The IVI Driver Standards

By Joe Mueller, President, IVI Foundation

The IVI Foundation exists to define standards that simplify programming test instruments. Although the IVI Foundation is responsible for a large breadth of standards, some of the most important for constructing systems are the IVI driver standards. The driver standards describe the software package that should accompany an instrument to provide a programmatic interface to the instruments.

The driver standards fall into three main pieces. These pieces are:

- The IVI Architecture standards. These are numbered 3.x. The architecture standards describe things that are common to all drivers independent of the specific instrument that the driver supports.
- The IVI Class standards. The class standards describe capabilities of a certain class of products. Thus, a driver that complies with the class standards implements a known set of functions for a particular class of instrument. For instance, a common set of functions for all multimeters.
- The IVI common components. Although the common components are delivered to customers directly by the IVI foundation, they are also defined by the IVI driver standards. The common components provide common software components that are necessary for the drivers, but are not specific to anyone driver. By providing them directly from the foundation, they ensure that the drivers themselves can work together without any conflicting vendor-specific pieces.

1 Importance of Driver Standards

The driver standards are particularly important because they define the software that enables a system integrator to combine instruments on a variety of interfaces from a variety of vendors. For instance, the driver standards enable a system integrator to use the same software interface to instruments regardless of if the instruments are on GPIB, the network device, or PXI interface. This is the reason that LXI requires an IVI driver be available with LXI instruments.

To understand the importance of this, it is useful to consider SCPI, another standard from the IVI Foundation. Most GPIB instruments follow the SCPI standard. SCPI defines a standard set of strings and syntax to control an instrument. SCPI was initially targeted at GPIB, but has been applied to many other interfaces as well. However, drivers provide some key benefits not available with SCPI alone.

1.1 SCPI Communication

The SCPI standard was originally written to provide a standard way of sending commands to GPIB instruments. However, as instruments migrated to computer standard interfaces such as USB and the network, instrument manufacturers have also provided SCPI on those interfaces.

The IVI Foundation owns the SCPI standards, as well as standards and common components for sending SCPI commands over computer standard interfaces. One recent notable effort is the HiSLIP standard which specifies an efficient way to send SCPI commands to network instruments. HiSLIP is one of the optional protocols specified in LXI.

There are two key issues with SCPI communication. First, SCPI is somewhat awkward for the programmer because all operations and results have to be expressed in strings that are sent to and from the instrument. So the SCPI user does not use familiar programming conventions such as calling subroutines to complete an action. For instance, if the programmer has a simple syntax error in their program, such as a misspelled command name, this is not detected until the program is run and the string is sent to the instrument, then the instrument detects that the string is wrong.

For the system integrator, this is further complicated by the difficulty of determining the error. To determine the error, the programmer needs to go to the physical instrument, either via the front panel, or by writing another program to determine from the instrument what the source of error was. If the programmer is working with an instrument that is located remotely, or if the instrument itself is not present because the programmer is developing the program without the instrument being available, this becomes extremely difficult.

In contrast, when using a driver, any syntax error will be immediately detected in the software development environment. The software development environment also provides the developer with various types of help when composing the program, such as Microsoft IntelliSense that prompts the programmer for function and parameter names and provides explanations of the various operations. In addition, since parameters and settings are passed to the driver functions using conventional data types, the programmer does not need to format them into strings to send them, or build them back into numbers when they come back from the instrument. In addition, when using a driver, the instrument does not need to be present when the program is being developed. The developer places the driver into a simulation mode, and then it works without requiring a connection to the instrument.

The second issue with SCPI communication is that it is not practical for many instruments. Modular instruments (such as PXI, PXIe, and VXI) do not have a natural way to provide a SCPI interface. The instrument does not have a microprocessor that is capable of receiving and interpreting SCPI commands, instead the main computer directly controls the hardware. More importantly, the parsing of SCPI commands and formatting the results into numbers and strings for the test program results in performance penalties that are inappropriate for high-speed instruments. Even instruments on GPIB and network interfaces occasionally have performance limitations due to the processing of SCPI commands.

Because of these limitations, the LXI specification does not require instruments to implement SCPI. Instead, LXI specifies that the instrument provide an IVI driver. By specifying an IVI driver, the LXI instrument is free to implement SCPI if it chooses. However many LXI instruments instead choose higher speed network protocols, they then expose the instrument capabilities to the programmer with a driver.

1.2 Driver Standards

Since the inception of VXI, most modular instruments have focused on programmatic interfaces as the primary instrument interface. Shortly after the introduction of the VXI specification, it became clear that a programmatic API standard was necessary. Drivers were needed to provide a faster interface than SCPI, as well as a higher-level, convenient interface for the programmer. However, without standardization customers struggled to create systems since drivers from different vendors were inconsistent and sometimes incompatible (for instance, each could use a different IO library).

The initial driver specifications were the VXI*plug&play* standards (also currently a part of the IVI Foundation). They define basic interfaces to the instrument in C and NI LabVIEW. Based on the VXI*plug&play* standards, any customer programming with C or LabVIEW was able to access VXI instruments consistently, regardless of the instrument manufacturer. Also, using these standards, the software could communicate with instruments in any VXI chassis.

Shortly after the introduction of the VXI*plug&play* standards, it became clear that SCPI based instruments such as GPIB instruments also needed drivers to provide customers with a consistent interface to instrument systems that included both GPIB and VXI instruments. In addition, the SCPI instruments needed to resolve the customer difficulties with SCPI outlined above.

Therefore, many instrument vendors began to produce VXI*plug&play* drivers for non-VXI SCPI instruments. These drivers provided customers with a high level interface and relieved many of the challenges of dealing with SCPI. It also provided the customer with a single programming interface for both SCPI and VXI instruments, considerably simplifying system integration.

After a few years experience with the VXI*plug&play* standards, some shortcomings became apparent that were ultimately resolved by the IVI standards. These are:

- The VXI*plug&play* standards only address C programming. Many modern programming environments, including C++, Visual Basic, and C# provide more sophisticated capabilities that can not be accessed by the VXI*plug&play* C libraries.
- The VXI*plug&play* standards specify very little about the programmatic interface itself. So the individual functions to program the instruments were inconsistent.

The IVI standards build on the VXI*plug&play* standards. By adding:

- additional features, such as an instrument simulation mode that permits developing software without the instrument being present

- the capability to work in new environments (such as COM) and provide the features built-in to those environments
- common functions provided by every driver
- class standards that specify a set of functions to be implemented by drivers of frequently used instrument classes (such a multimeters)

The following sections describe how the IVI standards provide these capabilities.

2 IVI Standards

As described above, the IVI standards fall into three groups: architecture standards, common components, and class standards.

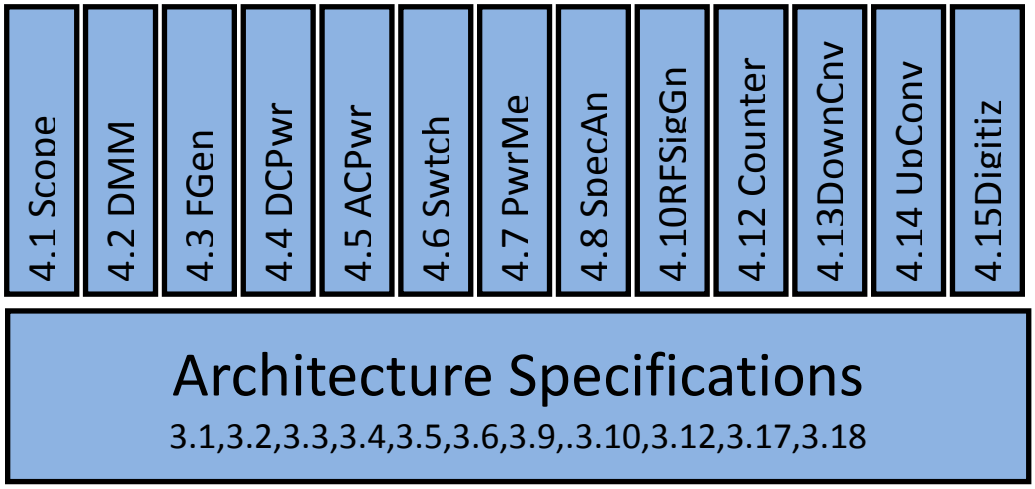

Figure 1 The various IVI Driver Specifications

The Architecture standards describe things that are common to all drivers. This includes:

- Installation
- Mnemonic formation
- Common commands
- Usage of the common components
- How to instantiate, initialize, and close the driver
- Common programming approaches (for instance setting up instrument triggers)
- Base technologies of C, COM, or .NET
- Basic testing requirements
- Functions to access underlying SCPI interfaces, where they are available

The architecture specifications describe three types of IVI drivers: IVI-C, IVI-COM and IVI.NET. The IVI-C and IVI-COM architectures were part of the initial IVI standards. The IVI .NET architecture was added in 2010.

The IVI Class standards are optional standards that specify the API for specific kinds of instruments. The instruments defined are:

- DC power supply
- AC power supply
- DMM
- Function generator
- Oscilloscope
- Power meter
- RF signal generator
- Spectrum analyzer
- Switch
- Upconverter
- Downconverter
- Digitizer
- Counter/timer

The IVI class specifications define calls that should be implemented by instruments of these kinds. These provide the programmer with the additional advantage of not needing to program to two instruments from different (or even the same) vendor that define the instrument calls differently.

3 Driver Types

The following sections briefly mention the three types of driver that are defined by IVI and how some of the basic driver capabilities are accessed in each.

3.1 IVI-C Drivers

IVI-C drivers are based on the VXI*plug&play* standards and provide a conventional C DLL. They are useful for many programming environments other than C, since many programming languages provide a mechanism to call C DLLs. The C DLL provides a driver that can access the capabilities of the language and it does not require much more than the basic ability to call a DLL.

Although callable from almost any programming environment, C DLL's do not provide many of the capabilities provided by the COM and .NET drivers. One common example of this is the use of attributes. IVI defines attributes for setting most of the values with an instrument. Most object-oriented programming environments permit the program to use attributes by making simple assignments. For instance:

```
myDriver.Resolution = 0.001;  // set the instruments resolution parameter
```

The C programming language does not provide a way to set an instrument parameter by using a simple assignment. As a result the IVI-C programming standards also define a mechanism for setting and

reading driver attributes. Thus, in addition to the functions that control the general operation of the driver, IVI-C drivers can read or set attributes using syntax similar to:

```
Ag34401A_SetAttributeReal64(vi, AG34401A_ATTR_RESOLUTION, 0.001);
```

This example also illustrates the C naming convention. Every function in a C program must be uniquely named. Although a straightforward limitation, this results in some complications. For instance, every driver has the need to set a Real64 attribute (as shown above), so each function name must be preceded by the name of the instrument to ensure that the call is unique. This is not just a requirement for functions like SetAttribute, every driver has to assure that its function names are unique so every call has this prefix. For instance, every call in the Agilent 34401A driver has a prefix of "Ag34401A_".

Another complexity of dealing with C programming environments is the allocation of memory for arrays. In the C programming environment, the management of memory is the responsibility of the client because the driver itself may never have the opportunity to free any allocated memory. Therefore, in IVI-C the memory allocation is always the responsibility of the client. This complicates simple calls that need to return an array of results. For instance, to get an array of readings from a DMM, the IVI-C interface requires:

```
ViReal64[] Readings;
ViInt32 ReadingCount;
Readings = (ViReal64*) malloc (100*sizeof(ViReal64));
Agilent34401A_MeasurementFetchMultiPoint(vi, 1000,
                                         100, Readings, ReadingCount);
```

There are several general capabilities that are not necessarily specific to drivers, but are needed by the IVI drivers. For instance, the common components define numeric constants to represent infinity. One of the common components provided by IVI is C components that provide these definitions that are included in the COM and .NET environments, but need a definition for IVI-C.

In summary, the IVI-C drivers provide a very portable C DLL that can be used in many different programming environments. However, since the C language defines only the most basic capabilities, the interface itself is not quite as convenient to use as the COM or .NET environments.

3.2 IVI-COM Drivers

IVI-COM drivers standardize how to provide a driver as a COM object. COM is a binary standard that permits object oriented interfaces to be provided independently of the programming environment they were created in. COM objects are very conveniently accessed from the Microsoft .NET languages (such as C# and Visual Basic). However COM is also conveniently accessible from many non-Microsoft environments including MathWorks MATLAB, National Instruments LabVIEW and Agilent VEE.

As mentioned above, the COM driver permits assigning values directly to instrument attributes. For instance:

```
myDriver.Resolution = 0.001;
```

In addition, calls to common functions like Initialize do not need to be prefixed with the instrument name because the specific driver type is specified when it is initialized. For instance, the following sequence can be used to set Initialize and set the resolution using an IVI-COM driver from C#:

```
Agilent34401A dmm = new Agilent34401A();

Dmm.Initialize("GPIB0::1", true, true, "");
Dmm.Resolution = 0.001;
```

Note here that the dmm is clearly an Agilent 34401A, however once the driver is instantiated with the "new" keyword, each call to the driver does not need to carry a prefix as it does in C.

Another advantage of the COM environment is its' memory management capabilities. In COM, the client can allocate a very simple structures that is subsequently filled in by the driver. For instance, the following is all that is necessary to return an array of readings IVI-COM:

```
Double[] Readings = {0};
Dmm.Measurement.FetchMultipoint(1000, Readings);
```

The one awkward aspect of this is that IVI-COM requires that an array be initially passed into the FetchMultipoint function. This is because some of the early IVI-COM programming environments did not permit an empty pointer to be passed to a function.

In summary, the IVI-COM environment provides a convenient interface that includes several programming conveniences such as built-in fundamental data types, properties, and exception handling.

3.3 IVI .NET drivers

IVI .NET is similar to COM in that it also provides an object oriented interface that is callable from many different programming environments. IVI.NET primarily differs from IVI-COM due to basic technology advances in .NET.

The .NET environment has several simplifications compared to the COM environment. Some of the key differences for driver applications are:

- The .NET environment provides a rich type system
- .NET defines names within a scope.

Neither COM nor C permit the driver to return complex data types because the software environments where COM and C drivers may be used do not guarantee that anything besides the most basic types are supported (scalar types and arrays of them). IVI.NET drivers require the full .NET common language runtime (CLR), therefore rich types including classes and objects may be returned from IVI .NET drivers.

One interesting example is accessing an enumerated value. Consider setting the function in a DMM. In each of the programming environments, the same basic operation is executed. In IVI-C:

```
Ag34401A_SetAttributeInt32(vi, AG34401_ATTR_FUNCTION,
                                  AG34401_VAL_DC_VOLTS);
```

You can see here that:

- C Requires the “Ag34401A” prefix on the SetAttribute function, and the value used to identify the measurement function attribute, as well as the value that specifies DC Volts.
- The values “AG34401_ATTR_FUNCTION” and “AG34401_VAL_DC_VOLTS” are just integers that were defined in the IVI-C include file. Therefore, the C compiler will accept any parameters here that are integers and so the compiler does not provide any consistency checking, for instance, verifying the DC volts value really corresponds to the function attribute.

The same operation from C# interfacing with a COM object looks like this:

```
vi.Function = Agilent34401AFunctionEnum.Agilent34401AFunctionDcVolts;
```

Here we see that COM significantly simplifies this call. However, the name scoping still requires the repetition of the “Agilent34401” prefix.

.NET further simplifies the call as follows:

```
vi.Function = Function.DcVolts;
```

In this example, since the compiler knows it is making as assignment to an attribute of type “Function”, the assignment does not need to include the full context.

4 Summary

Drivers are extremely important to provide with instruments. They provide a single style of interface for system integrators with a higher value than a SCPI interface. Drivers provide a more abstract interface that is easier use that is independent of the underlying physical interface to the instrument.