The IVI Open-Architecture
Driver Specifications:
An Overview for System Designers

Application Note 1409-4

Agilent Technologies
Introduction

The IVI (interchangeable virtual instruments) Foundation is a consortium of companies working together to define a standard test-and-measurement instrument driver model that allows engineers to interchange instruments without changing their test-system software. IVI Foundation members believe that this driver standard will help electronics manufacturers save significant time and money. The driver standard allows companies to maximize their resources, minimize downtime of their critical production test systems, and extend the life of their software when test instruments become obsolete. The standard also can simplify instrument-programming tasks significantly.

The IVI specifications can be grouped into three categories, as shown in Figure 1. The IVI Open Architecture specifications establish how to create a driver, and they require several common capabilities. The IVI Class specifications define common application program interfaces (APIs) for instruments within certain common classes. The IVI Measurement and Stimulus Subsystem (MSS) specifications define system architectures that you can use to create systems that are practical to maintain, yet allow you to exchange instruments.

This paper describes the IVI Open Architecture layer of the specifications and how it provides system developers with a robust set of drivers. It steps through the various tasks you face as you develop your system and describes what the IVI Open Architecture provides to standardize and enhance each of the steps.

On the surface, it may seem that the core of a driver specification is the calls that control the instrument. Although these are critical, the standards that determine how the driver interface is designed establish the user experience with the driver. The standards that determine how the driver is installed and what system resources it requires establish where the driver can be used. The IVI Open Architecture standards include both design and environmental requirements for IVI drivers.

Figure 1. Respective benefits of various layers of IVI standards

<table>
<thead>
<tr>
<th>Layer</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVI Measurement and Stimulus Subsystems (3.10)</td>
<td>System architecture for full interchangeability</td>
</tr>
<tr>
<td>IVI Class Specifications (4.1, 4.2, 4.3, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10)</td>
<td>Syntactic interchangeability for similar instruments</td>
</tr>
<tr>
<td>IVI Open Architecture (3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.11, 3.12)</td>
<td>Driver ease of use and consistency</td>
</tr>
</tbody>
</table>
The IVI Open Architecture

The IVI Open Architecture specifications fall into three categories. There are things that must be implemented by every driver. These things are limited to functions and features that are independent of the instrument being controlled.

The IVI Open Architecture also includes "design patterns" that specify how to approach certain design problems. Design patterns are important because they make it easy for you to transfer learning and techniques from one driver to another.

The IVI Open Architecture also specifies conventions you must follow. Examples of these conventions are the approach for creating an interface hierarchy, how methods are named and when and how to use properties.

System development tasks and the IVI Open Architecture

To understand how the IVI Open Architecture simplifies your system development task, it is helpful to consider the basic tasks that you face when you design a system. From an IVI perspective, you need to:

1. **Specify the system**—When you initially specify the system and then design it, you need to know what system requirements are imposed by the driver, and what development environments are compatible with the driver.

2. **Install the drivers and get them working**—The most difficult part of getting a system to work is getting to the point where you can run a simple program that demonstrates communication with the test instrument. (This is commonly known as a "hello world" program, because frequently its only action is to print out "hello world" on the console, as proof that communication has been established). You can only do this once the driver is installed and registered with the development environment, and you have successfully written and compiled the program. When you are using drivers in a system environment, this is intrinsically complicated because software (drivers) from several vendors need to be incorporated into a single system.

3. **Learn and use the driver API to control the instruments**—Once you have integrated the instrument and driver into the system, the real task of writing code to make measurements begins. The IVI Open Architecture establishes a series of requirements that guarantee you will not need to approach each new driver as a new learning experience. All of the instrument interfaces are based on common style requirements and design patterns. In addition, each driver is required to implement a series of standard functions that simplify the development of the system software.

4. **Update the system over time**—Once you have deployed your system, you need to think about system maintenance and how you will introduce new instruments into the system.

The following sections describe what the IVI Open Architecture provides to standardize and enhance each of the steps described above.

**Specify the system**

Before you write the first line of code, you need to assemble the various components for the system. For a typical system, you will select a software development environment, then gather together compatible versions of instrument drivers, I/O hardware, I/O software, leveraged measurement routines, etc.

The interactions among these building blocks can become quite complex. In the case of the driver, you need to consider:

- What I/O hardware and software must be supplied by the system to the driver?
- What system architecture choices are imposed by the driver? (For instance, does the driver support multi-threading?)
- What development environments (and versions) will the driver work with or is it supported on?

The IVI 3.1 Driver Architecture Specification describes standard capabilities and functions that must be implemented by all drivers. IVI 3.1 requires that all drivers behave correctly in a multi-threaded environment. Therefore, two cooperating threads in an application can interact with the same instance of a driver without causing aberrant behavior.

IVI 3.1 requires that an IVI driver work the VISA I/O libraries for instruments that are connected via GPIB or VXI. This is a straightforward requirement because the VISA library is generally available for GPIB and VXI interfaces.

The I/O requirements are less clear for increasingly popular interfaces such as LAN and USB or for RS-232 connected devices. The difficulty arises because
the fundamental attraction of these interfaces is that they do not require special test-and-measurement software. As a result, a driver writer is unlikely to use a library like VISA that requires customers to purchase that library. Therefore, IVI 3.1 does not require VISA for instruments that use conventional computer I/O. Typical drivers for instruments with these interfaces make direct use of operating system services and do not require additional I/O software, although the IVI 3.1 specification only guarantees that the solution will not interfere with other drivers that are using the VISA library.

In addition to these I/O and environmental requirements, IVI 3.1 specifies that each driver provide documentation (in an IVI standard form) that describes what instruments it supports, the driver revision, specifications it complies with, I/O requirements, and other standard information.

Install the drivers and get them working

Once you have selected a suitable driver, your next challenge is to install the driver and its support libraries. Once you have completed this task, you typically would create a simple “hello world” program that instantiates the drivers and establishes communication with the instrument.

IVI 3.1 describes in detail how IVI drivers are installed on a system. For users, this means:

- You can add, remove, or update a driver without any concern that another driver will be impacted.
- You can easily locate and use drivers from various vendors. This means adding references or including directories to a development environment that works consistently for all drivers.
- All the components necessary for the proper operation of the driver are installed, and that this is done in a way that does not break other drivers or system components.
- The IVI Configuration Server (described below) is appropriately informed of the status of the new, updated, or removed driver so that any IVI-based application will be able to access the driver.
- Once a system has been configured, it can be duplicated appropriately while keeping correct versions of drivers.

IVI 3.1 also specifies how drivers use the standard version and manufacturer entries associated with all dynamic link libraries (DLLs). By defining how IVI drivers will use these fields, a user, or application, can easily track versions on a particular system, simplifying system maintenance and the task of cloning systems (for instance to add production lines).

Once you have configured the system and installed the appropriate drivers and components, your next step is to develop the system software. Before you can compile any code, the development environment must access the driver. If you are using Visual Studio, this means adding the appropriate references to the project. If you are using C, this means identifying appropriate “include” directories. Since IVI specifies the directories, the potentially complex job of integrating software from various manufacturers is straightforward.

Once you have added the appropriate references, you can instantiate (open) the driver. Although you can do this using conventional mechanisms, the IVI Open Architecture provides the IVI Component Factory to complete this task and provide additional capabilities. The IVI Component Factory works with the IVI Configuration Server to move some of the basic configuration out of the system program. Putting this information in a system configuration file allows you to update the system configuration without needing access to the system source code. This becomes extremely important during system maintenance. For instance, by keeping an instrument's I/O address in a system configuration file instead of the source code, it is possible to change the system to use an instrument at a different address (or even on a different physical interface) without having to touch the system source code.

Learn and use the driver API to control the instruments

Most of your system development time is spent on programming the instruments. The IVI Open Architecture specifications speed up this development by establishing requirements around the design of all drivers. These requirements guarantee consistency, making it easier to develop systems. Once you learn how to program the first driver, you can learn and work with additional drivers more easily.

In addition to providing consistency and easy learnability, the IVI Open Architecture requires that certain common functions be implemented, and be implemented identically by all drivers. This allows you to create services that work with all drivers, regardless of the vendor that provided them.

The following specifications all specify areas of consistency and commonality among IVI drivers:

- **IVI 3.1: Driver Architecture Specification** describes standard capabilities and functions that must be implemented by all drivers.
- **IVI 3.2: Inherent Capabilities Specification** describes functions that must be implemented by all drivers.
- **IVI 3.3: Standard Cross-Class Capabilities Specification** describes standard capabilities that must be followed by all drivers that implement a given function.
- **IVI 3.4: API Style Guide** describes style patterns that must be followed by all driver API designs.
The following briefly describes some of the behaviors and patterns specified in these documents:

**Interface hierarchy**—Even the simplest instruments have hundreds of properties and methods. It can be very confusing when all of these functions are presented in a single flat interface. IVI drivers implement a hierarchy of functions that group related capabilities together. The technique for creating the hierarchy is specified by IVI (in the case of IVI-COM, interface reference pointers are used to create the hierarchy). IVI also establishes the style for creating the hierarchy, so once you have learned how to configure or get measurements from one driver, the learning will transfer to other drivers.

**Interpretation of units**—Many of the parameters sent to an instrument have physical units associated with them (for instance Hertz). IVI specifies that instrument state variables should always be set in base physical units. For instance, a frequency would be set to 6 kilohertz by programming the state variable to 6000 (Hz), not 6 (kHz). IVI also specifies the actions an instrument should perform when rounding floating point values to do comparisons.

**Mnemonic generation rules**—The topic of style guidelines and acronym generation is never an easy one. Although this may seem like a mundane contribution, consistently labeling methods and properties is critical to a good experience, including being able to transfer your learnings across drivers. For instance, if designers had no guidance in how to generate mnemonics, a configuration function might be called SETUP, CONFIGURE, CONFIG, CNFGR, CFIGURE—you would never be able to predict what a particular driver required.

**Simulation**—IVI requires that every driver implement basic simulation capability. That is, it must be possible to instantiate the driver and use it even if the instrument is not physically present. In addition to the overall requirement, IVI specifies how simulation is enabled and disabled.

**Repeated capabilities**—It is fairly common for instruments to have certain capabilities that are replicated. For instance, an oscilloscope usually has multiple independent input channels and may also have multiple independent triggers. IVI specifies standard approaches for handling these repeated capabilities, including a procedure for selecting the capability to be programmed (that is, the “active” one) and a way to designate friendly names for these capabilities (for instance, instead of referring to the power supply channels as “1” and “2”, you may refer to them in ways meaningful to the system, perhaps as “BiasSupply” and “NegativeSupply”).

**Automatic parameter setting**—Most instruments have numerous parameters that can be set by the system or deferred to the instrument to select values to simplify programming. For instance, you can specify the measurement range of a multimeter, or you can set the instrument to autorange, which means the instrument selects the best measurement range. In either case, the system must be able to interrogate it to ascertain if the automatic behavior is currently enabled, and whether or not it is enabled, what the current value is. IVI establishes standard patterns for this process.

**State caching**—IVI encourages driver designers to cache instrument state variables where doing so improves the performance of the driver. However, state caching can cause drivers to behave oddly in certain circumstances. Therefore, IVI requires that all drivers provide a standard way to enable and disable state caching. Providing this consistently allows standard services to be created that can change the behavior of all drivers in the system and simplifies system debug.

**Group execute trigger**—Most test-and-measurement-oriented I/O interfaces provide a way to send a trigger to an instrument. All IVI drivers for such instruments are required to implement a standard API to send this trigger (referred to as a SoftwareTrigger in IVI) and control its behavior.

**Synchronization**—When you are optimizing the throughput of a system, it is important to be able to force the computer to wait for an instrument to complete its task, and similarly, to be able to allow parallelism between various instruments and the computer. For instance, it is common in an optimized system for several instruments to be given a task to work on in parallel.
IVI specifies standard functions for interrogating the instrument to ascertain if it has completed its tasks, and for blocking the computer until its tasks are completed.

This list includes only a few of the areas where IVI establishes design patterns and basic requirements for drivers. These standards establish a baseline you can count on. You also can count on consistency where the design is left to the driver developer. This structure leads to a system that is easy to use and learn, even when system components come from many sources.

**Update the system over time**

Although many test systems remain static once they are developed and deployed, in some circumstance it is important to be able to change the measurement hardware in a system. Hardware changes are common when obsolete instruments needs to be replaced, or when a given system needs to be able to work with different instruments.

The overall IVI architecture has multiple solutions to these challenges. The solution for a particular application will vary as you make trade-offs between the complexity of the initial system development, the complexity of exchanging instruments, and the degree to which the system must behave identically with the new instrument.

The solution to this challenge lies primarily in the IVI Class specifications and the IVI Measurement and Stimulus Subsystem (MSS) specification. The IVI Class specifications extend the requirements of the IVI Open Architecture by defining a common set of functions for selected classes of instruments. If you develop a system with an instrument and driver that conform to the class specification, it is possible to introduce a new instrument and its driver into the system in place of the original, as long as the system limits its accesses to the common set of functions. However, the resulting system will not produce the same results, unless the underlying instrumentation is identical. Even if the hardware is different, the basic system will still load and run. For instance, a new oscilloscope may produce a trace with a different number of points and different measurement accuracy. If the system software has accounted for these differences in advance, the system will continue to function. However, if the system is unable to deal with a longer (or shorter) trace size, it may require some maintenance; regardless, the difference in measurement accuracy will impact the behavior of the system.

The IVI Open Architecture specifications make this exchange possible because they specify a fixed way that the system locates and loads the driver that is independent of the vendor that produced it. If these infrastructure capabilities were not part of the standard, the class specifications would be of little use. For instance, the interchange could fail because the new driver might:

- require a different run-time environment due to a conflict with loaded libraries or the threading model
- not be able to work with the test systems I/O solution
- produce different error and status results

The IVI MSS specification spells out additional architectural solutions that allow instruments to be exchanged in a given system with little or no impact on the system behavior. Because of variations in drivers and instruments, there is not a driver-based solution that reliably accounts for all of the behavior differences when a new instrument is introduced into a system. IVI MSS specifies how particular instrument software should be incorporated into measurement servers or stimulus servers that allow you to update the usage of the instrument as well as the instrument driver when a system is updated. This enables a system to continue to function virtually identically after an instrument exchange, but at the cost of greater system development time.

**IVI-COM and IVI-C**

IVI 3.1 defines two fundamental interface types, IVI-COM and IVI-C. IVI-COM drivers are based on COM (Component Object Model) technology from Microsoft Corp. Although a detailed description of COM is beyond the scope of this paper, the key characteristics that are important to IVI are:

1. COM objects can be used in most popular software development environments. Including Microsoft® Visual Basic, Visual C++, C#, and National Instruments LabVIEW.

2. COM provides a mechanism for dynamically loading on object module. This is especially important to IVI users who need to interchange instruments. In this case, the COM infrastructure is used to dynamically load a particular driver that provides a given interface. For instance, a run-time decision can be made to load a driver to control either of two oscilloscopes that each implement the same interface.

3. COM objects can incorporate (and in the case of IVI, do provide) syntactic and help information about their APIs. This information can be presented by the software development environment to simplify your job in creating programs.

IVI-C drivers are based on the existing VXIplug&play specifications. Although it is straightforward to apply the class specifications to VXIplug&play drivers, the basic DLL technology upon which it is built does not intrinsically support syntactic interchangeability.
Therefore, an additional intermediate driver must be called by the application program. This intermediate driver in turn calls the particular instrument driver to accomplish the function. The advantage of IVI-C drivers is that they work directly in National Instruments’ LabWindows and LabVIEW as well as conventional ANSI C programming environments. IVI-C drivers also can use the intermediate driver to perform other functions such as enhanced simulation and monitoring. You need to be aware that instrument vendors, in general, will not provide this intermediate driver; you must acquire it from a software vendor.

If you plan to use IVI drivers, you will choose between IVI-C and IVI-COM based on the availability of drivers, the programming environment you select, and the variety of features you need.

**Summary**

The IVI Open Architecture standards provide many benefits compared to existing standards and ad hoc driver solutions by:

- Specifically addressing the tasks a system designer faces
- Simplifying learning a new driver, because all drivers share common abilities and are designed with consistent patterns
- Having the support of all the key players in the test-and-measurement industry, including hardware and software companies

For more information on the IVI Standards see http://www.ivifoundation.org.

**Glossary**

**ANSI C**—A programming language standardized by a subcommittee of the American National Standards Institute

**API**—application programming interface; use the API to access functionality embedded in classes.

**Class**—a software module that provides methods and properties that you can incorporate into your code by calling its properties, methods and interfaces.

**DLL**—dynamic link library

**GPIB**—general purpose interface bus—used for instrument control

**IVI**—interchangeable virtual instruments; a standard instrument driver model allowing you to swap instruments without changing software. Learn more at http://www.ivifoundation.org/.

**IVI-C**—IVI-C drivers are based on the existing VXIplug&play specifications.

**IVI-COM**—IVI-COM drivers are based on Component Object Model technology from Microsoft Corp.

**LAN**—local area network; can be used for connecting instruments to PCs in test and measurement applications.

**MSS**—measurement and stimulus subsystem

**RS-232**—interface for connecting instruments to PCs in test and measurement applications. Although RS-232 is a low-cost solution, its low baud rate and connection limitations make it too slow and cumbersome for many of today’s measurement needs.

**USB**—universal serial bus; a standard bus on today’s PCs, can be used for connecting instruments to PCs in test and measurement applications

**VISA**—virtual instrument software architecture. This is the designation used for the instrument I/O library standard developed by the VXIplug&play foundation

**Visual Studio**—integrated software development environment from Microsoft that supports development in Visual Basic, C, C++ and C# as well as other languages

**VXI**—a hardware standard for test and measurement instruments on plug-in cards

**VXIplug&play**—a set of software standards that promotes interoperability between instruments made by different manufacturers. Before IVI, most instrument drivers were based on VXIplug&play standards. Learn more at http://www.vxipnp.org.

**Related Agilent literature**

<table>
<thead>
<tr>
<th>Literature type</th>
<th>Title</th>
<th>Publication number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brochure</td>
<td>Agilent’s Connectivity Products and Solutions</td>
<td>5988-6527</td>
</tr>
<tr>
<td>Data sheet</td>
<td>Agilent W1138A Connectivity Suite</td>
<td>5988-5756EN</td>
</tr>
<tr>
<td>Data sheet</td>
<td>Agilent W1150 Agilent Developer Network</td>
<td>5988-6596EN</td>
</tr>
<tr>
<td>Application note</td>
<td>Simplified PC Connections for GPIB Instruments</td>
<td>5988-5897EN</td>
</tr>
</tbody>
</table>

Download this literature from www.agilent.com/find/ADN
Agilent T&M Software and Connectivity
Agilent’s Test and Measurement software and connectivity products, solutions and developer network allows you to take time out of connecting your instruments to your computer with tools based on PC standards, so you can focus on your tasks, not on your connections. Visit www.agilent.com/find/connectivity for more information.

By internet, phone, or fax, get assistance with all your test & measurement needs
Online assistance: www.agilent.com/find/assist

Phone or Fax

United States:
(tel) 800 452 4844
Canada:
(tel) 877 894 4414
(fax) (905) 282 6495
China:
(tel) 800 810 0189
(fax) 800 820 2816
Europe:
(tel) (31 20) 547 2323
(fax) (31 20) 547 2390
Japan:
(tel) (81) 426 56 7832
(fax) (81) 426 56 7840
Korea:
(tel) (82 2) 2004 5004
(fax) (82 2) 2004 5115
Latin America:
(tel) (305) 269 7500
(fax) (305) 269 7599
Taiwan:
(tel) 0800 047 866
(fax) 0800 286 331
Other Asia Pacific Countries:
(tel) (65) 6375 8100
(fax) (65) 6836 0252
(e-mail) tm_asia@agilent.com

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.
Product specifications and descriptions in this document subject to change without notice.

© Agilent Technologies, Inc. 2002
Printed in the USA October 25, 2002
5988-7939EN