

: Comparing IVI-COM and IVI-C

Presented by:

Kirk Fertitta
Chief Technical Officer
Pacific MindWorks

TABLE OF CONTENTS

Table of contents 2

Introduction 4

Broadest ADE Support..... 4

.NET Support..... 5

LabVIEW Support..... 7

 IVI-C Drivers in LabVIEW7

 IVI-COM Drivers in LabVIEW.....15

Labwindows support 21

Class Drivers 22

Self Describing Components 24

Integrated Hierarchy 27

Name Collisions 30

Using Properties..... 32

The IntelliSense Experience..... 33

Type-Safe Properties..... 36

Repeated Capabilities..... 37

Multiple Instrument Class Support 41

Mixing Class-Compliant and Instrument-Specific Code 43

Error Handling..... 45

Better Data Types..... 46

Arrays and Strings..... 46

Object Parameters 49

Remote access and Location Transparency 50

Performance Myths 51

Conclusion 51

Appendix A – IVI-COM/IVI-C Comparison Chart..... 53

INTRODUCTION

The Interchangeable Virtual Instrument (IVI) Foundation¹ currently defines two interface standards for constructing IVI-compliant drivers – IVI-COM and IVI-C. While both technologies offer many of the same benefits (interchangeability, simulation, and state caching) it is crucial to understand the differences between IVI-COM and IVI-C when deciding upon which technology to adopt.

Ideally, instrument manufacturers would supply both IVI-COM and IVI-C drivers with their instruments. This provides the broadest possible “reach” for their drivers in terms of application development environments (ADEs)².

Practically, however, driver developers often are faced with choosing either IVI-COM or IVI-C as their driver platform. Time, resource, or budget constraints may preclude furnishing two sets of drivers for their instruments. In these instances, choosing IVI-COM will provide the best test-developer experience in the widest range of ADEs. This paper will examine in detail each of the advantages that IVI-COM enjoys over IVI-C.

Note: Refer to *Appendix A – IVI-COM/IVI-C Comparison Chart* for an at-a-glance comparison of the two driver technologies.

BROADEST ADE SUPPORT

IVI-COM drivers offer test programmers the best 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 descriptive information for use with object browsers, IntelliSense, and context-sensitive help. Visual Basic users are completely insulated from the complexities of COM and C++, while Visual C++ users enjoy the power and flexibility of direct COM programming, including a variety of convenience features, such as smart pointer wrapper classes.

¹ The Interchangeable Virtual Instrument (IVI) Foundation (www.ivifoundation.org) was formed in 1998 with a charter to simplify test system development and maintenance by standardizing instrument driver technology.

² Please refer to the paper “Navigating the Landscape of IVI,” available at www.pacificmindworks.com, for a thorough discussion of why providing both IVI-COM and IVI-C drivers is important for test and measurement customers.

.NET SUPPORT

The native support that .NET languages, such as VB.NET and C#, have for COM components is excellent. Any COM component, including IVI-COM drivers, can be integrated into a .NET client application using standard .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 process of generating these interop assemblies is simple and well-known, as is their use in application development. A single interop assembly can be generated and deployed alongside an IVI-COM driver. The IVI Foundation even provides a specification prescribing how this should be done, so that end users have a consistent experience with IVI-COM drivers in .NET. More importantly, the IVI Foundation provides the interop assemblies required for achieving .NET interchangeability with IVI-COM drivers.

By comparison, using IVI-C drivers in any .NET language is tedious and error prone. Since IVI-C drivers rely on function panel files (.fp) and attribute information files (.sub) that cannot be consumed by Visual Studio, there is no way to infer what IVI-C driver function signatures should look like in .NET. Thus, the end user must supply the function signature along with the requisite .NET interop attributes for every IVI-C driver method they wish to use. Constructing such function signatures directly is extraordinarily burdensome for the end user.

Consider an IVI-C driver that implements the IVI-defined ReadyTrace from the IviSpecAn class specification. This function is very representative of many IVI-C driver functions because it exhibits how IVI-C uses extra parameters to represent array parameter sizes.

```
Vi Status _VI_FUNC ke2810_ReadyTrace (
    Vi Session Vi,
    Vi ConstString TraceName,
    Vi Int32 MaxTimeMili seconds,
    Vi Int32 ArrayLength,
    Vi Int32* Actual Points,
    Vi Real 64 Amplitude[] );
```

This function triggers a measurement and returns the resulting data in the Amplitude output array. As per the IVI specifications, all memory in IVI-C drivers is allocated by the caller. The ArrayLength parameter specifies the number of elements in the Amplitude

array allocated by the caller. The driver implementation populates the Amplitude array with the measured data (up to a maximum of ArrayLength points) and then returns the actual number of elements read in the ActualPoints output parameter.

There is nothing in the C programming language to indicate this very important relationship between Amplitude, ArrayLength, and ActualPoints, so .NET requires this information to be supplied explicitly. In addition, the user must indicate the underlying data type for each of the function parameters. While some of these mappings are obvious, others may not be to some users. For instance, the user may have to consult the IviVisaType.h header file to learn that ViSession is defined as an unsigned 32-bit integer, while the ViStatus return type is defined as a signed 32-bit integer. Mapping string data types from IVI-C drivers to .NET also requires users to know that IVI-C drivers use the ANSI character instead of Unicode (which is used exclusively by IVI-COM drivers and by the .NET platform itself). Thus, the user must adorn their .NET method declaration with an attribute that indicates how string parameters should be marshaled between .NET and IVI-C.

One possible .NET declaration of the ReadYTrace IVI-C function is shown below:

```
[DllImport("ke2810.dll", CharSet=CharSet.Ansi)]
static extern int ke2810_ReadYTrace(
    uint session,
    string traceName,
    int maxTimeMilliSeconds,
    int arrayLength,
    out int ActualPoints,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex=3)]
    double[] Amplitude);
```

The DllImport attribute is used here to indicate which DLL contains the IVI-C function definition. The CharSet property instructs the .NET marshaler to translate .NET strings passed to the driver from ANSI to Unicode. It is very easy to forget the CharSet property, and, confusingly, this property has a different default value for C# and Visual Basic clients than for C++ clients.

The MarshalAs attribute is applied to the Amplitude array parameter to indicate how this parameter should be marshaled. The SizeParamIndex property tells .NET that the 4th parameter (zero-based index of 3) represents the size of the Amplitude array being passed to the IVI-C driver function. It might be expected that the client could somehow indicate that the ActualPoints parameter represents the number of array elements returned by the driver, but this is not supported. This has performance implications because the caller will pre-allocate the maximum number of array elements and all of

this data will be marshaled to the driver. That same number of elements will then be marshaled back to the client, even if the driver only populated a small amount of data. There is no way to tell .NET that some of the data coming back from the driver isn't "real" data – but that it is only "empty" pre-allocated memory.

Not only is the above .NET declaration for an IVI-C function tedious and error prone, there are multiple ways to provide a working declaration that achieves the same result. For instance, one might declare the Amplitude array using the IntPtr type instead of the standard .NET array shown above. Indeed, there may be compelling performance reasons to do precisely that. However, the crucial point is that there are no IVI standards explaining how these .NET declarations should be done or any IVI-shared components provided upon which users can rely. Not only does this lead to an inconsistent user experience, which IVI has sought to eliminate in the first place, but it makes .NET interchangeability with IVI-C drivers impossible. With IVI-COM, users install only the IVI Shared Components and the driver they receive with their instrument. With IVI-COM, users can write interchangeable .NET applications while with IVI-C they cannot.

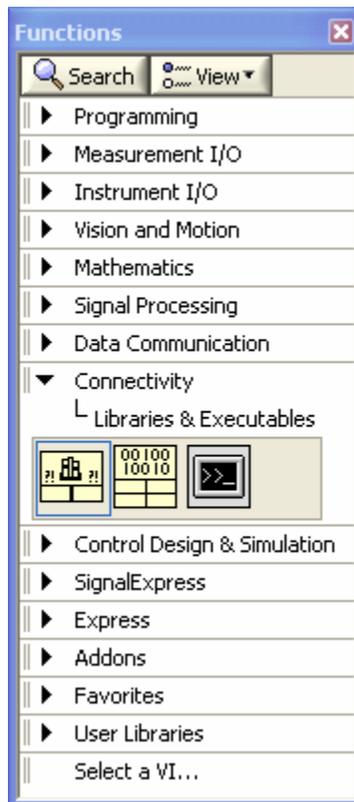
LABVIEW SUPPORT

Perhaps more than any other ADE, LabVIEW creates confusion and misinformation about IVI driver integration. The official position of National Instruments is that IVI-C drivers are the obvious choice for users building LabVIEW applications with IVI drivers. Many developers blindly accept this advice, assuming the ADE vendor would know which type of IVI driver works best in their own environment. However, careful examination of the rationale for this position shows very little technical justification can be found. In fact, as this section will show, IVI-COM drivers in LabVIEW turn out to enjoy several noteworthy advantages over IVI-C.

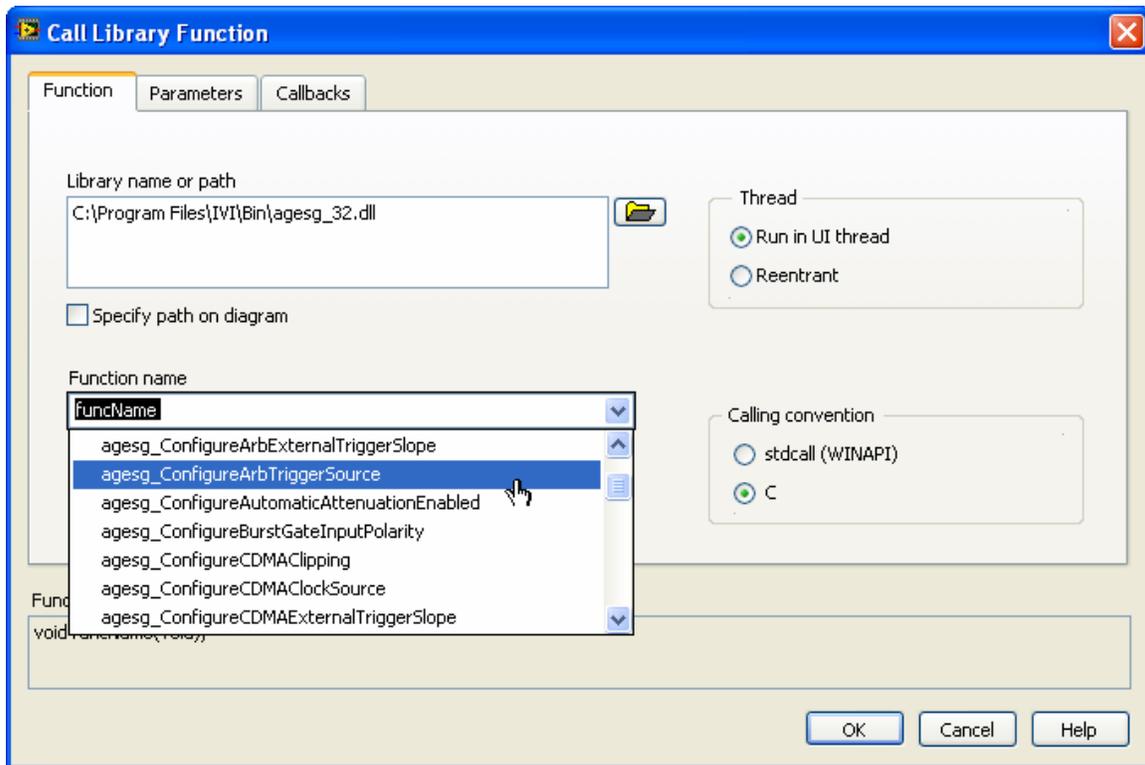
While the native COM support that LabVIEW provides has been available for a number of years, it always has been undersold. With LabVIEW 8.2, the support for native COM has gotten even better with a new Class Browser that exposes a uniform interface for browsing everything from COM objects, such as IVI-COM drivers, to .NET components, DAQ functions, and more. National Instruments always has advocated IVI-C over IVI-COM, so the fact that they found it in their interest to improve LabVIEW's COM support gives testimony to the importance and pervasiveness of COM in general.

IVI-C DRIVERS IN LABVIEW

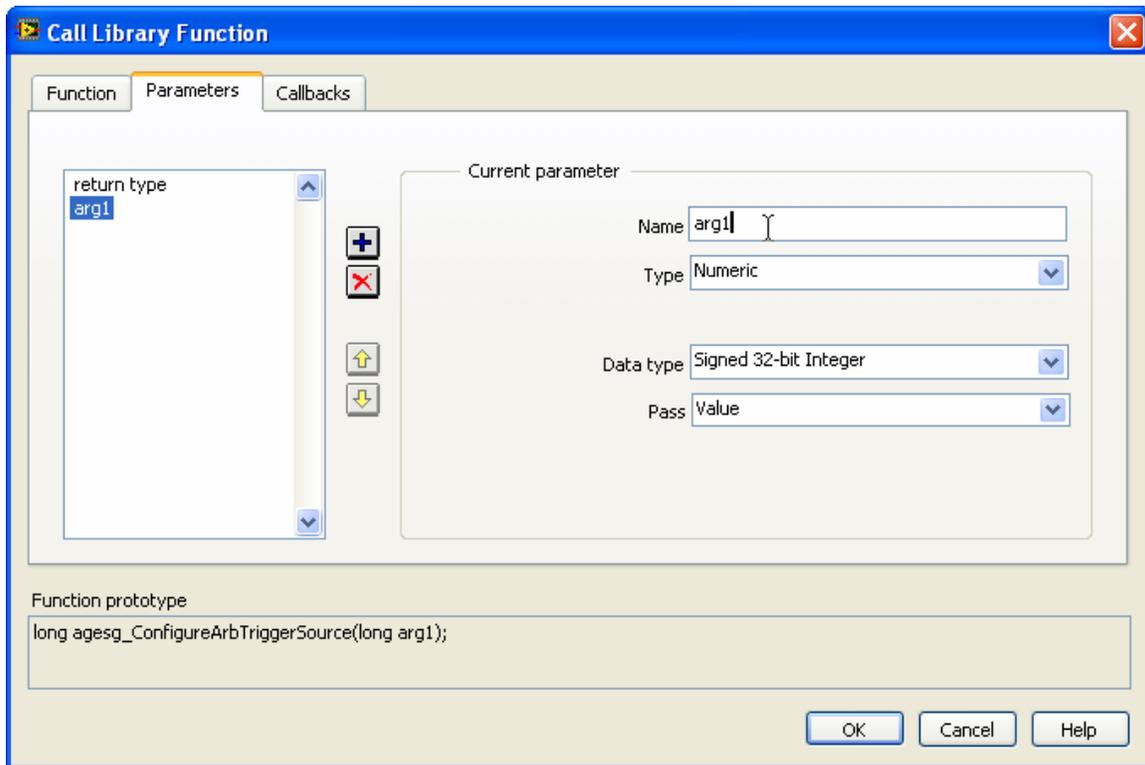
To understand how IVI-C and IVI-COM drivers work in LabVIEW, we start with an examination of LabVIEW's native support for conventional Win32 dynamic link libraries (DLLs). LabVIEW exposes a single primitive called the "Call Library Function Node". It appears in the **Libraries and Executables** subpalette of the **Connectivity** palette.



To call a function in a DLL, such as an IVI-C driver, you drop this node onto your LabVIEW diagram, right click on the node, and select **Configure**. This brings up a dialog that allows you to specify the location of the DLL and the name of the function you want to call.

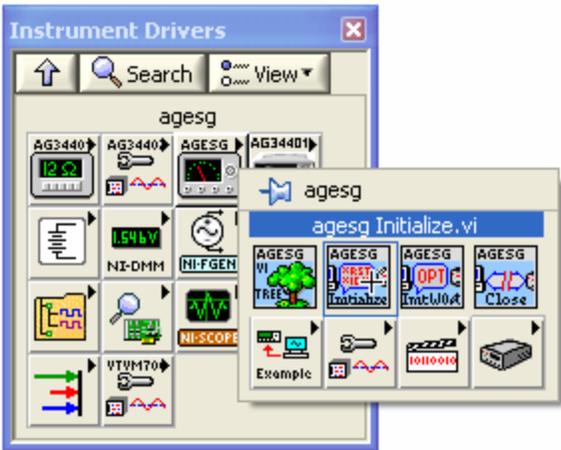


The real difficulty arises when you navigate to the **Parameters** tab, where you realize that you must manually specify each of the function arguments, their data type, and whether they are passed by value or by reference. This is shown in the image below:

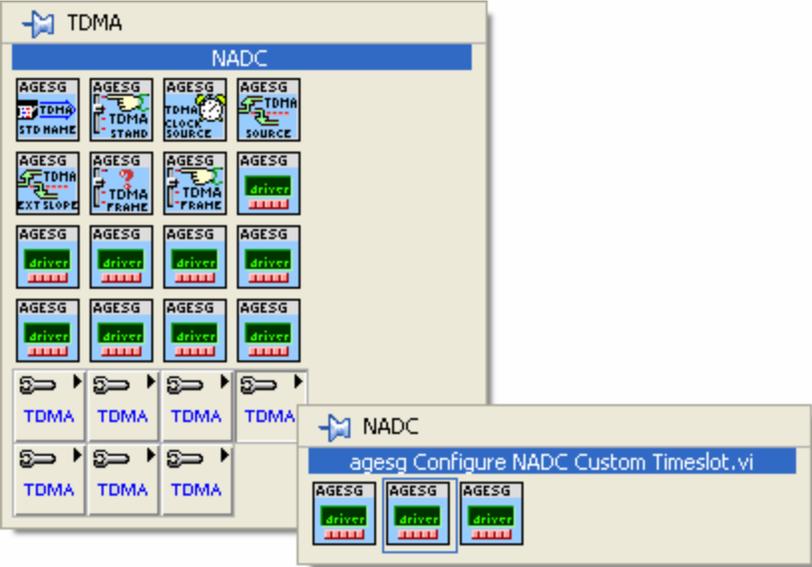


If you get even one of these parameter specifications wrong, then your program will crash – and probably take LabVIEW down with it. At this point, one realizes that there must be a better way. The Call Library Function Node is far too tedious and error prone to be practical. To be fair, it is not this way because the LabVIEW team couldn't figure out how to make it easier. Rather, it is because it is impossible to determine the function prototypes present in a conventional Win32 DLL by looking at the DLL alone. Raw DLLs simply do not contain this information. This point will be revisited later in the section on *Self Describing Components*.

To avoid the laborious process of configuring raw DLL calls in LabVIEW, National Instruments provides a conversion utility that reads the contents of an IVI-C driver's function panel (.fp) file and attribute information (.sub) file and generates LabVIEW Virtual Instruments (VIs) that look very much like other VIs encountered in traditional LabVIEW programming. After running the utility, the driver VIs appear in the Instrument I/O menu under the specific driver palette:

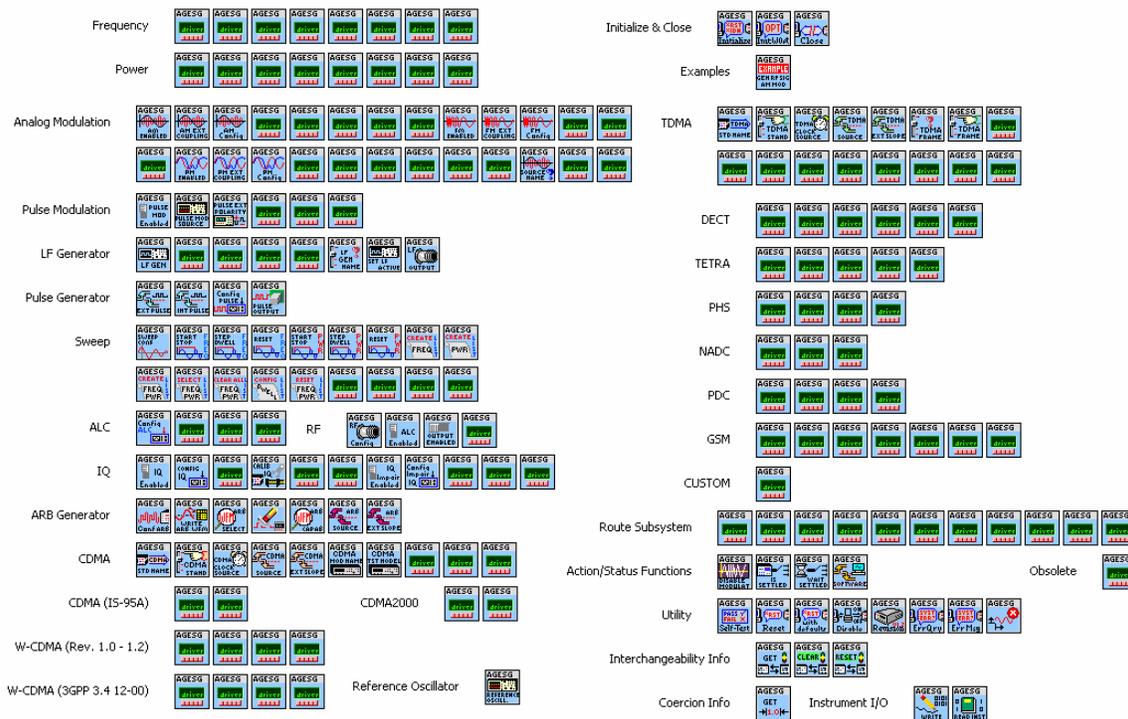


At first glance, this seems like LabVIEW nirvana – a completely unwieldy IVI-C DLL morphed into a familiar and easy-to-use set of LabVIEW VIs. And, to be fair, the tool is useful -- especially considering the alternative of the Call Library Function Node discussed earlier. However, a careful and objective examination of the utility reveals a less than obvious set of choices for the user. The TDMA palette looks like this:



All seven of the submenu icons on the TDMA palette are identical. Moreover, most of the VI icons are identical. Which of those seven submenus takes the user to the PDC

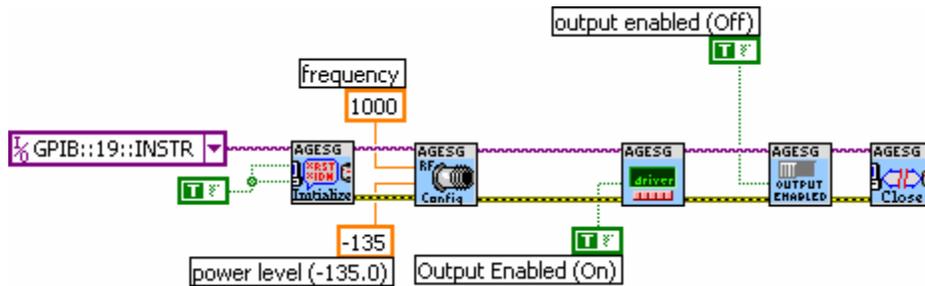
modulation functions? Which of the first-level TDMA functions are used to enable TDMA burst mode and what indication does the user have to guide them to the filter selection VI? This is far from an isolated case, as can be seen by glancing at the entire VI tree for this driver:



Although the diagram is difficult to read, it's easy to see how widespread the icon replication problem is. The IVI-C driver converter used a default icon for every function that was not part of the IVI specifications. Real-world drivers, such as the one above, almost always have more instrument-specific functions than IVI-defined functions. As instruments become more complex, this gulf only will grow for many types of instruments. Other instruments might follow the IVI specification more closely and suffer less from the problem above. The important point is that LabVIEW users need to be prepared for both eventualities. The IVI-C driver used in this discussion is not a contrived example – it is a real-world NI Certified IVI driver available on the NI public web site.

Another important point to make about the wrapper VIs is that all of the driver attributes are missing from the palettes.

Consider a simple application, built using this wrapped IVI-C driver:



This example configures the RF signal, enables modulation, and then enables the output. The diagram looks reasonably clear. Most of the icons are unique and a user can, at a glance, get a pretty good idea of what's going on. (The little type-N connector image is actually quite clever.) One minor annoyance is the difficulty in determining which of the 3rd and 4th VIs is enabling the output and which is enabling the output modulation. Neither icon gives a clear indication nor do the parameter names. The problem actually gets worse as the complexity of the driver increases.

The reason that the simple example above works fairly well is that it uses mostly IVI-defined functions. Real applications will invariably need to stray from this often limited set of capabilities. In these situations, the LabVIEW code for a wrapped IVI-C driver is far less inviting. Two examples of instrument-specific driver programming are shown below. One application configures a TDMA signal and the other configures a CDMA signal. Can you tell which one is which?

Test Application #1



Test Application #2



Even after trolling through the confusing driver Function palette, the LabVIEW programmer is left with code that is mostly opaque.³ What people often fail to remember is that while code may be written only once, it is read many, many times over. Few things are more important in long-term software maintenance than clear, readable code. The test applications above – simple, but far from contrived -- do not fit the bill.

Proponents of the IVI-C wrapper approach would argue that the icons should be “touched-up” to make each of them unique. This is a daunting task and far beyond what most end users would be willing to undertake in order to make their code readable. And, if the underlying IVI-C driver is updated or enhanced, the wrapper generation process would need to be performed again. National Instruments advises each instrument manufacturer to use the converter utility and generate these wrappers to be shipped with the underlying IVI-C driver. A few important obstacles make that far from an ideal strategy for the test and measurement community at large.

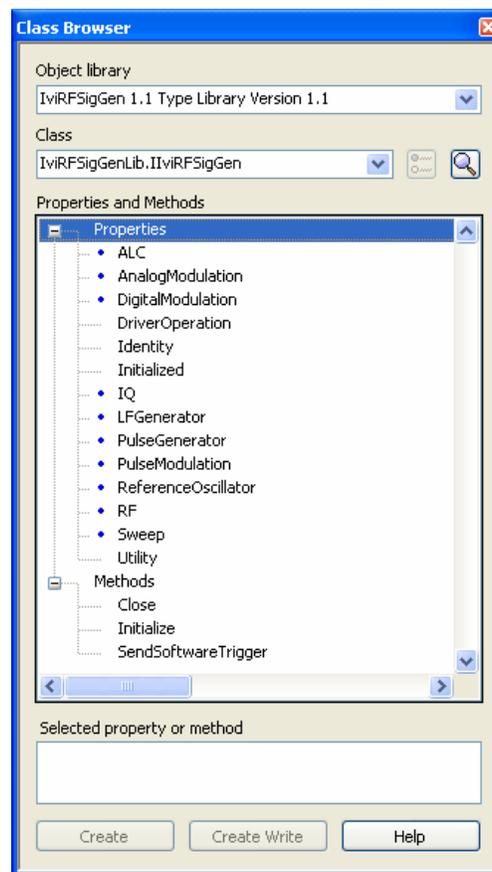
First, the burden of creating unique icons for a realistic driver with many methods and properties is almost always underestimated. This is particularly true of modern instruments where method and property counts easily reach into the hundreds. IVI drivers cover a broad spectrum of instruments, not just a few simple devices for which IVI may have created a class definition. The trend in instrument complexity is up, not down, so this burden will grow, not diminish.

Secondly, the converter utility is not even part of LabVIEW. At one time it was part of LabVIEW, but has been removed and is now a separate download that you must retrieve from the National Instruments' web site. Once there, you must answer a number of questions such as who developed the IVI-C driver, if you plan on sharing the driver, and what specific instrument models you intend to support. This means that instrument manufacturers who choose IVI-C as their base driver for supporting LabVIEW will be complicit in driving end-user traffic to their competitor's web site. To avoid aiding the competition, manufacturers would need to download and run the converter themselves revealing information about products they may have not announced yet. Neither of these consequences offers much confidence in an IVI-C driver strategy for LabVIEW leading to concerns over whether the underlying rationale for IVI-C is technically or commercially motivated.

³ While it is possible to display the VI names above each of the VIs, this adds a surprising amount of clutter to the diagram. This mainly is due to the long names of IVI-C driver functions, which must be globally unique across the entire driver.

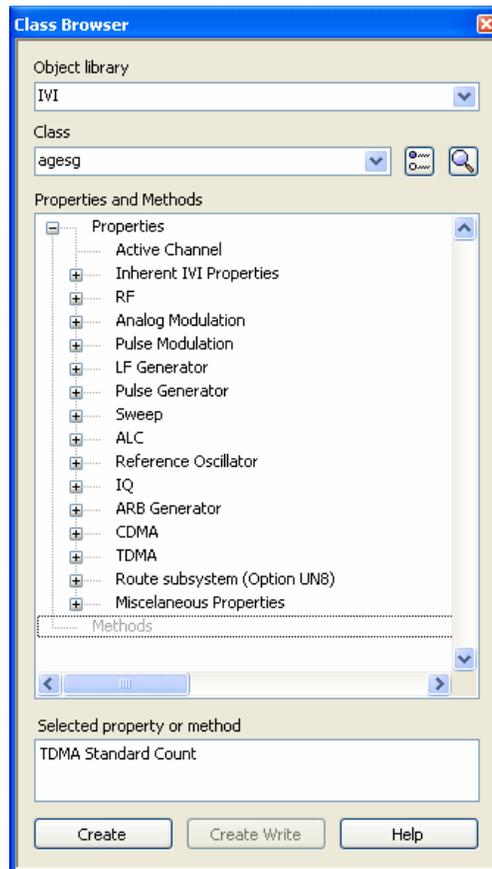
IVI-COM DRIVERS IN LABVIEW

IVI-COM driver access in LabVIEW works in exactly the same fashion as any other COM components. Since COM components are widely available, it is likely that a LabVIEW user will have used these features before, thereby reducing the effort required to integrate an IVI-COM driver. To get started with using an IVI-COM driver in LabVIEW, you call up the Class Browser from the **View** menu.



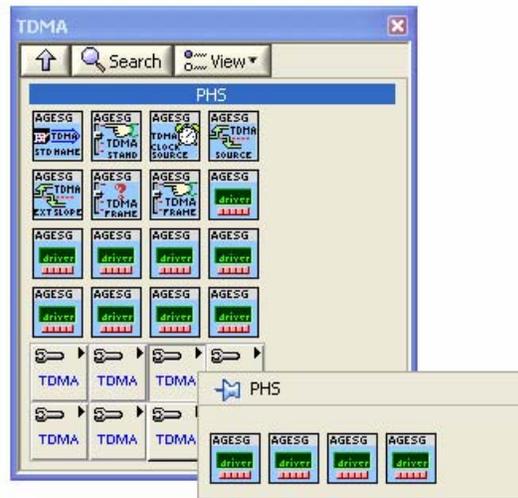
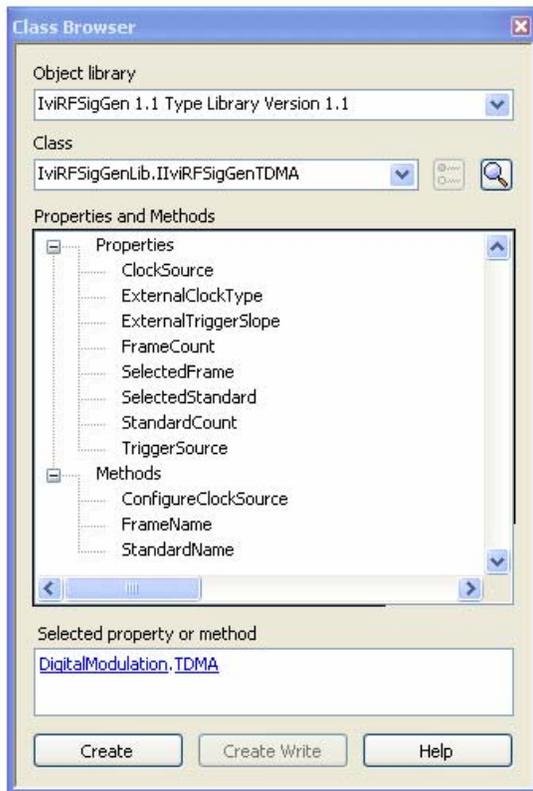
This browser presents a hierarchical view of all methods and properties available in the driver. This format is clearer than the graphical palette shown earlier and additionally shows driver properties, something the IVI-C wrapper palette does not. The fact that IVI-C drivers deal with properties very differently than methods creates confusion in other ADEs as well (this will be discussed more fully in the *Integrated Hierarchy* section). The

user can use the Class Browser to view IVI-C properties just as with IVI-COM, but they cannot see the methods. Here is what the Class Browser displays for the same IVI-C driver presented earlier in the discussion of *IVI-C Drivers in LabVIEW*.



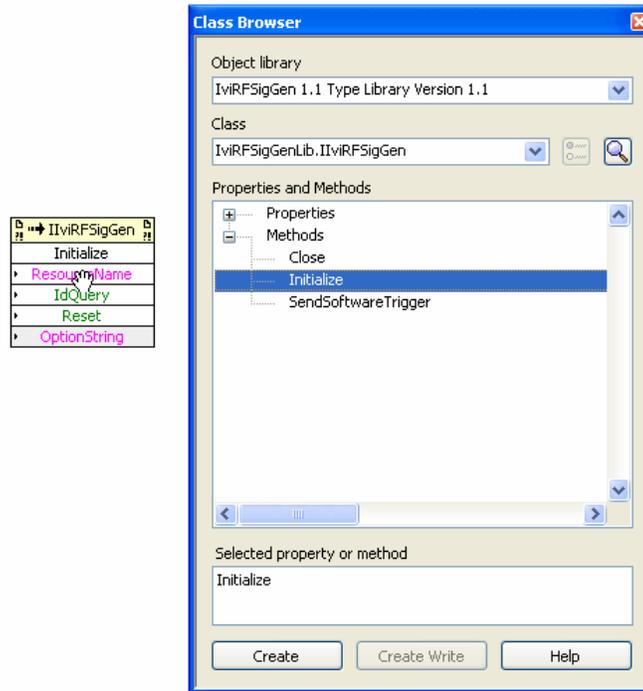
None of the methods for this IVI-C driver are displayed. This means that with an IVI-C driver, the user must toggle between the Class Browser and the menu palette when building an application. Moreover, in spite of what some IVI-C advocate's claim, most of the functionality in IVI drivers (COM or C) is exposed via properties, which means users will be dealing primarily with the Class Browser. The majority of the LabVIEW user experience will be identical with IVI-COM or IVI-C. Additionally, the ability to access IVI-COM methods and properties in once place is unquestionably an advantage IVI-COM drivers have over IVI-C drivers in LabVIEW.

From a navigation perspective, the IVI-COM driver makes it much easier to navigate to specific functionality. Compare what a user sees in navigating the TDMA functionality in the Class Browser with what they see in the IVI-C wrapper menu palette.



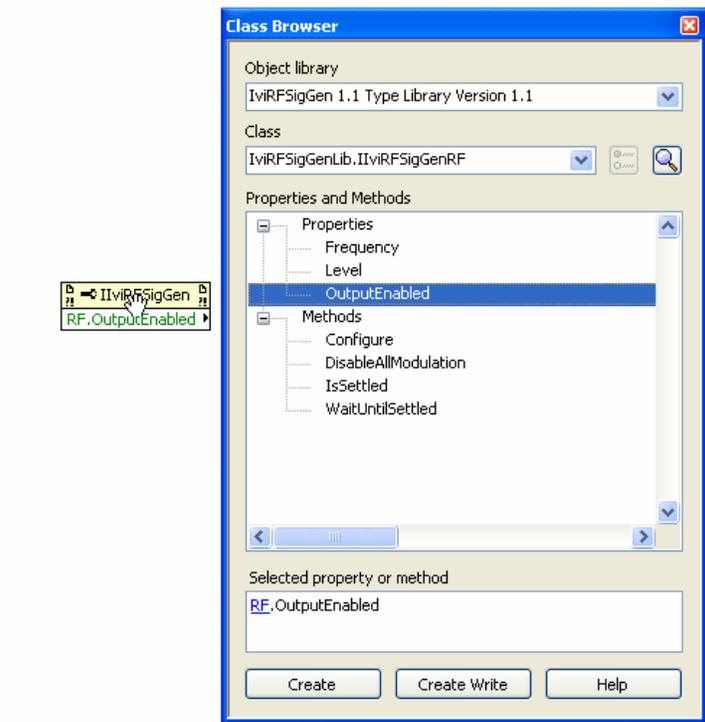
A single, clear interface exposes all IVI-COM driver methods and properties, while the IVI-C wrapper palette is ambiguous and does not provide access to properties. Another feature of the Class Browser is that when you close it and later recall it from the **View** menu, it automatically returns to the same driver at the exact same place in the hierarchy. With the VI menu palette used by IVI-C drivers, you must navigate from the top through the hierarchy each time you want to drop a function on the diagram. Alternatively, you can pin a particular submenu and it will remain visible, but this quickly results in a screen cluttered with little floating palette windows. Given these facts, claims that IVI-C drivers are easier to use in LabVIEW simply don't hold up.

To write LabVIEW applications with an IVI-COM driver, you simply select a method or property from the Class Browser and drag it on to the diagram, as in the image below:

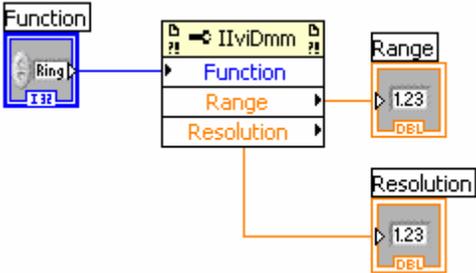


The Class Browser automatically creates a LabVIEW Invoke Node associated with the selected IVI-COM driver's Initialize method. The Invoke Node is well designed, intuitive to use, and understandable when viewed on a real diagram. The Node clearly indicates the IVI-COM parent interface name (IviRFSigGen), the method name (Initialize), and each of the input parameters. If the method had any output parameters, these also would be shown. Each parameter is conveniently color-coded to indicate the data type. For the Initialize node, the ResourceName and OptionString parameters are pink, indicating they are strings. Boolean parameters are indicated by the color green. The OptionString parameter has a slight but easily discernable gray background indicating that this parameter is optional. At a glance, a LabVIEW user easily can see everything there is to know about this method without clicking on a VI to bring up a front panel or without hovering over the node with the mouse. The user can even bring up the driver help page for the method by simply hitting the F1 key with the node selected. This feature is not available for IVI-C drivers in LabVIEW.

IVI-COM driver properties are accessed in exactly the same fashion as methods. Simply select a property from the Class Browser and drag it onto diagram.



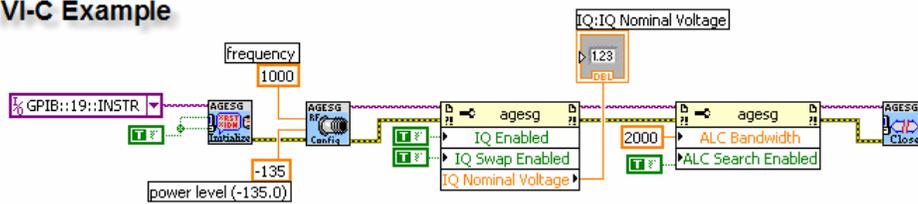
You also can expand the Property Node to access multiple properties at once. In addition, you can specify that some properties should be written and others read back all with a single Property Node. Such a node would look like this:



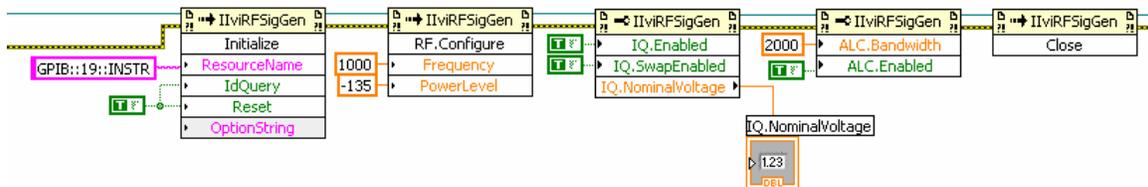
This node sets an IVI-COM DMM driver's Function property and then reads back the Range and Resolution properties. The coloring indicates the data types – blue is an integer type while orange is a floating-point date type. The input control and output indicators also have small annotations (“I32” for 32-bit integer and “DBL” for double-precision floating point) that provide a convenient visual cue.

LabVIEW code for accessing an IVI-C driver's attributes is identical to that shown above for IVI-COM. Property nodes are used in precisely the same fashion for both flavors of IVI drivers. Since real-world drivers consist primarily of properties, LabVIEW applications built on IVI-COM drivers will look very close to their IVI-C counterparts. As an example, the code below shows the same simple task of configuring an RF signal using an IVI-C driver and an IVI-COM driver:

IVI-C Example



IVI-COM Example



Even though this particular example shows three method Invoke Nodes and only two Property Nodes, the diagram real estate used is almost exactly the same. In a more realistic example, the properties would likely outnumber methods, so the diagrams would be even more similar than what is shown. Looking at the IVI-COM code, the LabVIEW user can easily see at a glance what is happening with each and every node – the method and property names are readily visible. With IVI-C code, though the properties are just as clear as with IVI-COM, the methods are more opaque. To know what is going on, the user has to visually decrypt the method icons or hover over each VI with the

mouse. Realistically, only the original icon developer can be expected to easily decipher these compact but indistinct visuals.

LABWINDOWS SUPPORT

The one environment where IVI-C drivers have a decided advantage over IVI-COM is National Instruments LabWindows/CVI, the environment for which IVI-C drivers were originally designed. LabWindows parses and displays IVI-C function trees and attribute hierarchies and renders soft front panels with graphical controls for each of the IVI-C driver methods. Suffice it to say that IVI-C drivers are very much at home in LabWindows.

IVI-COM drivers, however, fare poorly when used in LabWindows. End users 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. In fact, in these files, the MIDL compiler does not generate the proper data structures required if a C client needs to access the IVI-COM driver. The driver developer must manually “massage” these files to provide a working interface for C users. Not only is it likely a client might be provided with a driver that did not properly do this customization, there is no IVI-defined standard that guides driver providers in creating these files. Moreover, IVI does not require these files to be provided at all, meaning an end user could very easily end up completely on their own in integrating their IVI-COM driver with C.

LabWindows programmers using IVI-COM drivers also must contend directly with COM data types, such as VARIANTs, SAFEARRAYs, and BSTRs. The convenient C++ wrappers available to Visual C++ users are not available in LabWindows. As an example, consider the code required to create a COM SAFEARRAY and populate it with a few data values.

```
/* Create an array of 3 double elements */
SAFEARRAY* psa = NULL;
double *pData = NULL;
psa = ::SafeArrayCreateVector(VT_R8, 0, 3);
if (psa == NULL)
return E_OUTOFMEMORY;

/* Populate the array with data */
::SafeArrayAccessData(psa, (void HUGE **)&pData);
pData[0] = 3.14;
pData[1] = 1.00;
pData[2] = 2.79;
```

```
// Pass pData to IVI-COM driver
IAgent34401* pDriver = // ...
IAgent34401_Configure(pDriver, pData);
```

In any other environment, the code to perform this very basic IVI-COM driver task would be much simpler. In C#, the code would be trivial:

```
double[] data = { 3.14, 1.00, 2.79 };

// Pass data to IVI-COM driver
IAgent34401 driver = // ...
driver.Configure(data);
```

IVI-COM drivers are too unwieldy to be useful in LabWindows CVI. If you need to provide a reasonable experience for LabWindows users, then you must supply an IVI-C driver in addition to your IVI-COM driver.

CLASS DRIVERS

Interchangeability with IVI-C relies upon a special kind of driver known as a *class driver*. These class drivers expose generic function entry points that delegate to a *specific driver* written for a particular instrument. Users that want to write an interchangeable application use only functions exposed by the class driver. The following code demonstrates how to use an IviDmm class driver to write interchangeable code.

```
ViSession session;
IviDmm_InitWithOptions("MyDmm", VI_FALSE, VI_FALSE, VI_NULL,
    &session);
IviDmm_ConfigureTrigger(session, IVI_DMM_VAL_EXTERNAL, 0.1);
IviDmm_SetAttributeViReal64(session, VI_NULL,
    IVI_DMM_ATTR_APERTURE_TIME, 0.35);
```

The "MyDmm" parameter passed to the `InitWithOptions` function is mapped to a specific driver in the IVI Configuration Store.⁴ The call to `ConfigureTrigger` will call the class driver DLL `ConfigureTrigger` function, which looks up "MyDmm," locates the associated specific driver and subsequently invokes the `ConfigureTrigger` function implemented in the specific driver. All of the function names, attribute names, and attribute value names used in the application start with the generic `IviDmm` prefix, so the code has no dependencies on a specific driver. To use this code with a different instrument, only the IVI Configuration Store entry for "MyDmm" needs to be updated to point to the new instrument. The code itself does not need to be recompiled or re-linked.

Though the IVI-C code above works just fine, the trouble is that class drivers are proprietary products as opposed to shared components distributed and maintained by the IVI Foundation. In addition, a different class driver must be provided for each instrument class. National Instruments is the only company that provides IVI class drivers, so companies implicitly rely on a single supplier. For other instrument manufacturers, relying on a competitor to deliver the required components for IVI interchangeability is not an optimal strategy. Yet, with IVI-C, there are no other practical options.

If an end user or instrument manufacturer wants to achieve interchangeability for an instrument not defined by IVI, they can author what IVI terms a *custom instrument class*. For instance, a company such as Nokia may wish to define a generic cell phone instrument class to standardize functionality common to all of its mobile devices. For this to work with IVI-C, Nokia would have to author an IVI-C class driver. This involves implementing all of the delegation logic described above, as well as special error handling and session management logic prescribed by the IVI-C specifications.

By contrast, IVI-COM drivers require no special proprietary components to build interchangeable applications. The code for implementing IVI-defined functions is housed in the same DLL as the code for implementing instrument-specific functions. The following code shows how to write interchangeable code with an IVI-COM driver.

```
IviSessionFactory fac = new IviSessionFactoryClass();
IviDmm dmm = (IviDmm)fac.CreateDriver("MyDmm");
dmm.Initialize("MyDmm", true, true, "");
dmm.Trigger.Configure(
    IviDmmTriggerSourceEnum.IviDmmTriggerSourceExternal, 0, 1);
```

⁴ The IVI Configuration Store is an XML file and an accessor component known as the IVI Configuration Server. Both are freely distributed by the IVI Foundation (www.ivifoundation.org) as part of the IVI Shared Components, which must be installed in order to build applications that use an IVI-COM or IVI-C driver.

```
dmm. Advanced. ApertureTime = 0.35;
```

The `IviSessionFactory` component is provided by the IVI Foundation and automatically installed as part of the IVI Shared Components. Similar to the IVI-C class drivers, the `IviSessionFactory` locates a specific driver by looking up "MyDmm" in the IVI Configuration Store. A reference to the IVI-COM driver is returned in the "dmm" variable. The difference here is that the `IviSessionFactory` works with any instrument class – IVI-defined or user-defined. New instrument classes can be developed without authoring anything akin to the custom class driver that would be required for IVI-C. Moreover, the `IviSessionFactory` is not involved at all after the call to `CreateDriver`. From that point on, the user is accessing the IVI-COM driver directly.

SELF DESCRIBING COMPONENTS

The principal reason COM components, such as IVI-COM drivers, offer benefits over conventional Win32 DLLs, such as IVI-C drivers, is that the COM binary itself is fully self-describing. During compilation, the compiler preserves much of the detailed data it processes from source files and bakes a standard format *type library* directly into the DLL. As a result, you can open a COM DLL in any number of readily available object browsers and discover a great deal of important information, such as:

- Available interfaces.
- Names and full signatures of methods and properties.
- Directional information for parameters (input, output, input/output).
- Definitions and allowed values for enumerated types.
- Component version information.
- Help content for methods, properties, enums, and interfaces.

The presence of this rich library of information, generically referred to as *metadata*, forms the foundation for virtually every important feature that COM provides – from IntelliSense, to versioning, to remote access. The Microsoft .NET framework can accurately be thought of as an evolution of metadata. The impressive features available in .NET – from garbage collection to just-in-time compilation, can all be traced back to the richer metadata available in .NET DLLs.

Conversely, conventional Win32 DLLs have very little metadata. If you run the `dumpbin.exe` command line utility on an IVI-C driver, you will only see a list of function names. There is no information about such things as the type or number of method parameters or which attributes are supported. This is not because of a defect or missing

feature in the dumpbin utility, it is simply because the metadata is not in the DLL at all. At compilation time, the compiler discards the details in the header files and source files it processes. This is why end users need header files, which are not required in COM. It also is why users can easily import COM components into .NET environments, whereas a great deal of tedious coding is required for IVI-C. All of the metadata for IVI-C must be supplied manually as was demonstrated in the section on *.NET Support*.

To compensate for the lack of intrinsic metadata, IVI-C relies on additional external files to describe a driver – all of which are required by the IVI-C specifications. Function layout is described in the function panel (.fp) file, while attributes are described in the attribute information file (.sub). Linking information for driver functions is supplied in the import library (.lib) while the header file (.h) defines method signatures and data types. IVI-COM drivers describe with a single DLL what IVI-C drivers require five separate files to describe. Moreover, the five IVI-C files are dispersed, as per the IVI-C specifications, in four different directories.

To use an IVI-C driver in an application, you must make your development environment aware of all of these files and where they reside on disk. For instance, in Visual Studio you have to specify where the header file and import library live in order to compile an application, and then you have to make sure the environment can locate the driver DLL at runtime. Different ADEs have different mechanisms for doing this, so users must contend with these often unfamiliar and inconsistent features. With IVI-COM, not even the location of the single COM DLL need be specified. The entire contents of the driver can be imported with a single `#import` directive:

```
#define AGILENT34401_LIBID /
    "Iid: 74ed456f-1249-4a47-9108-811916aed143"
#import AGILENT34401_LIBID
```

The `#import` directive above makes no references to any directory. This is because IVI-COM drivers are not located based on a path; rather they are identified by a globally unique identifier (GUID). This makes it easy to write applications that will compile and run without modification even if the driver install directory changes or is not known by the developer.

Because of the self-describing and self-contained nature of COM, tooling and ADE support will continue to grow and evolve over time – as we've already seen in the excellent support .NET provides for COM and even in the purportedly "IVI-C friendly" LabVIEW environment. The prevalence of COM in the Windows Vista operating system gives testimony to its staying power. Most core operating system features in Windows Vista continue to be implemented using COM, not .NET, as many would be led to

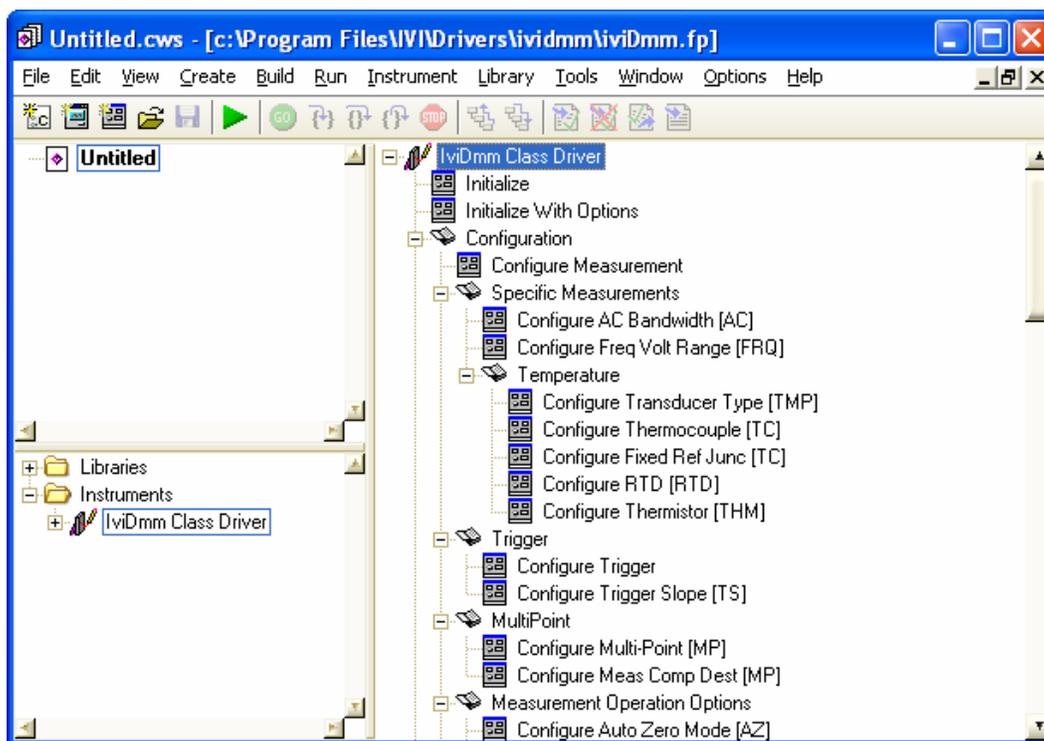
believe. Suffice it to say, COM will remain an important component technology on Windows for the foreseeable future.

INTEGRATED HIERARCHY

Modern instruments increasingly have large functional surface areas. Providing a driver that is both easy-to-use and functionally complete requires careful attention to the layout of the driver hierarchy. Both IVI-COM and IVI-C drivers allow users to define hierarchies of methods and properties, but IVI-C hierarchies suffer from a few noteworthy flaws.

One of the first things users confront when dealing with IVI-C driver hierarchies is the fact that functions and attributes are stored as two separate hierarchies. The function hierarchy is stored in the function panel (.fp) file, while the attribute hierarchy is stored in the attribute information (.sub) file. The .fp file uses a somewhat antiquated binary file format, while the .sub file uses simple ASCII data. LabWindows/CVI displays these two hierarchies separately, making it difficult to view related bits of driver functionality in one place.

The following image shows how LabWindows displays the function hierarchy.



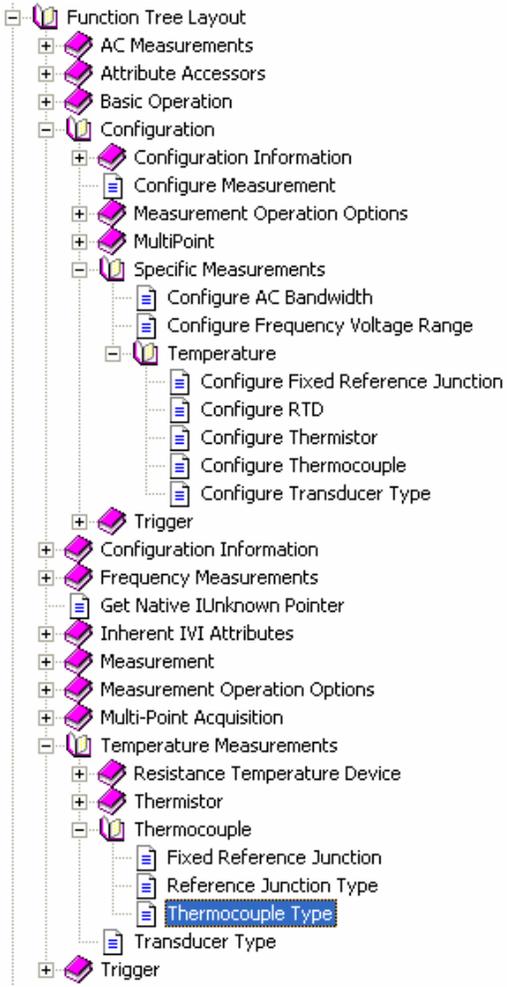
In order to view the attribute hierarchy, the user must navigate to one of the attribute accessor functions (such as SetAttributeViReal64), open the associated panel, click the "Operate Function Panel Window" button, and finally click on the Attribute ID control.

This brings up the following attribute hierarchy:



While this navigation is certainly inconvenient, it is a feature of LabWindows and not fundamentally a limitation of IVI-C drivers. Considering that LabWindows is the principal environment where IVI-C drivers are used, it is important to consider these behaviors. However, even if an ADE managed to merge the function and attribute hierarchies of an IVI-C driver and display those together, more fundamental problems would immediately become apparent.

Pacific MindWork's Nimbus Driver Studio software renders IVI-C attributes and functions in a single hierarchy. The image below presents the IVI-C function tree layout in a Nimbus-generated help file for an IVI-C DMM driver. The hierarchy shown presents only the items defined by the IVI-C portion of the IviDmm specification.



The trouble with this hierarchy is that the IVI specifications have actually defined IVI-C functions in different locations than their associated attributes. Notice that the Configure Thermocouple function is located in a completely different portion of the hierarchy than the Thermocouple Type attributes it controls.

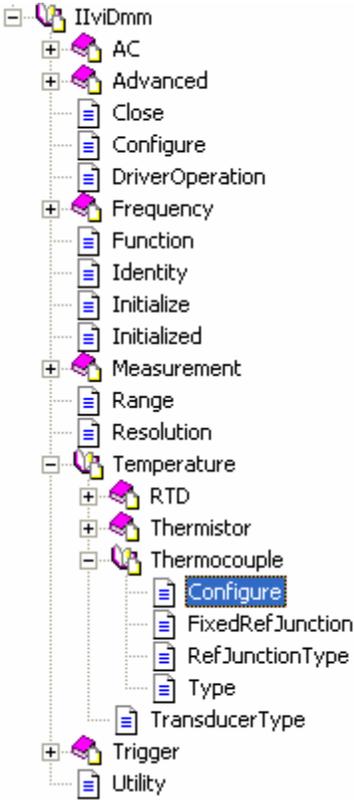
The "path" to the Configure Thermocouple function is:

`\Configuration\Specific Measurements\Temperature\Configure Thermocouple`

And the path to the Thermocouple Type attribute is:

`\Temperature Measurements\Thermocouple\Thermocouple Type`

This is far from an exceptional case in the IVI-C specifications; rather, it is the norm. The bi-furcated IVI-C hierarchy is fundamentally unavoidable and is a source of significant user confusion and inconvenience. Contrast that with the hierarchy for an IVI-COM DMM driver:



The IVI-COM hierarchy is noticeably simpler than the corresponding IVI-C hierarchy, even though they both represent the exact same functionality. For example, the Configure Thermocouple function is located in the exact same place as its associated attributes, including the Thermocouple Type attribute.

NAME COLLISIONS

All function and attribute names in IVI-COM are scoped by the interface in which they are defined. You can have a method called "Configure" in the IviDmmMultiPoint interface along with a completely different method called "Configure" in the

IviDmmThermocouple interface. Users distinguish between the two quite easily in their application code:

```
I Ivi Dmm dmm = // ...
dmm. Trigger. Multi Point. Configure(...);
dmm. Temperature. Thermocouple. Configure(...);
```

With IVI-C drivers, all functions and attributes are globally scoped, so they must have globally unique names, such as "ConfigureMultiPoint" and "ConfigureThermocouple".⁵ The IVI-C application code also easily discriminates between the two:

```
ag34401_ConfigureMulti Point(...);
ag34401_ConfigureThermocouple(...);
```

The problem that occurs with IVI-C is when a driver defines a method that logically does the same thing as an IVI-defined method, but does not match the exact signature of its IVI-defined equivalent. For example, consider the IVI-C definition of the ConfigureThermocouple function:

```
Ivi Dmm_ConfigureThermocouple(Vi Session vi ,
Vi Int32 thermocoupleType, Vi Int32 refJunctionType);
```

Even though the IVI-C driver might support this class-compliant function, it is more likely that the instrument-specific portion of the driver design will call for a ConfigureThermocouple function that has a different signature. The Agilent 34980A driver, for example, supports the IviDmm interfaces, but defines its thermocouple configuration function such that, in IVI-C, it would have the following signature:

```
ag34980_ConfigureThermocouple(Vi Session vi ,
Vi ConstString channelList, Vi Int32 Resolution,
Vi Int32 thermocoupleType);
```

Since IVI requires the IVI-defined version to be named "ConfigureThermocouple," and since both versions cannot have this same name, the driver developer is left with a difficult choice. The developer must expose only the IVI-defined function and leave the instrument-specific one out completely, or they must give the instrument-specific version a slightly different name. Since both functions do the same thing, any different name will

⁵ As per the IVI specifications, the IVI-defined functions and the instrument-specific functions in an IVI-C driver all start with the same specific driver prefix, such as "ag34401_". You cannot differentiate between IVI-C functions based on the instrument prefix at all.

inevitably lead to user confusion. Should the instrument-specific function add a suffix, such as “2”, “Ex”, “New” or simply “Specific”? The user would be confronted with something like the following in the driver header file:

```
ag34980_ConfigureThermocouple(ViSession vi,  
    ViInt32 thermocoupleType, ViInt32 refJunctionType);
```

```
ag34980_ConfigureThermocoupleSpecific(ViSession vi,  
    ViConstString channelList, ViInt32 Resolution,  
    ViInt32 thermocoupleType);
```

Which should the test programmer use in their application? Even after a unique name has been chosen, the driver developer must decide where in the IVI-C hierarchy to put the instrument-specific version. If they put it in the hierarchy next to the class-compliant version, then test programmer sees two confusingly similar names in the function tree right next to one another. If the driver developer attempts to avoid this problem by burying the instrument-specific version in a different portion of the function tree, then how would a test programmer who comes across the class-compliant version first even know they might need to look elsewhere in the tree for the more fully functional instrument-specific version?

The bottom line is that there are no good options. Furthermore, there are no IVI requirements or guidelines for implementing any of the bad options. This leads to obscure and distressing inconsistencies between instrument manufacturers and is an intrinsic weakness in the IVI-C driver architecture. Instruments are becoming more complex, not simpler, which means the mismatch between IVI-defined functions and instrument-specific functions will grow. This problem will only get worse over time.

USING PROPERTIES

IVI drivers consist of methods and properties. Independent of the instrument's complexity, there typically is a greater number of properties than methods. The more complex the instrument, the more properties tend to outnumber methods. Users demand fine-grained control over the instrument – often for flexibility and performance reasons. Thus, the ease with which driver properties are accessed is an important consideration when comparing IVI-COM and IVI-C. For a variety of reasons that will be detailed in this section, accessing driver properties is much simpler with IVI-COM drivers than with IVI-C. Consider the following simple C# code fragment that initializes an IVI-COM driver and sets the Resolution property:

```
IVI Dmm dmm = new Agilent34401();  
dmm.Initialize("GPIB::6", true, true, "");  
dmm.Resolution = 0.35;
```

The same operation with an IVI-C driver requires using an *attribute accessor* function, as in the following example:

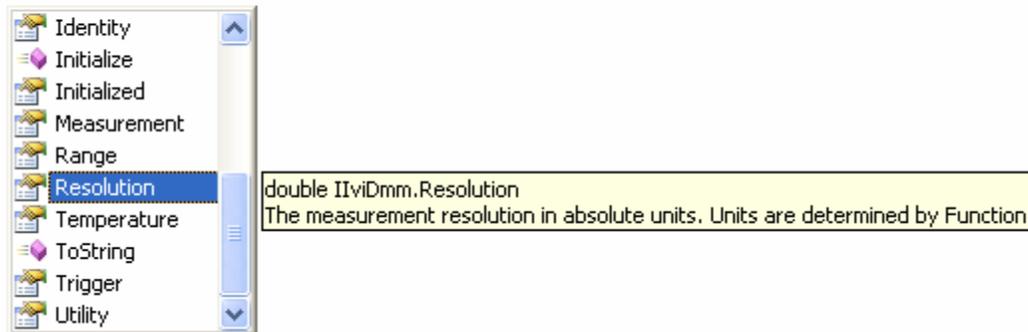
```
ViSession session;  
ViStatus status;  
status = ag34401_InitWithOptions("GPIB::6", VI_TRUE, VI_TRUE,  
    &session);  
status = ag34401_SetAttributeViReal64(session, VI_NULL,  
    AG34401_ATTR_RESOLUTION, 0.35);
```

The IVI-COM driver code is certainly much more concise than the IVI-C driver code, especially considering the fact that the IVI-C example is not even checking the return status for errors. The IVI-COM code implicitly does this without the need to explicitly test function return values. This important advantage will be discussed further in the *Error Handling* section.

THE INTELLISENSE EXPERIENCE

Developers need to consider the test programmer's IntelliSense experience when setting properties on IVI drivers. IntelliSense makes it much easier to reference methods or properties on an object by displaying a list of members after you type an object name followed by a period. Developers now rely very heavily upon IntelliSense, so optimizing its behavior is an important consideration for IVI drivers. With the IVI-COM DMM code above, after the developer types "dmm" and then a period, they are immediately presented with a short list of methods and properties available in that context, as in the figure below.

```
IIVIvDmm dmm = new Agilent34401Class();
dmm.Initialize("GPIB::6", true, true, "");
dmm.R
```



With very little typing or *a priori* knowledge of the driver, users can quickly use IntelliSense to explore the entire set of methods and properties available. Note that even the help information for the selected method is displayed and tracks the user's selection. Many programmers claim that IntelliSense often is more useful and convenient than help files for initially understanding what functionality an IVI-COM driver offers.

When typing the equivalent code for the IVI-C driver, IntelliSense is far less effective. After the programmer types "ag34401" (the first characters of the attribute accessor), IntelliSense is not automatically activated. Even if the user knows how to activate IntelliSense by typing Ctrl-spacebar, the list presented will include everything in the global namespace – which is an enormous number of data types and functions. All of the IVI-C functions appear together in a flat list – alongside the IVI-C attributes. Modern IVI drivers might have hundreds of functions, so scrolling through the list to select the desired function becomes tedious. If the developer doesn't already know the exact name of the function they want to use, they will quickly become frustrated scrolling through this global flat list. Even if the developer does know the exact name of the function, they will have to type a significant portion of the name before IntelliSense navigates close enough to the desired function for them to select it. For example, many IVI-C functions start with "Configure", such as "ConfigureRange". For the programmer to call this function, they will likely need to type "ag34401_Conf", followed by Ctrl-spacebar, and then scroll to the "ag34401_ConfigureRange" function. By the time they have performed all of the required keystrokes, they have lost most of the value of IntelliSense.

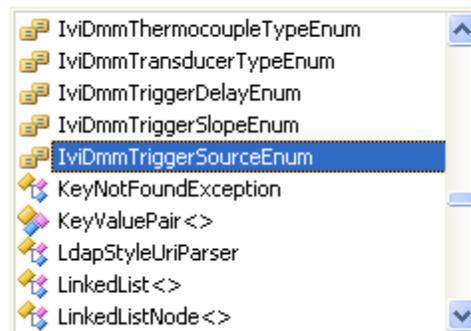
This IntelliSense impediment is actually encountered twice for each call to an attribute setter or an attribute getter. Consider again the SetAttributeViReal64 function:

```
ag34401_SetAttributeViReal64(session, VI_NULL,
    AG34401_ATTR_RESOLUTION, 0.35);
```

The third parameter to this function is the attribute ID. It specifies which attribute should be set. Even after the user has typed the SetAttributeViReal64 function name, they will have to locate the AG34401_ATTR_RESOLUTION constant for the attribute ID parameter. At a minimum, they will have to manually type "AG34401_ATTR" before they can easily locate the desired attribute ID, because all IVI-C attributes share this prefix.

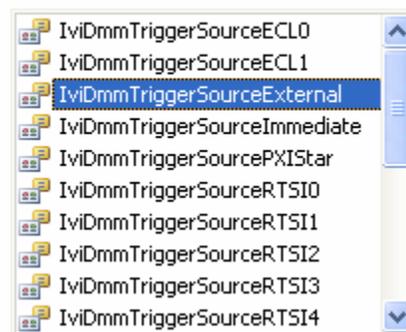
Properties that use enumerated types show perhaps the biggest difference in the IntelliSense experience with IVI-COM and IVI-C drivers. For example, consider the TriggerSource property for a DMM. With IVI-COM, the proper enumeration for TriggerSource automatically will be presented to the user as they type, as in the figure below:

```
dmm.Trigger.Source =
```



Typing a period (or pressing the tab key) displays the allowed values for the TriggerSource enumeration:

```
dmm.Trigger.Source = IviDmmTriggerSourceEnum.
```



In the end, the property assignment above requires 72 characters, only about eight of which the user had to type after the initial "dmm" (this includes punctuation and whitespace).

IVI-C drivers use #define'd constants to represent enumerated values, as in the following:

```
ag34401_SetAttributeVInt32(session, VI_NULL,
    AG34401_ATTR_TRIGGER_SOURCE, AG34401_VAL_EXTERNAL);
```

As with the attribute ID parameter, IntelliSense can offer little support in locating valid constants for the attribute value. All of the associated constants start with "AG34401_VAL_", so the user must at least type that much of the name manually before IntelliSense can help at all. Worse still, allowed values for a specific property will be listed alphabetically in IntelliSense – not grouped according to the attribute to which they apply. This makes it difficult to determine which values go with which attribute and requires the user to scroll through the entire list of constants defined for the entire driver. In the current example, consider the number of items that would occur between the trigger source value AG34401_VAL_EXTERNAL and AG34401_VAL_TTL7. Since "E" and "T" are on opposite ends of the alphabet, these values will be separated by almost all of the defined attribute values for the entire driver.

Though seemingly minor annoyances, when multiplied across dozens of attribute operations in a typical application these syntactic battles compromise the end-user experience.

TYPE-SAFE PROPERTIES

The IVI-C attribute mechanism is susceptible to specific kinds of errors that cannot occur with IVI-COM drivers. Consider the following erroneous IVI-COM code which attempts to assign a floating-point value to a property that accepts an integer data type:

```
dmm.Scan.Count = 15.5; // ERROR: Compiler reports problem
```

This error will be caught at compile time because the property is type safe – the compiler inherently knows that Count is an integer. It isn't possible for the programmer to ship a driver with such a flaw.

With IVI-C drivers, all of the attributes are represented by #define'd integer constants, such as AG34401_ATTR_SCAN_COUNT. The compiler has no information as to the proper data type for the IVI-C Count property and there is no way to detect the following commonly encountered error:

```
// ERROR: Undetected until runtime
ag34401_SetAttributeViReal64(session, VI_NULL,
    AG34401_ATTR_SCAN_COUNT, 15);
```

The above code is mistakenly using the attribute accessor for floating-point data types to set an integer attribute. Even though the value 15 appears to be compatible, this code will return an error at runtime.

REPEATED CAPABILITIES

Instruments often contain multiple instances of the same type of functionality. An oscilloscope, for instance, might have several channels with the same measurement capabilities, or a spectrum analyzer might support multiple traces from a series of acquisitions. IVI refers to these as *repeated capabilities*, and both IVI-COM and IVI-C drivers provide standard mechanisms for accessing functionality on a repeated capability.

Recall the definition of the SetAttributeViReal64 function:

```
Vi Status IviDCPwr_SetAttributeViReal64(ViSession vi,
    ViConstString repCapName, ViAttr attrId,
    ViIntReal64 value);
```

The repcapName parameter specifies the repeated capability instance to which the attribute operation should be applied. For instance, to enable output channel 1 on a DC power supply driver, the user code would look similar to the following:

```
IviDCPwr_SetAttributeViBoolean(session, "Channel 1",
    IVIDCPWR_ATTR_OUTPUT_ENABLED, VI_TRUE);
```

In practice, several operations on the same repeated capability often are performed together, requiring code such as the following:

```
IviDCPwr_SetAttributeViReal64(session, "Channel 1",
    IVIDCPWR_ATTR_CURRENT_LIMIT, 12.5);

IviDCPwr_SetAttributeViReal64(session, "Channel 1",
    IVIDCPWR_ATTR_VOLTAGE_LEVEL, 3.0);

IviDCPwr_SetAttributeViBoolean(session, "Channel 1",
    IVIDCPWR_ATTR_OUTPUT_ENABLED, VI_TRUE);
```

With IVI-C repeated capabilities, the user must take care to pass in a valid repeated capability name to each of the attribute setters AND ensure that the same string is passed to the function. As with many other aspects of the IVI-C attribute model, the compiler cannot help at all in flagging these errors at compile time.

IVI-COM repeated capabilities primarily use collections to access per-instance functionality. The following example shows how to set properties on the same instance of the DC power supply output channel:

```
IIVI DCPwrOutput output = dc.Outputs.getItem("Channel 1");  
output.CurrentLimit = 12.5;  
output.VoltageLevel = 3.0;  
output.Enabled = true;
```

With IVI-COM, a single call to the Item method retrieves the correct output channel. All subsequent operations will inherently be performed on the same, correct output channel. This advantage becomes even more compelling when considering repeated capabilities nested within other repeated capabilities. IVI terms these *nested repeated capabilities*. For example, a device might have multiple triggers for each output. Accessing nested capabilities is arguably clearer with IVI-COM drivers, as the syntax is a natural extension of the non-nested case. For example, consider the following code:

```
I Agilent34401Trigger trig =  
    driver.Outputs.getItem("Out1").Triggers.getItem("Trig1");  
trig.Level = 0.45;
```

It is clear from the code that the trigger repeated capability is nested within the output repeated capability. The IVI-C equivalent requires the use of special IVI-defined nested repeated capability syntax:

```
ag34401_SetAttributeVirtual64(session, "Out1:Trig1",  
    AG34401_ATTR_TRIGGER_LEVEL, 0.45);
```

The string "Out1:Trig1" denotes a trigger nested within an output. However, users would likely have to consult the IVI-3.1 Architecture specification to discover this IVI-defined notation. Other users reading the code also would not necessarily understand the required colon notation as easily as they would the equivalent IVI-COM syntax.

A more fundamental problem with IVI-C repeated capabilities is that there is no way to immediately discern whether a particular attribute even applies to a repeated capability at all. All attributes use the same attribute accessors and all of these functions, as we've seen, require the user to pass in a repCapName parameter. If the attribute does not apply to a repeated capability, then the user passes in VI_NULL. Since the IVI-C attribute

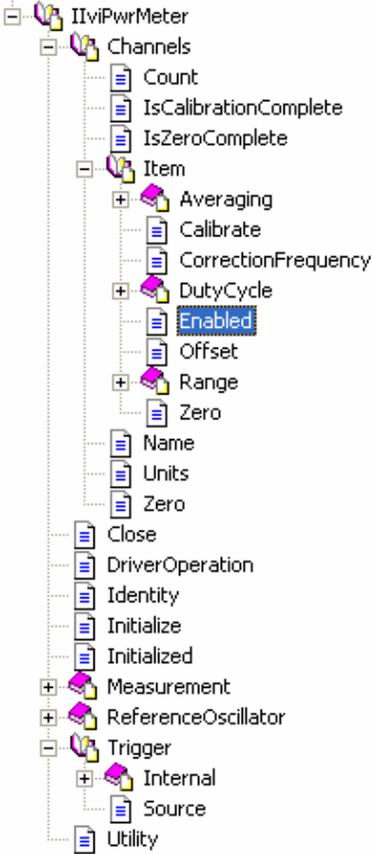
model is not type safe, there again is no way for the compiler to keep the user from making the mistake of supplying a repCapName when none applies or from omitting the repCapName when it is required. The following code demonstrates some of these kinds of common errors.

```
// Wrong – must indicate which trigger repeated capability
ag34401_SetAttributeViReal64(session, VI_NULL,
    AG34401_ATTR_TRIGGER_LEVEL, 0.45);

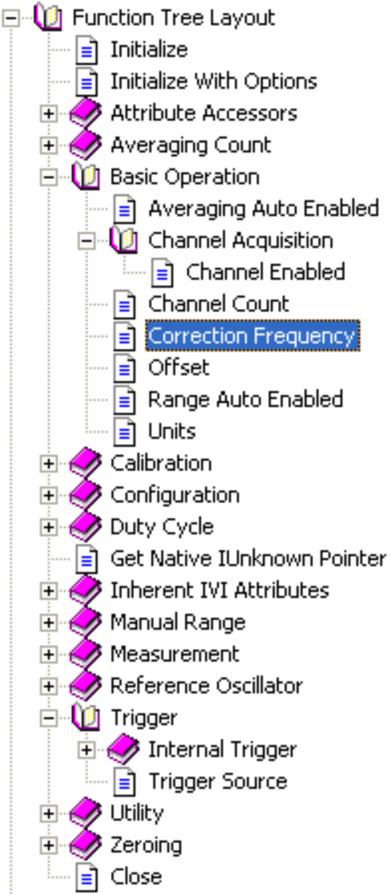
// Wrong – Range applies to the whole driver, not to Channel 1
ag34401_SetAttributeViReal64(session, "Channel 1",
    AG34401_ATTR_RANGE, 100);

// Wrong – Enabled is a property of output repeated capability,
// not the trigger repeated capability
ag34401_SetAttributeViBoolean(session, "Out1: Trig1",
    AG34401_ATTR_OUTPUT_ENABLED, VI_TRUE);
```

The above errors cannot occur with IVI-COM repeated capabilities because any property that applies to a repeated capability can be accessed only through a method that requires a repCapName. This also is evident from the IVI-COM hierarchy, whereas it is not clear at all from looking at the IVI-C hierarchy. Examine the following IviPwrMeter hierarchy for an IVI-COM driver. The attributes that apply to the Channel repeated capability, such as Enabled, are clearly grouped under the Channels->Item method. Other attributes, such as Trigger Source, apply to the driver as a whole.



The corresponding IVI-C hierarchy looks like the following:



In stark contrast to the IVI-COM hierarchy, the IVI-C hierarchy gives no indication as to which attributes apply to repeated capabilities and which do not. For instance, Correction Frequency applies to a specific Channel repeated capability while Units applies to the driver as a whole. Yet, these attributes both are defined as children of the patently nondescript Basic Operation group.

MULTIPLE INSTRUMENT CLASS SUPPORT

Some instruments have functionality that spans instrument classes. For instance, a switch-measure unit, such as the Agilent 34980A, contains both switching capabilities and DMM capabilities. IVI defines switches and DMMs in separate instrument class specifications –

lviSwch and lviDmm, respectively. IVI-COM drivers can seamlessly support multiple IVI instrument classes in a single driver. The Agilent 34980A IVI-COM driver capitalizes on this by exposing both the lviSwch and lviDmm interfaces. This allows the driver to be used interchangeably in a far wider variety of applications.

In contrast, IVI-C drivers only can support a single IVI instrument class. This is a fundamental limitation of the attribute model that IVI-C drivers employ. For example, the Bandwidth attribute from the lviSwch instrument class and the Trigger Delay attribute from the lviDmm class are assigned the same attribute ID value (1250005) in their respective IVI-C header files. In other words, they are defined as follows:

```
// In lviSwch.h
#define I VI SWTCH_ATTR_BANDWI DTH          1250005

// In lviDmm.h
#define I VI DMM_ATTR_TRI GGER_DELAY       1250005
```

The specific driver must support all attributes, irrespective of where they are defined, with a single set of attribute accessors such as the SetAttributeViReal64 function defined below:

```
Vi Status ag34980a_SetAttri buteVi Real 64(Vi Sessi on vi ,
      Vi ConstStri ng repCapName, Vi Attr attri buteId,
      Vi I ntReal 64 val ue);
```

It is impossible to implement the above function so that it can distinguish between calls that want to set the lviSwch Bandwidth attribute and those that want to set the lviDmm Trigger Delay attribute. From the perspective of the IVI-C driver, the following two functions calls look to be setting the exact same attribute:

```
ag34980a_SetAttri buteVi Real 64(sessi on, VI _NULL,
      I VI SWTCH_ATTR_BANDWI DTH, 10000);

ag34980a_SetAttri buteVi Real 64(sessi on, VI _NULL,
      I VI DMM_ATTR_TRI GGER_DELAY, 0.07);
```

There is no solution to this problem as it is simply a fundamental limitation of the IVI-C architecture. IVI-COM uses a completely different mechanism for dealing with attributes so it is not subject to this limitation. IVI-COM attributes are scoped by their parent interface, so they can implement any number of IVI instrument classes simultaneously.

MIXING CLASS-COMPLIANT AND INSTRUMENT-SPECIFIC CODE

The use of IVI-C drivers becomes noticeably more complex than IVI-COM when one considers applications that use IVI-defined functionality (also called *class-compliant* functionality) together with instrument-specific functionality. The class-compliant functions often cover only a portion of the functionality that a user might need. IVI recommends an approach that leverages the class-compliant functions as much as possible along with instrument-specific functions where required. To achieve this with IVI-C requires carefully managing separate instrument sessions for calls to the class driver versus calls to the specific driver. This is considerably less intuitive and more error prone than the equivalent code in IVI-COM. The example below shows the IVI-C code that is required to invoke a single class-compliant attribute and a single instrument-specific attribute. Even though two independent instrument sessions are in use, one for the class-driver and one for the specific driver, the real work still is still done in the specific driver.

```
Vi Sessi on cl assSessi on;
I vi Dmm_Ini tWi thOpti ons("MyDmm", VI_FALSE, VI_FALSE, VI_NULL,
    &cl assSessi on);

Vi Sessi on speci fi cSessi on;
I vi Dmm_GetSpeci fi cDri verCHandl e(cl assSessi on, &speci fi cSessi on);

I vi Dmm_SetAttri buteVi Real 64(cl assSessi on, VI_NULL,
    I VI DMM_ATTR_APERTURE_TIME, 0.35);

ag34401_SetAttri buteVi Real 64(speci fi cSessi on, VI_NULL,
    AG34401_ATTR_FOO, 1.0E6);
```

In the example above, `IVIDMM_ATTR_APERTURE_TIME` is a class-compliant attribute, while `AG34401_ATTR_FOO` attribute is an instrument-specific attribute that is not understood by the class driver. If a user is dealing with a class-compliant attribute or an instrument-specific attribute, they must do four things: 1) use the appropriate function prefix (`IviDmm` versus `ag34401`); 2) specify the correct session parameter (`classSession` or `specificSession`); 3) pass instrument-specific attributes to instrument-specific functions; and 4) pass class-compliant attributes to class-compliant functions. It's easy to mismanage these requirements and produce buggy applications. Unlike with IVI-COM, the compiler has no ability to detect these errors with IVI-C and they will not be discovered until runtime.

Consider the following four ways to set the AG34401_ATTR_FOO attribute. All of these examples look very similar and all compile without error, but only one of them will execute at runtime without generating an error.

```
// Correct – compiles and runs
ag34401_SetAttributeViReal64(specificSession, VI_NULL,
    AG34401_ATTR_FOO, 1.0E6);

// WRONG – Passing specific driver session to class driver
IviDmm_SetAttributeViReal64(specificSession, VI_NULL,
    AG34401_ATTR_FOO, 1.0E6);

// WRONG – Passing instr-specific attribute to class driver
IviDmm_SetAttributeViReal64(classSession, VI_NULL,
    AG34401_ATTR_FOO, 1.0E6);

// WRONG – Passing class driver session to specific driver
ag34401_SetAttributeViReal64(classSession, VI_NULL,
    AG34401_ATTR_FOO, 1.0E6);
```

An even more subtle error occurs if a user inadvertently passes an IVI-defined attribute to an instrument-specific function. In the example below, the user mistakenly passes the correct IVI-defined attribute (IVIDMM_ATTR_APERTURE_TIME) to the instrument-specific version of the attribute accessor.

```
// Wrong – Bypassing class driver, but compiles and runs!
ag34401_SetAttributeViReal64(specificSession, VI_NULL,
    IVIDMM_ATTR_APERTURE_TIME, 1.0E6);
```

The code above will compile and run without any errors, but the function call completely bypasses the class driver, which should be used for all IVI-defined attributes and functions. This can be confusing for the developer. Class drivers might be implementing important logic for state caching, range checking, coercion, simulation, or a variety of other features -- and all of that would be silently short-circuited by the benign-looking error above.

With IVI-COM drivers, the code is much simpler and eliminates the occurrence of the previously discussed errors. The IVI-COM driver is more type-safe because the compiler can ensure that instrument-specific attributes are mated with instrument-specific driver code and class-compliant attributes are mated with class-compliant driver code. The example below shows how simple it is to access both kinds of functionality with IVI-COM drivers.

```
Ivi SessionFactory fac = new Ivi SessionFactoryCl ass();  
I Ivi Dmm dmm = (I Ivi Dmm)fac. CreateDri ver("MyDmm");  
dmm. Ini ti al i ze("MyDmm", true, true, "");  
dmm. Advanced. ApertureTi me = 0. 15;  
I Agi l ent34401 agDmm = (I Agi l ent34401)dmm;  
agDmm. Foo = 15. 1E6;
```

No instrument-specific properties are accessible from the dmm variable, nor are any class-compliant properties accessible from the agDmm variable.

ERROR HANDLING

Error handling is an aspect of test application development that is all too often mismanaged. Both IVI-C drivers and IVI-COM drivers use error codes to indicate a failure condition. In IVI-C, these error codes are simple integer constants defined as ViStatus while in IVI-COM they are special HRESULT codes. The advantage of IVI-COM driver error handling is that users can leverage a standard exception handling mechanism built into COM and supported by all popular ADEs. When an error occurs in an IVI-COM driver, the driver returns an error object that contains much more information than does a simple IVI-C error code return value. The COM error object contains:

- The HRESULT error code.
- A string identifying the original source of the error.
- A GUID identifying the specific interface that caused the error.
- A driver-defined description of the error.
- Links to the page in the driver help file that explains the error.

The last item is particularly useful because it allows users to immediately navigate to the related help page whenever they encounter an error dialog in their application.

Not only is the IVI-COM error reporting richer, but it also reduces the code that developers must write. With IVI-C return codes, users must take care to always check the return value of each function. This adds considerable code to the application and introduces the opportunity for silent errors if a developer forgets to check a return value somewhere in the application. In contrast, IVI-COM driver errors are transformed into exceptions that get thrown on the client. This happens automatically, irrespective of whether the ADE in use is Visual Basic 6, C#, Visual C+, or any of numerous other environments. This means that the user need not have explicit error handling checks at

each function call and an error that occurs will be detected immediately at the initial point of failure.

BETTER DATA TYPES

Both IVI-COM and IVI-C drivers are written using a modest set of fundamental data types. This helps ensure drivers will be easily accessible in the widest range of ADEs. However, IVI-COM data types allow for the design of simpler and easier-to-use driver functions.

ARRAYS AND STRINGS

IVI-C drivers rely on traditional C-style arrays and strings. C-style arrays are simply pointers to a contiguous block of memory as are C-style strings. There is no way to carry array size or dimensional information in a C-style array or string. This information has to be manually provided when calling methods on IVI-C drivers. Consider the IVI-C Get Channel Name function defined in the IviScope instrument class:

```
IviScope_GetChannelName(ViSession vi,
                        ViInt32 index,
                        ViInt32 bufferSize,
                        ViChar name[]);
```

The bufferSize parameter specifies how many ViChar characters have been allocated in the name output string. This is important so that the method implementation knows how many characters can be written to the name output parameter. Users would invoke this method using code similar to the following:

```
ViChar name[50];
IviScope_GetChannelName(session, 2, 50, name);
```

When dealing with arrays, the situation often is more complex than the simple preceding example. Arrays often represent some sort of measurement data. With IVI-C drivers, the client application must specify how many array elements have been pre-allocated. The method implementation must pass back a different parameter indicating how many array elements were actually written to the output parameter. The Fetch Waveform method defined in the IviScope specification provides an excellent example of this situation:

```
IviScope_FetchWaveform (ViSession vi,
```

```
Vi ConstString channel ,  
Vi Int32 waveformSize,  
Vi Real 64 waveform[],  
Vi Int32 *actual Points,  
Vi Real 64 *initialX,  
Vi Real 64 *xIncrement);
```

The waveformSize parameter is an input that tells the method implementation how many elements the user allocated in the waveform array. The actualPoints parameter is an output that tells the user how many points were actually read from the instrument and written to the waveform array. It takes three parameters to manage a single array in IVI-C. If a method needed to deal with more than one array, and there are many real-world examples where this occurs, then there might need to be even more extraneous method parameters.

IVI-COM arrays and strings are fully self-describing. The BSTR data type is used for string parameters and intrinsically carries length information with it. The SAFEARRAY data type is used universally in IVI-COM drivers for array parameters. A single SAFEARRAY parameter contains element type information, size information, the number and size of array dimensions, as well as the array data itself. This allows IVI-COM methods such as Fetch Waveform to be simpler than their IVI-C counterparts, as can be seen from the IVI-COM definition for the Fetch Waveform function.

```
FetchWaveform(SAFEARRAY(double)* WaveformArray,  
double* InitialX,  
double* XIncrement);
```

A corollary to this discussion is the fact that the memory for all output parameters is allocated by the caller with IVI-C and allocated by the driver with IVI-COM. Having the IVI-COM driver function allocate memory makes sense because typically only the driver knows how much memory is needed to return data retrieved from the instrument. In contrast, an application using an IVI-C driver must know ahead of time how many elements will be returned before calling a function such as Fetch Waveform. Often an application does not know so it must literally "guess," call the Fetch Waveform function with the estimated size, and then check the actualPoints parameter to see how many elements were returned. If the actualPoints is equal to the waveformSize passed to the function, then the application must assume that there is more data to be retrieved. It must call the function again, merge the resulting array with the initial array, and start the sequence all over again. The required code is complex:

```

Vi Sessi on sessi on = // ...;
Vi Status status;
Vi Int32 actual Poi nts;
Vi Real 64 Inti al X;
Vi Real 64 XI ncrement;
Vi Real 64 waveformChunk[100];
Vi Real 64* waveform;
Vi Int32 waveformPoi nts = 0;
do
{
    // Retrieve no more than 100 elements
    status = I vi Scope_FetchWaveform(sessi on, "Channel 1", 100,
        waveformChunk, &actual Poi nts, &Inti al X, &XI ncrement);

    if (status == VI _SUCCESS)
    {
        Vi Int32 chunkSi ze = actual Poi nts * si zeof(Vi Real 64);
        Vi Int32 waveformSi ze =
            waveformPoi nts * si zeof(Vi Real 64);
        Vi Int32 newSi ze = chunkSi ze + waveformSi ze;
        Vi Real 64* tempWaveform = (Vi Real 64*)mal loc(newSi ze);
        memcpy(tempWaveform, waveform, waveformSi ze);
        memcpy(tempWaveform + waveformSi ze, waveformChunk,
            chunkSi ze);
        waveform = tempWaveform;
        waveformPoi nts = waveformPoi nts + actual Poi nts;
    }
} whi le ((actual Poi nts == 100) && (status == VI _SUCCESS));

```

The call to FetchWaveform retrieves 100-element chunks of data. Inside the If-statement, the code allocates a new array large enough to hold the data from previous calls to FetchWaveform and the data from the current chunk. The previous data and the current chunk are then copied into the waveform variable. If the actualPoints parameter returned from the function is 100, then the application must assume more data is available so FetchWaveform is called repeatedly. This continues until the driver signals that no more data is available by returning a value less than 100 in the actualPoints output parameter.

The equivalent code in a C++ IVI-COM client is mercifully simpler:

```

I I vi ScopePtr scope = // ...
SAFEARRAY* waveform = NULL;

```

```
double initialX;
double xIncrement;
scope->FetchWaveform(&waveform, &initialX, xIncrement);
```

Since the IVI-COM method is responsible for allocating all of the memory for the waveform data, the user can retrieve everything that is available in a single call to `FetchWaveform`.

OBJECT PARAMETERS

Driver functions often return related data as separate output parameters. Consider the following IVI-C function definition:

```
IviRFSigGen_QueryWaveformCapabilities(
    ViSession vi,
    ViInt32* MaxNumberWaveforms,
    ViInt32* WaveformQuantum,
    ViInt32* MinWaveformSize,
    ViInt32* MaxWaveformSize);
```

To use this function, the developer must declare four local variables to retrieve each of the output parameters:

```
ViSession session = //...
ViInt32 MaxNumberWaveforms;
ViInt32 WaveformQuantum;
ViInt32 MinWaveformSize;
ViInt32 MaxWaveformSize;

IviRFSigGen_QueryWaveformCapabilities(session,
    &MaxNumberWaveforms, &WaveformQuantum,
    &MinWaveformSize, &MaxWaveformSize);

if (MaxWaveformSize > 1000)
    // ... do something
```

It would be much more convenient and logical to deal with all of the outputs as a single unit, such as a C-style struct, particularly if the calling code was only interested in one or two of the output parameters. Unfortunately, IVI-C drivers cannot use structs in their public APIs. Structs are not a data type that is allowed by the IVI-C specifications.

On the other hand, IVI-COM drivers naturally group related items together in interfaces. Navigating between interfaces in an IVI-COM driver involves calling a property that returns an interface which contains a group of related methods and properties. Interfaces also are used to access independent instances of repeated capabilities, as shown earlier in the section on *Repeated Capabilities*. In a similar fashion, IVI-COM driver methods can return interfaces that bundle output parameters into a single object. Such a method definition would transform the above into the following:

```
QueryWaveformCapabilities(IIVI RFSi gGenWfmCapabilities** caps);
```

The IVI-COM client code for calling this function would be much simpler than the IVI-C code shown previously:

```
IIVI RFSi gGenWfmCapabilities caps = QueryWaveformCapabilities();  
if (caps.MaxWaveformSize > 1000)  
// ... do something
```

REMOTE ACCESS AND LOCATION TRANSPARENCY

One of the more widespread trends in application development today is the use of distributed systems. Applications might access devices, components, or resources that reside in a different application or on a different machine on the network. The LXI Consortium (www.lxistandard.org) is an example of a rapidly moving effort with a goal of making it easier to create distributed test systems. As the prominence of network-centric LXI devices grow, so too will the need for remote communication between software components. In this area, IVI-COM enjoys a compelling advantage over IVI-C.

As with any COM component, IVI-COM drivers can be accessed remotely without requiring any extra effort on the part of the driver developer and no additional coding on the part of the application programmer. The COM runtime takes care of all the details of shuttling method calls and their parameter data payload between applications running in different processes on the same computer, or between applications on different computers on the network. In fact, programs working with an IVI-COM driver can do so without any knowledge of where the driver is physically running. This feature is known as *location transparency*. Conventional Win32 DLLs, such as IVI-C drivers, have no such built-in remote communication mechanism. Consequently,

each IVI-C client program that needs to deal with remote resources must contend with low-level, inter-process communication facilities, such as sockets, named pipes, or memory mapped files.

PERFORMANCE MYTHS

One common misconception about IVI is that IVI-C drivers are intrinsically faster than IVI-COM drivers. In reality, this is not the case. A wide variety of factors influence the performance of an application that uses IVI drivers – far too many factors to categorically rate one type of driver more efficient than the other. The only fundamental difference between calling an IVI-COM driver method and an IVI-C driver method is that IVI-COM method calls are virtual. This simply means that the microprocessor must jump through one extra hardware address register. The added overhead is a handful of clock cycles which, on a modern processor, might add up to a nanosecond. If the application is interchangeable, then the IVI-C application needs to use a class driver. This involves two driver function calls per application function call. This overhead is actually larger than the single virtual method call that IVI-COM drivers require. In either case, such a miniscule amount of time is insignificant in an environment where I/O operations consuming tens or hundreds of milliseconds swamp out almost any other performance factor. Other factors such as the internal structure of the driver, the amount and type of data being passed, and the severity of virtual memory paging, all influence driver performance dramatically. Neither driver technology enjoys a decided advantage over the other in any of these areas.

CONCLUSION

This paper has attempted to present a comprehensive, detailed and objective explanation of the benefits IVI-COM drivers enjoy over IVI-C. Misinformation and confusion about IVI is widespread and, in the perpetual jockeying between proponents of one technology over the other, the details of the real issues often get lost. When you put both technologies under the microscope and examine the actual code end users must deal with, IVI-C begins to show its warts. There is no single fatal flaw in IVI-C drivers that render them decidedly inferior to IVI-COM – which is why all the small blemishes and annoyances rarely get the attention they should. But it is the collection of these IVI-C

limitations and irritations that, when viewed in the aggregate or when magnified by repeated exposure, culminate in a greatly diminished end-user experience.

Pacific MindWorks actively supports both IVI-COM and IVI-C in our services and with our Nimbus Driver Studio product, so we have no inherent interest in advocating one type of driver over the other. The goal here is simply to promote IVI as a driver standard by ensuring it always has its best foot forward. That best foot, in almost all circumstances, is unquestionably IVI-COM.

APPENDIX A – IVI-COM/IVI-C COMPARISON CHART

	IVI-COM	IVI-C
.NET Support	<ul style="list-style-type: none"> No manual coding is required to support .NET applications. Interchangeable .NET applications can easily be developed. 	<ul style="list-style-type: none"> Special .NET attributes and method declarations are required in client applications. Interchangeable .NET applications are not possible.
LabVIEW Support	<ul style="list-style-type: none"> Readability of code using Invoke Nodes and Property Nodes is superior to that of wrapped driver VIs. All driver functionality, methods and properties, is accessible from the Class Browser. Navigation of driver functionality is clearer and more convenient in the Class Browser than in the Function palette. Property access is identical to IVI-C. 	<ul style="list-style-type: none"> Palette icons for accessing driver functions are often ambiguous. Development of unique icons is burdensome. Without unique icons, application code is difficult to understand at a glance. Users must toggle between the Class Browser for accessing attributes and the Functions palette for accessing functions. Users must download and run a separate tool to generate wrapper VIs. Application code contains a mixture of wrapper VIs and Property Nodes.
LabWindows Support	<ul style="list-style-type: none"> Too difficult to be practical for most users. 	<ul style="list-style-type: none"> Simple and natively supported.

	IVI-COM	IVI-C
.NET Support	<ul style="list-style-type: none"> No manual coding is required to support .NET applications. Interchangeable .NET applications can easily be developed. 	<ul style="list-style-type: none"> Special .NET attributes and method declarations are required in client applications. Interchangeable .NET applications are not possible.
Interchangeability	<ul style="list-style-type: none"> Interchangeable applications require no 3rd party components. A single driver can easily support more than one IVI instrument class. Applications can easily access different IVI-defined class interfaces. Applications can mix calls to IVI-defined functions and instrument-specific functions easily and clearly. Errors are reported at compile time. 	<ul style="list-style-type: none"> Interchangeability requires proprietary class drivers, currently only available from a single supplier. A single driver can only support a single IVI instrument class. Mixing calls to IVI-defined functions and instrument-specific functions is error prone, and many common errors cannot be detected until runtime.
Deployment	<ul style="list-style-type: none"> Drivers are fully self-describing – only a single file needs to be imported by client applications. Client applications do not require knowledge of where the driver is installed. 	<ul style="list-style-type: none"> Drivers require a header file, a driver DLL, a library file, an .fp file, and a .sub file. Application code must explicitly specify the location of DLL, header, and library files.

	IVI-COM	IVI-C
.NET Support	<ul style="list-style-type: none"> No manual coding is required to support .NET applications. Interchangeable .NET applications can easily be developed. 	<ul style="list-style-type: none"> Special .NET attributes and method declarations are required in client applications. Interchangeable .NET applications are not possible.
Function Hierarchy	<ul style="list-style-type: none"> Functions are located next to their associated attributes. Function and attribute names are inherently scoped by their parent interface, allowing designers complete freedom in choosing meaningful names. 	<ul style="list-style-type: none"> Functions and attributes are located in two completely different hierarchies. All function and attribute names must be globally unique, often leading to confusing name collisions between instrument-specific and class-compliant names.
Attribute Access	<ul style="list-style-type: none"> Drivers provide simple and familiar "dot" syntax for accessing attributes. The compiler reports many types of common programming errors. Users enjoy a rich IntelliSense experience. Very little typing is required for clients to access an attribute. 	<ul style="list-style-type: none"> Attribute access rely upon special attribute accessor functions. Many common errors cannot be detected until runtime. IntelliSense is only marginally useful. Even simple attribute access operations require a significant amount of typing.

	IVI-COM	IVI-C
.NET Support	<ul style="list-style-type: none"> No manual coding is required to support .NET applications. Interchangeable .NET applications can easily be developed. 	<ul style="list-style-type: none"> Special .NET attributes and method declarations are required in client applications. Interchangeable .NET applications are not possible.
Repeated Capabilities	<ul style="list-style-type: none"> Familiar collection interfaces clearly indicate what functionality applies to repeated capabilities. Nested repeated capability access is intuitive. 	<ul style="list-style-type: none"> Users have no indication as to which attributes apply to a repeated capability and which apply to the driver as a whole. Accessing nested repeated capabilities require a special IVI-defined string-based syntax. User must take care to pass the same repeated capability name to each function and attribute call.
Error Handling	<ul style="list-style-type: none"> Driver error objects contain rich error information that is automatically reported in almost all popular ADEs. Errors in the driver automatically generate exceptions which cannot be ignored by the client. 	<ul style="list-style-type: none"> Errors are reported as function return codes, which can easily be ignored by the application. Application code must make explicit error checks after each driver call and then make a separate call into the driver to retrieve the error string.

	IVI-COM	IVI-C
.NET Support	<ul style="list-style-type: none"> No manual coding is required to support .NET applications. Interchangeable .NET applications can easily be developed. 	<ul style="list-style-type: none"> Special .NET attributes and method declarations are required in client applications. Interchangeable .NET applications are not possible.
Data Types	<ul style="list-style-type: none"> Arrays and strings are fully self-describing and require no additional parameters to carry size information. Memory is always allocated by the driver, so the client need not know ahead of time how much data will be returned. Related items can be returned as a single object. 	<ul style="list-style-type: none"> Input arrays require an extra size parameter to be passed to the driver. Output arrays require an extra buffer size parameter to be passed to the driver and an actual size parameter to be returned from the driver. Output strings require an extra string length parameter to be returned from the driver. Memory is always allocated by the client, so the application must know ahead of time how much data might be returned from the driver. Struct parameters are not allowed or supported, so related items must always appear as separate parameters.
Remote Access	<ul style="list-style-type: none"> Accessing a driver across a process or network requires no additional development work for the driver developer or the application programmer. 	<ul style="list-style-type: none"> No standard mechanism for cross-process or cross-machine access exists. Driver writers and application developers must resort to direct use of low-level Windows APIs.