



Understanding the Benefits of IVI

By

Kirk G. Fertitta, Pacific Mindworks

INTRODUCTION

The Interchangeable Virtual Instrument (IVI) Foundation was formed in 1998 with a charter to simplify test system development and maintenance by standardizing instrument driver technology. Working to that end, the IVI Foundation composed a series of specifications to facilitate the development of IVI instrument drivers. Many instrument manufacturers have a wide array of IVI drivers accompanying their instrument products and more manufacturers are moving to adopt IVI. In addition, the LXI (LAN eXtensions for Instruments) Consortium has selected IVI as the driver technology for its important instrument platform.

While much attention has been given to the benefits of instrument interchangeability, IVI drivers offer a number of benefits beyond interchangeability. This paper presents an objective examination of the benefits of IVI driver technology.

WHAT IS IVI?

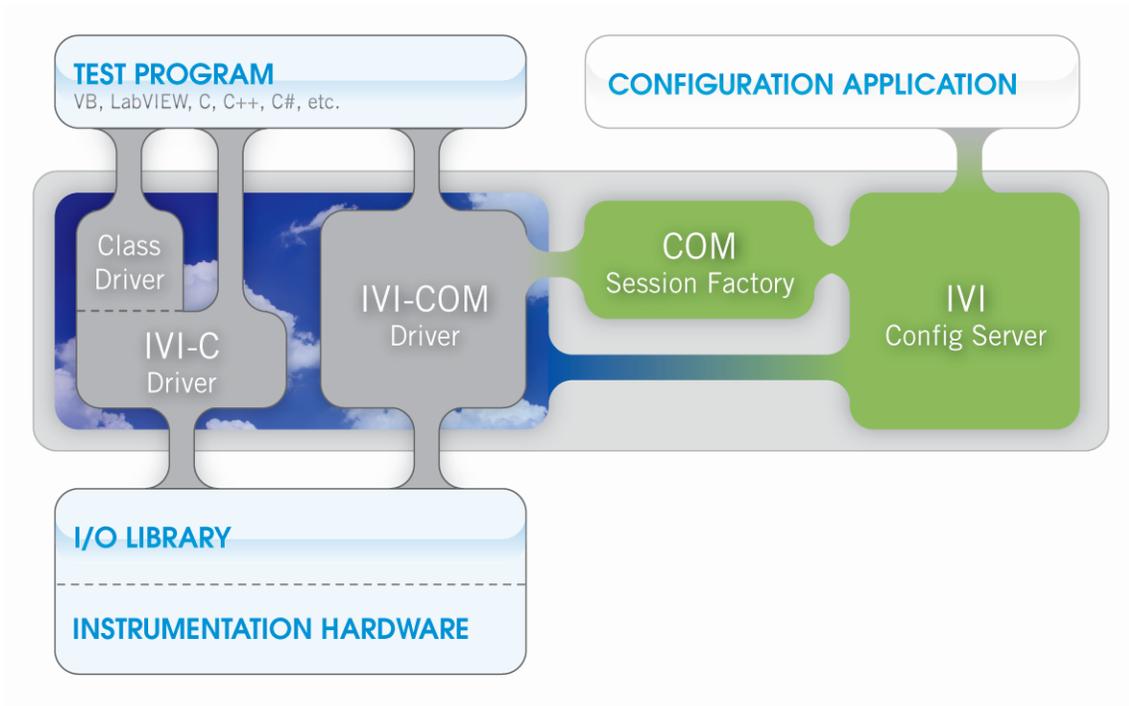
The IVI standard is grouped into three technology areas; 1) a set of instrument class specifications; 2) a collection of architecture specifications; and 3) a library of shared software components.

The **class specifications**, or instrument classes, define several types of traditional instruments such as DMMs, function generators, and spectrum analyzers. Each class specification precisely lays out the required functionality of an instrument class along with the detailed application programming interface (API) that class-compliant IVI drivers must expose.

The **IVI architecture specifications** cover a broad range of generic functionality that IVI drivers must support – even if they do not comply with any IVI-defined instrument class. These specifications prescribe everything from driver installer and help file requirements to standardized error reporting and API style. Indeed, the IVI architecture specifications are arguably the most valuable aspect of IVI.

The **IVI shared components** are a series of freely available and redistributable software modules developed and maintained by the IVI Foundation. These components are designed to ensure consistent implementations of important IVI driver features, such as configuration, error reporting, and multithreading. The IVI shared components must be installed on any system that will use IVI drivers – be it a driver developer workstation or an operator test station.

The following illustration shows how the various components of an IVI-enabled test system fit together.



IT'S NOT ALL ABOUT INTERCHANGEABILITY

Some of the instruments used in a particular test system may not have an associated IVI instrument class, making interchangeability impossible. For those instruments that do fall within an IVI instrument class, it also may be the case that the IVI-defined functionality covers only a small portion of the instrument's capability. In fact, as the instrument complexity goes up, the ability to interchange actually goes down. Modern spectrum analyzers, for example, typically have much more functionality than is defined in the IVI spectrum analyzer instrument class. Test programs that use functionality outside of the class specification are obviously not interchangeable.

Confronted with the preceding interchangeability challenges, some manufacturers and test programmers choose to not consider IVI any further. Some even go so far as to completely dismiss IVI if their instrument does not comply with an existing IVI-defined instrument class. These manufacturers and test programmers often make these decisions without a full understanding of everything IVI offers.

WHAT COMPLIANCE REALLY MEANS

For the manufacturer and the programmer alike, it's important to understand the meaning of IVI compliance. There are essentially two "degrees" or "levels" of IVI compliance – **basic compliance** and **instrument class compliance**. What many fail to recognize is that they can author a fully compliant IVI driver for an instrument that does not support any IVI-defined instrument class. IVI defines a number of standard functions and features that compliant drivers must support, as well as numerous other architectural requirements that drivers must meet. These capabilities and requirements are completely independent of whether the driver supports an IVI-defined instrument class. Drivers with this basic level of compliance can be advertised as being fully IVI compliant and can be used in development environments that support IVI drivers. Such a driver even meets the IVI driver requirements of the LXI Consortium for instruments that want to advertise LXI compliance. Moreover, drivers that support this basic level of compliance offer all of the same features and benefits as class-compliant IVI drivers, with the sole exception of interchangeability.

The second level of IVI compliance is IVI class-compliance. If an instrument falls into one of the IVI-defined instrument classes, such as DMM, oscilloscope, or function generator, then it is possible to create an IVI driver that supports the IVI-defined interfaces for the corresponding instrument class. Class-compliant drivers have the same features, benefits, and architectural requirements as non-class-compliant drivers – with the added benefits of interchangeability.

ONE DRIVER TO RULE THEM ALL

One of the principal challenges instrument manufacturers and software integrators face in the Test and Measurement industry is adapting their software to a large number of application development environments (ADEs). Users can choose from a variety of ADEs, such as Agilent VEE , MathWorks MATLAB,

Microsoft Visual Basic, or Microsoft C#, while other users need to work in National Instruments LabWindows or LabVIEW. Before IVI, the driver strategy that many instrument manufacturers pursued was to develop, distribute, and maintain separate drivers for each environment they wanted to target. LabVIEW customers preferred LabVIEW drivers, while Visual Basic customers required a Visual Basic driver. This led to duplicate work and increased overall cost. As a matter of practicality, manufacturers also would have to choose a subset of ADEs to support, and this would invariably alienate or frustrate that segment of customers working in one of the unsupported ADEs.

IVI drivers truly provide the ability to develop a single driver and provide a first-class user experience in virtually every popular ADE. **IVI drivers work seamlessly in a wide variety of ADEs**, including:

- Agilent VEE
- MathWorks MATLAB
- Microsoft Visual Basic 6.0
- Microsoft Visual Basic.NET
- Microsoft Visual C++
- Microsoft Visual C#
- Microsoft VBA environments (Excel, Word, PowerPoint, etc.)
- National Instruments LabVIEW
- National Instruments LabWindows/CVI

USER FAMILIARITY

One of the most effective ways to frustrate test programmers is to provide them with a dozen different ways to accomplish the same common programming tasks. By following the IVI standard, manufacturers provide the test programmer with a driver that is at least familiar, if not interchangeable. If the user has worked with any other IVI driver from any other vendor at any point in their career, then they will instantly know how to at least perform some basic tasks with any new IVI driver they encounter. They also will know how to access instrument-specific and other advanced features of the driver. The net result is a tremendous advantage in the “out-of-the-box” experience with an instrument manufacturer’s product.

Many simple tasks, often taken for granted, cause a great deal of test programmer confusion if not performed in a standard fashion. Driver instantiation, initialization, and shutdown are some of the most basic tasks every test programmer must perform. Every IVI driver provides the same functions for performing these basic operations. The specific behavior of these functions, with respect to resource management and instrument I/O, also is prescribed by IVI. If a test programmer can quickly create a simple program that communicates with a newly received instrument, then that will positively influence their initial overall satisfaction.

Configuration and installation are common tasks that test programmers need to be able to perform without having to learn something new. The IVI Configuration Store provides a single location where the user can discover what drivers are installed on their system. They can further discover a number of important details about their driver, such as the type of driver (IVI-COM or IVI-C), specific instrument

models it supports, and the IVI interfaces it exposes. Without a standardized driver, programmers might have to dig through any number of header files, registry settings, help documents, and readme files. Simply having a single, well-known place to set the instrument's I/O resource address provides a very real test programmer benefit.

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 provides a uniform mechanism for accessing them. Test programmers work with the same well-known methods and properties for discovering repeated capabilities, iterating through a list, accessing a specific repeated capability, and even applying a user-specified virtual name to selected repeated capabilities. Since all but one of the existing IVI instrument classes define repeated capabilities, it's important to provide a consistent, familiar and easy-to-use interface.

Another excellent example of a seemingly benign task that causes a surprising amount of programmer frustration is basic error reporting. Windows provides a dizzying array of options for reporting errors to application programs. One can use simple return codes or perhaps COM HRESULTs (both of which can easily be ignored by the test programmer's application). Alternatively, components can use the GetLastError/SetLastError idiom, which the test programmer only knows about from reading the documentation. These functions are thread-based and can easily produce erroneous results (errors within errors) if used improperly. Windows also offers a couple of exception types -- structured exceptions and C++ exceptions, which the user must be careful not to mix within an application. COM adds to this its own error-reporting mechanism via the IErrorInfo interface.

With drivers, the error reporting situation is further complicated by the need to support at least three sources of errors -- those coming from the driver, those coming from the I/O layer (such as VISA) and those coming from the instrument itself. IVI standardizes error reporting, so the programmer has a well-known set of functions for enabling error reporting, discovering if an error has occurred, and retrieving detailed error information. Without such standardization, the test programmer is left to contend with any number of unfamiliar error reporting schemes. IVI goes one step further by providing shared software components that assist driver developers in implementing features such as error reporting, thereby improving consistency across manufacturers.

TOOLS, TOOLS, TOOLS

What often makes a software standard compelling is the quality and availability of tools. With IVI, there is no shortage of developer and test programmer tools on the market. Even though IVI drivers are internally more complex and offer a broader array of features than other types of drivers, the robust tooling makes IVI drivers easier to develop than non-standard drivers.

Tooling is what makes standards thrive, and it is encouraging to survey some of the currently available tools with built-in IVI support.

- Agilent Connection Expert
- Agilent VEE

- MathWorks MATLAB
- National Instruments LabWindows/CVI
- National Instruments LabVIEW
- National Instruments Measurement Automation Explorer
- National Instruments TestStand
- National Instruments Signal Express
- Pacific MindWorks Nimbus
- Teradyne TestStudio

By furnishing an IVI driver with its instruments, the instrument manufacturers' products will instantly integrate into any of these environments, as well as a number of others. The usability and accessibility of their instrument will automatically improve – with no additional effort required on their part.

TRACKING THE STANDARDS

IVI drivers are coupled to a variety of disparate standards and Windows technologies. Without exception, these standards are moving targets – continually evolving and growing. Drivers must track all of these changes, irrespective of whether the drivers are IVI drivers or not. Most fundamentally, drivers must keep pace with changes in the Windows platform itself. Some of the Windows technologies on which a driver must rely include Windows Installer, Windows help, the .NET platform, the Windows API, and security. Important ADEs, such as Microsoft Visual Studio also are moving targets, and the IVI Foundation goes to great lengths to ensure IVI drivers operate well in such environments. All of these technologies require considerable expertise to master and a great deal of resources to track.

Windows Vista introduces a host of new challenges, as does 64-bit application development. Each standard and operating system must be carefully studied and followed if drivers are to remain robust, performant, and easy to use. Most instrument manufacturers and test programmers find this a daunting challenge and have neither the resources nor the desire to commit to tracking a large number of software technologies. This is where the IVI Foundation provides enormous value.

Many members of the IVI Foundation provide a wide array of software products to the Test and Measurement industry. Consequently, they must, for their own interests, carefully track the same set of software and hardware standards on which drivers rely. In order to guarantee their products continue to support IVI, members must ensure that IVI drivers evolve with these standards correctly and in a timely fashion. To that end, member companies bring considerable software talent to the IVI Foundation meetings to address how IVI should evolve to meet the changing software landscape. Much of the real detailed work required to incorporate new technology into IVI also is done outside of the IVI meetings, typically at the member company facilities using the company's own R&D resources. When the collective knowledge of all of these resources is harnessed at the IVI Foundation meetings, the group is very well empowered to keep IVI moving in the right direction. In a very real sense, all IVI users are directly leveraging the valuable software talent of numerous test and measurement industry leaders.

INDUSTRY MOMENTUM

Good standards are often built upon other good standards. Important associations within the test and measurement industry have decided upon IVI as their driver technology of choice. The LXI Consortium requires an IVI driver to be provided with any device claiming LXI compliance. The LXI Consortium recommends IVI-COM, although IVI-C is considered acceptable. As one of the most promising, active and dynamic standards bodies in the industry today, LXI lends a considerable amount of credence to IVI by relying on IVI for LXI's standard software interface.

The SCPI standard and the VXI *plug&play* standard are two mature and pervasive standards that are now part of the IVI Foundation. In order to facilitate long-term maintenance and to ensure consistency with future software standards, both of these organizations felt it was best to be acquired by the IVI Foundation.

All of the industry bodies that are turning to IVI for software standardization give testimony to the argument that IVI will continue to grow and that future industry organizations will look to IVI for driver technology.

DESIGN FLEXIBILITY

A common misperception about IVI drivers is that the design of the driver interface is too restrictive. Developers look at the IVI-defined interfaces for a particular instrument class and immediately conclude that IVI is not suitable for them because their device supports a broader array of functionality than IVI specifies or because their device models instrument behavior very differently than IVI does. In fact, IVI drivers are composed of two sets of functionality – class-compliant functionality defined by IVI and instrument-specific functionality defined by the instrument manufacturer. The class-compliant functionality is actually optional, so manufacturers who do not feel their instrument matches an IVI definition at all can simply choose to ignore the class specifications. The resulting driver can still be IVI compliant, as discussed earlier in the section entitled *What Compliance Really Means*.

The instrument-specific functionality in an IVI driver need not follow any prescribed set of functionality. Rather, driver developers have tremendous freedom in designing a driver interface that is intuitive for their particular customer base. Many IVI experts argue that the instrument-specific interfaces are the most important part of an IVI driver because they expose the unique features of the instrument – features which may have been the primary reason the customer selected the instrument in the first place. It is important to understand that a fully compliant IVI interface can easily be designed to accommodate virtually any way of abstracting the instrument's functionality.

In addition to giving the driver developer design flexibility, IVI provides a series of design guidelines to follow that ensure the driver works well in most ADEs. A great deal of effort has been invested by IVI Foundation members to explore and document subtle design requirements that, if ignored, would render many drivers unusable in certain environments. For example, one rule of IVI-COM interface design is that methods cannot have more than one output-only parameter. If multiple output parameters are needed, then they must be specified as input-output (two-way) parameters. The reason is that Visual Basic 6 will leak memory if a method has more than one output-only parameter. Memory leaks in applications are notoriously difficult to find and they often are even more difficult to fix once they have been located.

Without the IVI specifications to guide them in their designs, many driver developers would fall into this trap and most would have great difficulty understanding what was going on.

IVI drivers also are constructed in hierarchies of functions and attributes. This makes it easy for test programmers to navigate the available functionality of the instrument. These hierarchies are particularly important for instruments with large functional surface areas, such as spectrum analyzers and RF signal generators. The IVI specifications provide guidance on how to properly construct these hierarchies so that they are usable in a wide variety of ADEs.

EXTENDED DRIVER FEATURES

The IVI specifications describe four features of IVI drivers which provide unique capabilities beyond other driver technologies. These features are range checking, coercion recording, state caching, and simulation.

Simulation is by far the most important IVI driver feature. Consequently, the IVI specifications require that all IVI drivers implement simulation. When simulation is enabled, the IVI driver performs no instrument I/O. Rather, it synthesizes values for output parameters so that test programmers can begin developing and testing their applications without requiring an actual instrument. With long procurement cycles for many types of instruments, having physical access to an instrument is a luxury many test system developers do not always enjoy. The simulation support provided by IVI drivers is indispensable in such situations.

Range checking in IVI drivers validates input parameters against valid values accepted by the instrument. Often, range checking is performed within the instrument itself.

Coercion recording allows test programmer applications to query the driver for cases where parameters passed into a driver method or property had to be changed by the driver to values suitable for the instrument. For instance, a particular DMM may accept voltage range settings of 3, 30, and 300 Volts. When a test programmer's application attempts to set the voltage range to a value of 50, the driver may change the value to 300 to ensure the instrument is properly configured to perform the desired measurement. IVI drivers internally take note of these changes and store them for retrieval by test programmer's applications.

State caching is an optional feature of IVI drivers and can improve overall test application performance by eliminating redundant instrument I/O calls. When an application sets a property on an IVI driver to a new value, the driver stores this value in local memory. Subsequent queries for that property are serviced by directly accessing local memory, rather than by performing a time-consuming instrument I/O call. Similarly, subsequent calls to set the value of the property will not trigger an I/O operation unless the value supplied is different than the one stored in the driver's local memory cache.

Not only do the IVI specifications explain how these features should work, they also specify the functions that must be exposed so that the test programmer can control these driver behaviors. Standard functions

mean that test programmers have a common, well-defined mechanism for enabling state caching, range checking, and simulation and for reading coercion information from the driver. This improves the test programmer's overall comfort level and confidence in building their application.

BACKWARDS COMPATIBILITY WITH VXI PLUG-N-PLAY

IVI-C drivers are built on many of the same fundamental technologies as previous-generation VXI *plug&play* drivers. Users familiar with VXI *plug&play* (VXIpnp) drivers will find using IVI-C drivers very familiar and natural. IVI-C drivers use the same data types as VXIpnp drivers, such as `ViStatus`, `ViSession`, `ViInt32`, and `ViBoolean`. The details and hierarchy of IVI-C functions and attributes are represented in the same function panel (.fp) files and attribute information (.sub) files as VXIpnp drivers. IVI-C drivers also use the exact same attribute programming model as VXIpnp. Specifically, IVI-C drivers use functions such as `SetAttributeViInt32` along with a `#define`'d constant to set the values of driver attributes. This is a familiar idiom for programmers experienced with VXIpnp drivers. Error handling also builds upon the existing VXIpnp specifications.

The IVI Foundation even went so far as to break its own naming conventions in certain places in order to facilitate backwards compatibility with VXIpnp. IVI requires that driver functions start with an uppercase character. Yet, some IVI-C functions, such as `init`, `close`, and `reset`, all start with a lowercase character because that is how these functions were defined by the VXI *plug&play* standards.

CONCLUSION

While interchangeability is often understood to be IVI's chief benefit, IVI drivers offer test programmers many more benefits than interchangeability. Above all, instrument manufacturers can focus their energies on developing and maintaining a single driver that will provide a first-class user experience in a wide variety of development environments. The collective expertise of IVI Foundation member companies is continually applied to ensure IVI drivers stay in lock step with the ever-changing software and hardware landscape and that test programmers will enjoy a consistent and familiar experience with IVI drivers. An impressive array of IVI-enabled tools is available from an assortment of suppliers, and more IVI tools are on the way. As LXI instruments continue to emerge, IVI drivers will become more pervasive, establishing not only a core industry competency and comfort level in their use, but in fact creating a fundamental end-user expectation. Taken as a whole, IVI offers more to test programmers, driver developers, and instrument manufacturers than any other driver option.