indeed, the IVI-C architecture was designed with the C user in mind. Since IVI-C drivers rely upon C-style data types (e.g. C-style strings, C-style arrays), nothing resembling the cumbersome SAFEARRAY or BSTR data manipulations is necessary.

Like IVI-COM driver methods and properties, IVI-C driver functions and attributes are organized in hierarchies. (Note the use of the terms “methods” and “properties” when referring to IVI-COM drivers and the use of the terms “functions” and “attributes” when referring to IVI-C drivers. These logically refer to the same thing, but the distinct terminology is consistent with the IVI specifications, and so will be used by this author). The hierarchical organization of functions in an IVI-C driver is referred to as the *function tree*, and it is expressed in the function panel file (.fp). Attributes in IVI-C drivers are organized in hierarchies described separately in the *attribute*

information file (.sub). However, this hierarchy is “flattened” when exposing the driver functions and attributes from the driver DLL. This means that ANSI-C users of IVI-C drivers will not have to endure the coding gymnastics that are required to traverse the hierarchy of IVI-COM drivers in C environments. Compare the following code for setting the Enabled attribute of an IVI-C driver with that shown previously for an IVI-COM driver.

```
/* Definition of viStatus and vi session omitted */  
viStatus = IviRFSigGen_SetAttributeViReal64(  
    vi, "", IVIRFSIGGEN_ATTR_FM_ENABLED,  
    true);
```

Even though the FM Enabled attribute is buried a couple of levels deep in the attribute hierarchy (RF -> FM -> Enabled, to be specific), only a single function call is required to access the attribute. Note that the second parameter to the function call above is the repeated capability selector, and its value is set to an empty string. Had the Enabled attribute been located on a repeated capability, then only this parameter would need to be changed, whereas additional function calls are required when accessing IVI-COM repeated capabilities from ANSI-C environments.

LabWindows/CVI

Not coincidentally, IVI-C drivers work particularly well in National Instruments LabWindows/CVI. LabWindows understands how to parse both the function tree hierarchy in the .fp file and the attribute hierarchy in the .sub file. The extra metadata in these files allows LabWindows to present nice graphical displays of both sets of hierarchies. LabWindows/CVI includes graphical function panels that can be used to operate each IVI-C driver function and attribute individually. This is an extremely convenient tool for both driver developers and end users. Driver developers can use the function panels as a unit testing tool – a way to individually validate each function and attribute before shipping the driver. End users can quickly begin performing simple operations on their instrument using the driver through these function panels.

SUPPORTING IVI-COM AND IVI-C WITH A SINGLE DRIVER

From the previous discussion, it should be clear that neither IVI-COM nor IVI-C provides an ideal solution for all IDEs. ANSI-C and LabWindows/CVI users will not have a pleasant experience building applications with an IVI-COM driver, nor will C# or Visual Basic users be as productive using an IVI-C driver. This seemingly leaves the instrument vendor with a difficult choice – either alienate some portion of the end-user community or implement, deploy, and maintain both an IVI-COM and an IVI-C driver.

Understanding IVI Driver Wrappers

Developing and maintaining a single IVI driver can be a formidable enough proposition, not to mention the prospect of needing to develop two IVI drivers for each instrument. Building and maintaining separate code bases for IVI-COM and IVI-C drivers would lead to a great deal of duplicate effort and a variety of maintenance headaches. Two separate IDEs would be needed – LabWindows/CVI for the IVI-C driver and Visual Studio for the IVI-COM driver. This would likely require separate individuals to develop and maintain the driver, as many developers skilled enough to develop an IVI driver in one of those environments likely lacks the requisite expertise to develop the other type of IVI driver in the other environment. Help documentation would have to be developed and maintained separately, even though the actual help content for each driver method and property is virtually identical between the IVI-COM and IVI-C drivers. Separate installation programs would also have to be developed and maintained, and each would have to ensure compliance with the IVI driver installer requirements laid out in *IVI-3.1: Driver Architecture Specification*.

All of the issues above point to a need for a single driver code base that can satisfy both IVI-COM and IVI-C driver requirements. This can be accomplished by using an IVI driver wrapper. The IVI Foundation foresaw the possibility that driver vendors might want to create IVI driver wrappers, and so they are discussed specifically in the IVI specifications.

Two types of wrappers are allowed by IVI: IVI-COM wrappers on native IVI-C drivers and IVI-C wrappers on native IVI-COM drivers. Both techniques use a single source code base to implement the driver logic, and then add a thin veneer to expose the alternate driver interface. Choosing which technique to use in a real-world driver involves a number of factors.

Choosing an IVI Wrapper Architecture

While building an IVI-COM wrapper on top of an IVI-C native driver is certainly viable, there are a variety of reasons why we have chosen to implement a solution based on IVI-C wrappers on top of native IVI-COM drivers. This section will present some of the motivations for choosing this approach.

Broadest Client Environment Support

As the previous sections explained, IVI-COM drivers provide the best end-user experience in virtually all popular client environments, with LabWindows/CVI being the notable exception. Even in such ANSI-C environments, using an IVI-COM driver can be reasonably accomplished, albeit considerably less convenient. IVI-C drivers, on the other hand, simply do not provide a reasonable experience for .NET users, such as C# and VB.NET. The interop mechanism for calling functions in an IVI-C driver from .NET applications is so tedious as to be unusable. That being the case, driver vendors should consider an IVI-COM driver to be required for broad-based application support.

Single Development Environment

LabWindows/CVI is currently the only viable choice for native IVI-C driver development. Similarly, Microsoft Visual C++ is the only viable choice for native IVI-COM driver development. With either an IVI-C wrapper or an IVI-COM wrapper approach, considerable automatic code generation will be required to make the entire process practical. Naturally, this automatic code generation would take place in the chosen development environment. Implementing COM interfaces is much more naturally done in C++ than any other language, including straight C. A code generator based in an ANSI-C environment, such as LabWindows/CVI, would struggle to produce reasonable looking code for an IVI-COM

driver. COM developers would, for example, expect such code to be in C++, and obviously ANSI-C environments can not compile C++ code. While one might argue that the nature of auto-generated code is unimportant, the sections below will explain why a wrapper approach must take care to produce code that the driver developer can reasonably follow.

By contrast, a wrapper generator based in Visual C++ has little difficulty generating IVI-C implementation code that closely mimics a hand-written implementation. This stems from the simple fact that IVI-C drivers are structurally far simpler than IVI-COM drivers. All of this leads to the realization that an IVI-C wrapper approach allows all of the development to take place in a single environment – in this case, Microsoft Visual C++.

Single Component Architecture

It follows somewhat from the single-IDE argument above, that an IVI-COM wrapper approach would necessarily eliminate (or at least make very difficult) the prospect of bundling both the IVI-COM and IVI-C drivers into a single component DLL. With the IVI-COM wrapper approach, the IVI-C driver would be built in LabWindows/CVI into one DLL while the auto-generated wrapper code would likely need to be built in a separate stage using Visual C++ and producing a second DLL.

With an IVI-C wrapper approach based in Visual Studio, the native IVI-COM driver code and the IVI-C wrapper code can be generated and compiled together in a single DLL. This simplifies deployment and versioning of both IVI-COM and IVI-C interfaces.

Customizing the Wrapper Implementation

While both the IVI-COM and IVI-C wrapper approaches rely heavily on auto-generated code, both approaches must also provide a mechanism for the driver developer to customize the generated code in some fashion. For instance, the IviRFSigGen specification contains no fewer than forty-one functions that exist in the IVI-C hierarchy but *not* in the IVI-COM hierarchy. These are termed *IVI-C-only functions*. Correspondingly, the IVI specifications contain a

number of *IVI-COM-only* methods and properties. This means that the native driver (with either approach) would have no implementation to “wrap”. Thus, the driver developer would be required to manually implement these functions. Alternatively, the driver developer may simply wish to override the auto-generated wrapper implementation. In any event, the critical point is that an IVI-COM wrapper approach would require the developer to work with straight C code for implementing COM functions, which, as mentioned previously, would be inconvenient at best.

An IVI-C wrapper approach, however, would provide the full power of the C++ language to implement or customize IVI-C wrapper functions. Moreover, the developer could leverage numerous convenience classes, such as smart pointers, CString, CComSafeArray, CComBSTR, and a variety of data type converters.

Supporting Multiple IVI Instrument Classes

IVI-COM drivers have the unique ability to expose more than one IVI class-compliant interface from a single driver. For instance, a switch measurement unit might expose the *IviDmm* interface for controlling the scanning DMM portion of the instrument, while also exposing the *IviSwch* interface for managing the switches. By contrast, IVI-C drivers can only support a single IVI instrument class. This is due in part to the fact that IVI-C attribute name and value definitions are not unique across instrument classes.

With an IVI-COM wrapper approach, it would not be possible to build an IVI-COM driver that supported multiple class-compliant interfaces, since the underlying native IVI-C driver would have no way to support those functions. However, an IVI-C wrapper approach could still allow the underlying IVI-COM driver to support multiple class-compliant interfaces. The wrapper generator would simply be instructed as to which of the class interfaces should be exposed in the IVI-C wrapper.

Function Hierarchy

IVI-C drivers possess an inherent “mismatch” between the function hierarchy and the attribute

hierarchy. Recall that the function hierarchy is maintained in the .fp file, while the attribute hierarchy is expressed in the .sub file. The difficulty is that the .sub file specification only allows two levels of depth in the attribute hierarchy, while the .fp file specification allows any number of levels in the function hierarchy. This often forces IVI-C methods to be organized in different locations than attributes with which they may be closely related.

IVI-COM hierarchies have no such limitation on depth; consequently, IVI-COM drivers often have deeper hierarchies. However, an IVI-COM wrapper generator would have no way to infer a hierarchy deeper than two levels, since the underlying IVI-C hierarchy of the native driver would have that restriction. By contrast, an IVI-C wrapper generator could easily “flatten” an IVI-COM hierarchy that was too deep and produce a well-organized IVI-C hierarchy in the wrapper.

Repeated Capabilities

Instruments often possess multiple instances of the same type of functionality, such as multiple channels on an oscilloscope or multiple traces in a spectrum analyzer. The IVI specifications refer to these as *repeated capabilities*, and supporting them is an integral part of both IVI-COM and IVI-C driver development. Repeated capabilities are easy to spot in an IVI-COM driver hierarchy, as they typically exist on a special repeated capability collection interface and they are often implemented on distinct COM classes. This makes it very easy for an IVI-C wrapper generator to determine from a native IVI-COM driver which IVI-C wrapper methods apply to a repeated capability.

Looking at the functions exposed from an IVI-C driver, however, reveals nothing about whether a particular function or attribute applies to a repeated capability. Every IVI-C function includes a *repeated capability selector* parameter, whether that function applies to a repeated capability or not. Thus, an automated IVI-COM wrapper generator would have no way to determine how to generate the correct repeated capabilities on the IVI-COM wrapper. The wrapper generator would have to gather additional input from the user somehow, asking (for each function and attribute) if the item belongs to a repeated capability. Add to this the need to account for nested repeated

capabilities, and the IVI-COM wrapper approach quickly becomes unattractive.

USING NIMBUS TO BUILD IVI-COM AND IVI-C DRIVERS

With a solid understanding of the benefits of building IVI-COM drivers with IVI-C wrappers, we now present a solution that addresses the technical challenges in making the development of such unique drivers practical. Our approach relies upon the Nimbus Development Suite from Pacific MindWorks. This product integrates directly with Visual Studio, so that the driver developer has full access to the Visual C++ tools required for effective IVI-COM driver development.

Intercepting the Visual Studio Build Process

One of the most compelling reasons to use Visual Studio as the primary development environment is the degree of IDE customization available. Virtually every aspect of the Visual Studio IDE can be accessed programmatically. Much of the IVI-C wrapper technology relies upon the ability to customize the Visual Studio build process. Specifically, we intercept the beginning of each IVI-COM driver build so that we can generate IVI-C wrapper code.

When a user selects the “Build” command to compile their IVI-COM driver, the Nimbus code generator intercepts the build command and performs a variety of *pre-build* steps. Recall that one of the goals is to produce both IVI-COM and IVI-C drivers in a single DLL. This requires the IVI-C wrapper implementation code to be generated *before* the main IVI-COM driver project build begins.

When the Visual Studio build process begins, the IDE fires an event that Nimbus traps. It is at this point that the Nimbus project file, which summarizes all of the IVI-COM driver design information, is inspected. Both the IVI-C function and attribute hierarchy are inferred from the natural hierarchy of the IVI-COM interfaces. This allows the .fp and .sub files to be generated. From the .fp and .sub files, Nimbus proceeds to

automatically generate the IVI-C header file (.h) and the implementation file (.cpp).

Generating each of the IVI-C driver files during this pre-build step offers two major advantages – automatic code round-tripping, and easy user customization of the auto-generated IVI-C files. Round-tripping of the driver refers to the ability to automatically update the driver source code in response to design changes made by the developer. For efficient IVI-COM development, this is crucial, since developers typically add, remove, rename, and otherwise modify methods and properties definitions throughout the driver development cycle. Changing even a single method definition requires tedious changes to several source code files – an extraordinarily time-consuming and error prone process. With an IVI-C wrapper in the picture, design changes are even more difficult to contend with, since the required source code modifications must now be made in two sets of source code files. Automatically re-generating the files during pre-build ensures that all of the source files for both IVI-C and IVI-COM drivers are in sync with the latest driver design information, with no involvement required from the developer.

The pre-build step also provides a means for the driver developer to customize how the .fp, .sub, .h, and .cpp IVI-C driver files are created. The sections below will provide details on how the user can selectively customize the generation of each of these files.

Customizing the IVI-C Hierarchy

As previously explained, the Nimbus code generator can infer the hierarchy of IVI-C functions and properties by examining the natural hierarchy of the underlying IVI-COM driver. However, it may be the case that the driver developer wishes to customize or even completely re-arrange the IVI-C hierarchy based on driver-specific needs. For instance, in the case of IVI-COM properties, it may not be possible to export corresponding IVI-C attributes in the desired location since the IVI-C attribute hierarchy is limited to a depth of two levels. Thus, the auto-generated placement of these attributes may not be ideal, so the developer may wish to reorganize them in the IVI-C hierarchy. Alternatively, it may simply be the case that the developer does not wish to expose certain IVI-COM methods or properties in the IVI-C wrapper at all.

The Nimbus Designer is shown below in Figure 1. The left-hand portion of the editor displays the Project Explorer view of the IVI-COM interfaces, methods, and properties. The IVI-C Editor in the middle shows the IVI-C export information for a single method or property. In the example below, the IVI-COM Configure method is highlighted in the Project Explorer and the corresponding IVI-C export information for the Configure method is shown in the editor pane. The **Export in IVI-C driver** checkbox allows the user to control whether the Configure method appears in the IVI-C driver at all. The **Location** combo box allows the user to use simple file path syntax to indicate where in the IVI-C hierarchy the Configure function should be placed. The **Node name** text box controls the name of the node where the Configure function itself appears in the IVI-C function tree. Finally, the **Function name** text

box specifies the actual Win32 function name exported from the driver. This is the function name that application programs would use. Note that the driver prefix (which is “acme1234_” in this example) is not shown in the **Function name** text box.

Each of the controls in the IVI-C Wrapper editor are initially populated with the location, node name, and function names inferred automatically from the IVI-COM hierarchy. The driver developer need only change those items that they would like to re-arrange. During the pre-build step, the information appearing here is used by the Nimbus code generator to produce the .fp and .sub files.

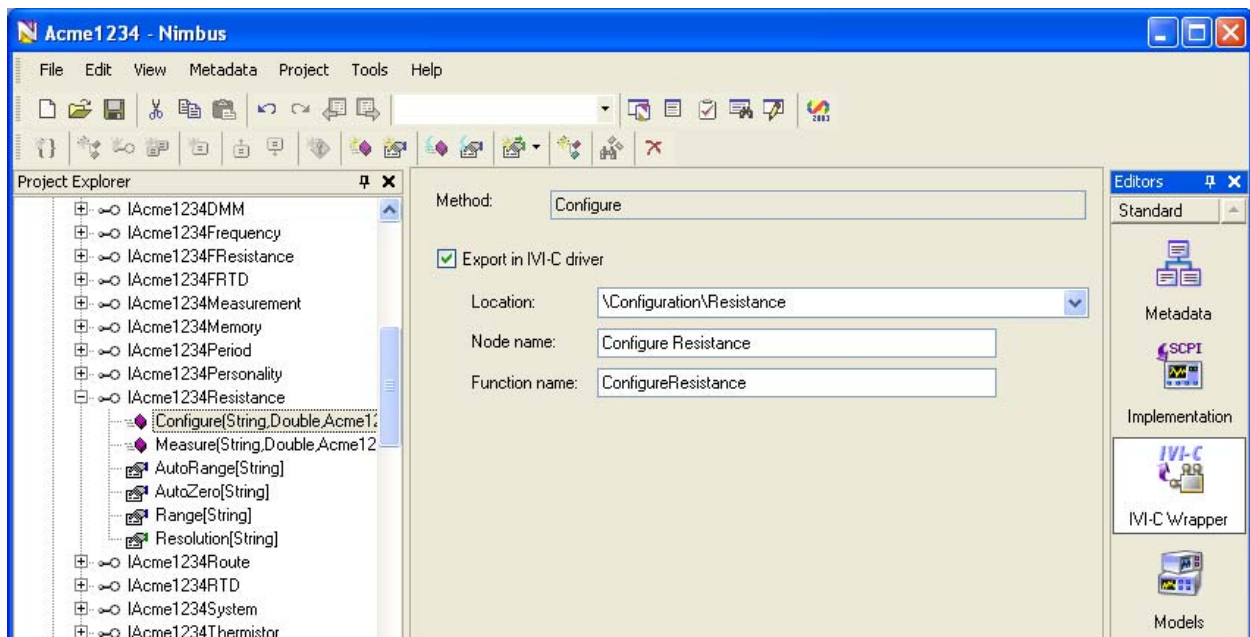


Figure 1. Customizing the IVI-C Hierarchy

Implementing the IVI-C Wrapper Functions and Attributes

During the pre-build step, the Nimbus code generator emits a .cpp file that implements each of the IVI-C wrapper functions and attributes. Only those functions and attributes with the **Export in IVI-C driver** checkbox enabled (shown above in Figure 1) are included in this file.

The chief tasks for implementing an IVI-C wrapper function are to first retrieve a reference to the underlying IVI-COM driver, which contains the actual method implementation. Then the appropriate IVI-COM interface must be retrieved. Each of the IVI-C parameters (both input and output parameters) must be converted from C-based data types (VISA data types, to be precise) to COM data types. For instance, C-style strings must be converted to BSTRs and C-style arrays must be converted to SAFEARRAYs. This task alone can be quite onerous if done manually, especially when dealing with [out] parameters, since IVI-COM [out] parameters are allocated by the callee (the driver method) and IVI-C [out] parameters are allocated by the caller (the client program).

The code sample below shows a typical auto-generated implementation of an IVI-C wrapper function. The function is the SetPath function from the IviSwtch specification and appears in the acme1234.nimbus.cpp implementation file.

```
////////////////////////////////////  
// Auto-generated implementation file  
// acme1234.nimbus.cpp  
  
ViStatus _VI_FUNC acme1234_SetPath(  
    ViSession Vi, ViConstString PathList )  
{  
    IUnknown* pDriver = NULL;  
    ViStatus status = GetDriver(Vi, &pDriver);  
    if (status == VI_SUCCESS)  
    {  
        IIVI_SwtchPath* pPath = NULL;  
        HRESULT hr =  
            pDriver->QueryInterface(&pPath);  
        status = TranslateHRESULT(Vi, hr);  
        if (SUCCEEDED(hr))  
        {  
            CBSTR _PathList(PathList);  
            hr = pPath ->SetPath(_PathList.GetVal());  
        }  
    }  
}
```

```
        status = TranslateHRESULT(Vi, hr);  
        pPath ->Release();  
    }  
    pDriver->Release();  
}  
return status;  
}
```

The implementation begins by calling the GetDriver utility function, which retrieves a generic IUnknown pointer to the underlying IVI-COM driver. The GetDriver function is re-used across each IVI-C wrapper method and attribute implementation and also provides the driver developer with convenient access to the IVI-COM driver manually implementing wrapper code (discussed shortly).

The specific IVI-COM method required is located on the IIVI_SwtchPath interface, so the code uses the QueryInterface function to retrieve a pointer to the IIVI_SwtchPath interface. If an error occurs, then the TranslateHRESULT utility function formats the COM error into a suitable IVI-C error, in accordance with the IVI specifications. The error mechanisms for IVI-COM and IVI-C drivers are quite different.

Before invoking the IIVI_SwtchPath::SetPath function on the IVI-COM driver, the code must translate the ViConstString PathList parameter to the BSTR data type. This task is performed by the CBSTR converter class. Once again, these converter classes are also an important feature for custom wrapper code development, since they encapsulate a lot of tedious and error-prone operations.

After calling through the pPath pointer into the IVI-COM driver's SetPath method, any error produced is translated to IVI-C format and both of the driver interface pointers are released.

Customizing the IVI-C Wrapper Code

For a variety of reasons, it may be the case that the auto-generated implementation of an IVI-C wrapper function or attribute is not appropriate for a specific driver. In these cases, the user can manually implement a wrapper function in a separate user-managed implementation file. During the pre-build step, Nimbus relies upon special capabilities of Visual Studio to import the custom wrapper implementation code.

One of the most powerful features of the Visual Studio IDE is a built-in mechanism for both parsing and generating C++ source code. Due to its complexity, C++ code is notoriously difficult to parse, so implementing a custom tool for performing the kinds of operations described below would not be practical. Rather, Nimbus relies upon a Visual Studio-managed in-memory object model for all of the C++ code in a driver project. The API used to access this object model is referred to as the Visual Studio Code Model. During the Nimbus pre-build step, the user implementation file is read via the Code Model interfaces and any custom wrapper code is imported and used to augment the generated IVI-C wrapper files.

Three types of customization are currently supported in the user implementation file: overriding the auto-generated wrapper implementation, implementing an IVI-defined IVI-C-only function or attribute, and adding a custom IVI-C only function or attribute. All of these use cases are supported via two special import tags placed just above the custom code – IMPLEMENT_EXPORT and ADD_EXPORT.

When the IMPLEMENT_EXPORT tag is encountered in the user implementation file, the function to which it is applied is used for the IVI-C wrapper implementation. In the way, the IMPLEMENT_EXPORT tag can be used to effectively override the auto-generated wrapper implementation of an IVI-COM method or property. Alternatively, the IMPLEMENT_EXPORT tag can be used to implement IVI-C-only functions and attributes. Recall from the previous discussion that IVI-C-only functions are those items that IVI requires to be in an IVI-C driver but that have no corresponding method in the IVI-COM driver. Thus the driver developer *must* have a convenient way to implement these manually.

The code snippet below demonstrates how one would implement the IVI-C-only function IsDebounced, which is defined in the IviSwitch instrument class specification. Note that the GetDriver, TranslateHRESULT, and data type converter classes are all available to the driver developer when authoring these functions.

```

////////////////////////////////////
// User implementation file
// acme1234.cpp

IMPLEMENT_EXPORT()
ViStatus _VI_FUNC acme1234_IsDebounced(
    ViSession vi, ViBoolean* IsDebounced)
{
    ViStatus status = VI_SUCCESS;

    // ... implement logic ....

    return status;
}

```

In order to add a custom IVI-C-only function or attribute, the driver developer simply applies the ADD_EXPORT tag to the function in the user implementation file. The ADD_EXPORT tag requires a parameter that specifies where in the IVI-C hierarchy the new function should appear. Additionally, the user may apply specially formatted comments using XML syntax. These comments are imported during the pre-build step and the inserted into the generated IVI-C help file. The syntax used in these comments is not detailed in this paper, but, in general, the format follows the well-known C# comment feature, which is well documented in the MSDN help files.

The code fragment below demonstrates how to add a custom IVI-C only function to the generated wrapper.

```

////////////////////////////////////
// User implementation file
// acme1234.cpp

/// <summary>
/// This function provides direct access to
/// the underlying I/O interface.
/// </summary>
/// <param name="Vi">
/// Instrument session.
/// </param>
/// <param name="Command">
/// Command string to be sent to the instrument.
/// </param>
ADD_EXPORT("\\System\\IO Write\\IOWrite")
ViStatus _VI_FUNC acme1234_IOWrite(
    ViSession vi, ViConstString Command)
{
    ViStatus status = VI_SUCCESS;

    // ... implement logic ...

    return status;
}

```

In the example above, the ADD_EXPORT tag specifies an export "path" of "\\System\\IO Write\\IOWrite". As a result, when this code is imported by Nimbus during the pre-build, the IOWrite function will be placed under the "System" heading in the function tree and given a node name of "IO Write". The actual function name exported from the DLL will be "acme1234_IOWrite".

CONCLUSION

This paper has presented a series of arguments as to why both IVI-COM and IVI-C drivers must be considered when instrument vendors are planning an IVI development strategy. Both types of drivers provide an excellent user experience in various development environments, but neither provides an ideal experience in all environments. Due, in no small part, to the ease of IVI-COM integration with .NET, and to the corresponding difficulty of IVI-C integration with .NET, IVI-COM drivers should be considered required for broad-based application support. Building IVI-C drivers on top of native IVI-COM drivers provides an excellent end-user experience in all IDEs and simplifies driver development and maintenance.