



Getting Started with IVI Drivers

**Guide to Using IVI.Net Drivers
with Visual C# and Visual Basic
.NET**

**Version 1.0
Aug 8, 2016**

**© Copyright IVI Foundation, 2016
All rights reserved**

The IVI Foundation has full copyright privileges of all versions of the IVI Getting Started Guide. For persons wishing to reference portions of the guide in their own written work, standard copyright protection and usage applies. This includes providing a reference to the guide within the written work. Likewise, it needs to be apparent what content was taken from the guide. A recommended method in which to do this is by using a different font in italics to signify the copyrighted material.

Table of Contents

Chapter 1	Introduction.....	4
	Purpose.....	4
	Why Use an Instrument Driver?	4
	Why IVI?	5
	Why Use and IVI Driver?	7
	Flavors of IVI Drivers.....	8
	VISA I/O Library.....	8
	Shared Components.....	9
	Download and Install Drivers.....	9
	Familiarizing Yourself with the Driver.....	10
	Examples.....	11
Chapter 2	Using IVI.NET with Visual C# and Visual Basic .NET.....	12
	The Environment.....	12
	Example Requirements.....	12
	Download and Install river.....	12
	Create a New Project and Reference the Driver.....	13
	Create an Instance of the Driver.....	14
	Use Object Browser for Help.....	15
	Configure the Instrument	17
	Display the Results.....	18
	Check for Errors.....	18
	Close the Session.....	19
	Build and Run the Application.....	20
	Visual Basic.NET Example Code.....	21
	Further Information.....	23
Chapter 3	IVI-COM vs IVI.NET.....	24
	Overview.....	24
	IVI Driver Source Code.....	25
	Side-by-side Deployment of IVI Drivers.....	25
	PrecisionTimeSpan.....	26
	IVI.NET's Richer Type System.....	26
	Initializing the Driver.....	26
	Error Handling.....	27
	Events from the Driver.....	27
	Syntax for Enumerations and Repeated Capabilities.....	28



Chapter 1

Introduction

•••

Purpose

Welcome to ***Getting Started with IVI Drivers: Your Guide to Using IVI.NET Drivers with Visual C# and Visual Basic .NET***. This guide introduces key concepts about IVI drivers and shows you how to create a short Visual Studio project using an IVI.NET Driver. The guide is part of the IVI Foundation's series of guides, ***Getting Started with IVI Drivers***.

Getting Started with IVI Drivers is intended for individuals who write and run programs to control test-and-measurement instruments. Each guide focuses on a different programming environment. As you develop test programs, you face decisions about how you communicate with the instruments. Some of your choices include Direct I/O, VXIplug&play drivers, or IVI drivers. If you are new to using IVI drivers or just want a quick refresher on the basics, ***Getting Started with IVI Drivers*** can help.

Getting Started with IVI Drivers shows that IVI drivers can be straightforward and easy-to-use tools. IVI drivers provide a number of advantages that can save time and money during development, while improving performance as well. Whether you are starting a new program or making improvements to an existing one, you should consider the use of IVI drivers to develop your test programs.

So consider this the “hello instrument” guide for IVI drivers. If you recall, the “hello world” program, which originally appeared in *Programming in C: A Tutorial*, simply prints out “hello world.” The “hello instrument” program performs a simple measurement on a simulated instrument and returns the result. We think you'll find that far more useful.

Why Use an Instrument Driver?

To understand the benefits of IVI drivers, we need to start by defining instrument drivers in general and describing why they are useful. An instrument driver is a set of software routines that controls a programmable instrument. Each routine corresponds to a programmatic operation, such as configuring, writing to, reading from, and triggering the instrument. Instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the programming protocol for each instrument.

Starting in the 1970s, programmers used device-dependent commands for computer control of instruments. But lack of standardization meant even two digital multimeters from the same manufacturer might not use the same commands. In the early 1990s a group of instrument manufacturers developed Standard

Commands for Programmable Instrumentation (SCPI). This defined set of commands for controlling instruments uses ASCII characters, providing some basic standardization and consistency to the commands used to control instruments. For example, when you want to measure a DC voltage, the standard SCPI command is “MEASURE : VOLTAGE : DC?”.

In 1993, the *VXIplug&play* Systems Alliance created specifications for instrument drivers called *VXIplug&play* drivers. Unlike SCPI, *VXIplug&play* drivers do not specify how to control specific instruments; instead, they specify some common aspects of an instrument driver.

If you have been programming instruments without a driver, then you are probably all too familiar with hunting around the programming guide to find the right SCPI command and exact syntax. You also have to deal with an I/O library to format and send the strings, and then build the response string into a variable.

By using a driver, you can access the instrument by calling a function in your programming language instead of having to format and send an ASCII string as you do with SCPI. With ASCII, you have to create and send the device the syntax “MEASURE : VOLTAGE : DC?”, then read back a string and build it into a variable.

As programming technology has advanced and with such environments as .NET and *Microsoft® Visual Studio® IntelliSense* within the development environment provides a hierarchy to all functionality of the driver from an initial object reference. This makes programming easier since you can navigate logically and are presented with specific choices that are valid in configuring the device. You will be syntactically correct with your configuration of the device since the compiler will inform you of any errors in using the driver.

Why IVI?

The *VXIplug&play* drivers do not provide a common programming interface. That means programming a Keithley DMM using *VXIplug&play* still differs from programming a Keysight DMM. For example, the instrument driver interface for one may be `ke2000_read` while another may be `kt34401_get` or something even more diverse. Without consistency across instruments manufactured by different vendors, many programmers still spent a lot of time learning each individual driver.

In 1998 a group of end users, instrument vendors, software vendors, system suppliers, and system integrators joined together to form a consortium called the Interchangeable Virtual Instruments (IVI) Foundation. All agreed on the need to promote specifications for programming test instruments that provide consistency, better performance, reduce the cost of program development and maintenance, and simplify interchangeability.

The IVI Driver specifications were created to achieve this goal and to extend *VXIplug&play* by providing COM and .NET versions of drivers.

The IVI specifications enable drivers with a consistent and high standard of quality, usability, and completeness. The specifications define an open driver architecture, a set of instrument classes, and shared software components. Together these provide consistency and ease of use, as well as the crucial elements needed for the advanced features. IVI drivers support: instrument simulation, automatic range checking, state caching, and interchangeability.

The IVI Foundation has created IVI class specifications that define the capabilities for drivers for the following thirteen instrument classes:

Class	IVI Driver
Digital multimeter (DMM)	IviDmm
Oscilloscope	IviScope
Arbitrary waveform/function generator	IviFgen
DC power supply	IviDCPwr
AC power supply	IviACPwr
Switch	IviSwch
Power meter	IviPwrMeter
Spectrum analyzer	IviSpecAn
RF signalgenerator	IviRFSigGen
Upconverter	IviUpconverter
Downconverter	IviDownconverter
Digitizer	IviDigitizer
Counter/timer	IviCounter

IVI Class Compliant drivers usually also include numerous functions that are beyond the scope of the class definition. This may be because the capability is not common to all instruments of the class or because the instrument offers some control that is more refined than what the class defines.

IVI also defines custom drivers. Custom drivers are used for instruments that are not members of a class. For example, there is not a class definition for network analyzers, so a network analyzer driver must be a custom driver. Custom drivers provide the same consistency and benefits described below for an IVI driver, except interchangeability.

IVI drivers that conform to the IVI specifications are permitted to display the IVI-Conformant logo.



Why Use an IVI Driver?

Why choose IVI drivers over other possibilities? Because IVI drivers can increase performance and flexibility for more intricate test applications. Here are a few of the benefits:

Consistency – IVI drivers all follow a common model of how to control the instrument. That saves you time when you need to use a new instrument.

Ease of use – IVI drivers feature enhanced ease of use in popular Application Development Environments (ADEs). The APIs provide fast, intuitive access to functions. IVI drivers use technology that naturally integrates in many different software environments.

Quality – IVI drivers focus on common commands, desirable options, and rigorous testing to ensure driver quality.

Simulation – IVI drivers allow code development and testing even when an instrument is unavailable. That reduces the need for scarce hardware resources and simplifies test of measurement applications. The example programs in this document use this feature.

Range checking – IVI drivers ensure the parameters you use are within appropriate ranges for an instrument.

State caching – IVI drivers keep track of an instrument's status so that I/O is only performed when necessary, preventing redundant configuration commands from being sent. This can significantly improve test system performance.

Interchangeability – IVI class compliant drivers also enable exchange of instruments with minimal code changes, reducing the time and effort needed to integrate measurement devices into new or existing systems. The IVI class specifications provide syntactic interchangeability but may not provide behavioral interchangeability. In other words, the program may run on two different instruments but the results may not be the same due to differences in the way the instrument itself functions.

Flavors of IVI Drivers

To support all popular programming languages and development environments, IVI drivers provide either an IVI-C, IVI-COM (Component Object Model), or IVI.NET API. Driver developers may provide multiples of these interfaces, as well as wrapper interfaces optimized for specific development environments.

Although the functionality is the same, IVI-C drivers are optimized for use in ANSI C development environments; IVI-COM drivers are optimized for environments that support the Component Object Model (COM) such as the .NET programming environment. IVI.NET drivers present more flexibility with API data types, access to examining/modifying driver source, and clean handling of Events.

IVI-C drivers extend the *VXI plug&play* driver specification and their usage is similar. IVI-COM and IVI.NET drivers provide easy access to instrument functionality through methods and properties. Only one version of an IVI-COM or IVI-C driver can be installed at a time, whereas IVI.NET allows multiple versions of a driver to be installed.

The getting started examples communicate with the instruments using the Virtual Instrument Software Architecture (VISA) I/O library, a widely used standard library for communicating with instruments from a personal computer. The VISA standard is also provided by the IVI Foundation.

VISA I/O Library

IVI drivers typically require an I/O library to communicate with the instrument. IVI drivers that communicate with GPIB or VXI instruments are required to use the VISA (Virtual Instrument Software Architecture) I/O Library. This guarantees that drivers for GPIB or VXI instruments will work with any interface hardware that supports VISA. If the instrument uses some other interface, such as LAN, USB, or PCI, the driver may use other libraries. In many cases, the necessary I/O support is provided with the operation system.

This getting started example communicates with the instrument using the VISA I/O library. The VISA standard is provided by the IVI Foundation.

Multiple vendors provide VISA I/O libraries – Anritsu, Bustec, Keysight, Kikusui, National Instruments, Rohde & Schwarz, and Tektronix. Many non-modular instruments (box type instruments) will work with different vendors' VISA. A particular vendor's IVI.NET driver installation will recommend either their own or another vendor's VISA I/O Library be installed.

Some PXI/PXIe modular instruments use a specific I/O library and do not require the VISA I/O library.

Shared Components

To make it easier to combine drivers and other software from various vendors, the IVI Foundation members have cooperated to provide common software components, called IVI Shared Components. These components provide services to drivers and driver clients that need to be common to all drivers. For instance, the IVI Configuration Server enables administration of system-wide configuration. It should be noted that different vendors may handle this differently such that these components may be installed with the IVI.NET driver installer, with the I/O package, or may need to be downloaded and installed separately. The ReadMe file for each vendor's IVI.NET driver should be specific on what you must do.

Important! **You must install the IVI Shared Components before an IVI driver can be installed.** This is often controlled by the vendor's install program and some will not permit proceeding with the install without the proper sequence.

The IVI Shared Components can be downloaded from vendors' web sites as well as from the IVI Foundation Web site. In some cases, the installation of the VISA I/O Libraries automatically installs the IVI, IVI.NET, and VISA.NET Shared components.

To download and install shared components from the IVI Foundation Web site:

- 1 Go to the IVI Foundation Web site at <http://www.ivifoundation.org>.
- 2 Locate the Shared Components tab
- 3 Choose the IVI and IVI.NET Shared Components install files.
- 4 The VISA and VISA.NET Shared components should come installed with the Vendor's recommended VISA I/O Library for the driver.

NOTE: installing drivers with different APIs, for modular vs. box-type instruments, or from different vendors may require paying more attention to installation dialog messages and ReadMe files to assure all necessary shared components are loaded and in the correct sequence.

Download and Install IVI Drivers

After you have installed the VISA I/O and Shared Components, you are ready to download and install an IVI driver. For most ADEs, the steps to download and install an IVI driver are identical. For the few that require a different process, the relevant **Getting Started with IVI Drivers** guide provides the information you need. IVI Drivers are available from the hardware or software vendors' web site or by linking to them from the IVI Foundation web site.

The IVI Foundation requires that compliant drivers be registered before they display the IVI conformant logo. To see the list of drivers registered with the IVI Foundation, go to the **Driver Registry** section of the IVI web site at http://www.ivifoundation.org/registered_drivers/driver_registry.aspx

Once on the IVI Foundation Website's Driver Registry, you can use the **Narrow Results by** column and filter on **IVI.NET Driver Type**. This will reveal all of the registered IVI.NET drivers. The **Supported Models** column provides links to the various IVI.NET drivers, which usually reside at the particular vendor's Website. The **Generation** column specifies which version of the IVI standard the IVI.NET driver is conformant. IVI-2014 is the latest version of the standard, and all newly released instrument drivers must conform to that standard.

Follow the embedded hyperlink to the vendor's Website, and once there, you will find additional information (usually a ReadMe File or Application Note) specifying how to download, install, and use the particular IVI.NET driver, along with the recommended VISA and IVI Components.

Familiarizing Yourself with the Driver

Getting Started with IVI Drivers guides provide a driver example that shows the important aspects of using an IVI driver. For this guide, the example is oriented towards using an IVI.NET driver.

You will want to familiarize yourself quickly with drivers you haven't used before. Most ADEs, such as Microsoft Visual Studio, provide a way to explore IVI drivers to learn their functionality using an **Object Browser**.

In addition, browsing an IVI driver's help file often proves an excellent way to learn its functionality. The help and example files are part of the instrument driver installation and are installed under the IVI Foundation directories:

For IVI-C and IVI-COM:

Program Files / IVI Foundation / IVI / Drivers / ...

Program Files (x86) / IVI Foundation / IVI / Drivers / ...

...then look under the various Vendor directories for Examples and Help files

For IVI.NET:

Program Files / IVI Foundation / IVI / Microsoft.NET / Framework64 / ...

Program Files (x86) / IVI Foundation / IVI / Microsoft.NET / Framework32 / ...

...the version of the .NET Framework (2.x, 3.0, 3.5, etc.) is presented as directories for the particular environment the IVI.NET driver was targeted. Look for Vendor Examples and Help files under these directories.

Help files are typically in the form of CHM or Compiled HTML Help, but they may be other formats.

Examples

Each vendor supplying an IVI.NET driver is required to provide example code and documentation that describes how to install all of the necessary components, including the IVI.NET driver, as stated in the previous section. For IVI.NET drivers, C# and Visual Basic .NET are the obvious choices. The examples are usually similar to the example in this guide but will often show much more detail in configuring the instrument for its typical operation, so you can quickly get your test system code running.

Each guide in the **Getting Started with IVI Drivers** series shows you how to use an IVI driver to write and run a program that performs a simple measurement on a simulated instrument and returns the result. The examples demonstrate common steps using IVI drivers. Where practical, every example includes the steps listed below:

- Download and install the VISA, Shared Components and IVI driver
- Determine the VISA address string. Use an I/O application such as **National Instruments Measurement and Automation Explorer** (NI-MAX) or **Keysight Connection Expert**.
- Reference the IVI driver and any IVI Shared Components.
- Create an instance of the driver object
- Write the program to setup and control instrument.
 - Configure the instrument
 - Access an instrument's properties.
 - Set timeout for instrument interactions
 - Make a measurement
 - Display the measurement result
 - Check the instrument for any errors
 - Close the I/O session to the instrument

Note - it is good programming practice to call **Close()** to close the I/O session to the instrument. For .NET, the object is disposed when the main driver object is destroyed, whether or not the **Close()** is called or not.



Chapter 2

Using IVI.NET with Visual C# and Visual Basic .NET

• • •

The Environment

C# and **Visual Basic.NET** are object-oriented programming languages developed by Microsoft. They enable programmers to quickly build a wide range of applications for the Microsoft .NET platform. This chapter provides detailed instructions in **C#** as well as illustrating the equivalent code for **Visual Basic.NET**.

Since many test systems are a hybrid of modular and box-type products and from different vendors, this example mixes resources from different vendors to show that compatibility: National Instruments VISA for instrument connectivity, IVI Foundation Shared components, and an IVI.NET driver from Keysight.

Example Requirements

- Windows 7/8/10 64-bit operating system
- Microsoft Visual Studio 2012 and later
- Visual C#
- National Instruments – [NI-VISA 16.0](#)
- IVI Foundation - [IviSharedComponents64_241.exe](#)
- IVI Foundation - [IviNetSharedComponents64_Fx20_1.2.0.exe](#)
- Keysight IntegraVision IVI.NET Driver – a custom driver that has some similarities to a Scope. [IntegraVision IVI.NET driver](#) link takes you to the download page from which you will select the IVI.NET 64-bit version of the driver.

Download and Install the Driver

If you have not already installed the VISA, Shared Components and IVI.NET driver, refer to Chapter 1 for more specifics. The installer for the Keysight IntegraVision driver will prompt you if any of the required shared components are not already installed on your system, and there is also a ReadMe file you can follow to understand the installation of all necessary components and any order dependency.

Create a New Project and Reference the Driver

Begin by creating a new project and add any necessary references to the IVI Driver and Shared Components.

- 1 Launch Visual Studio and create a new Console Application in Visual C# by selecting File -> New -> Project and selecting a Visual C# Console Application.

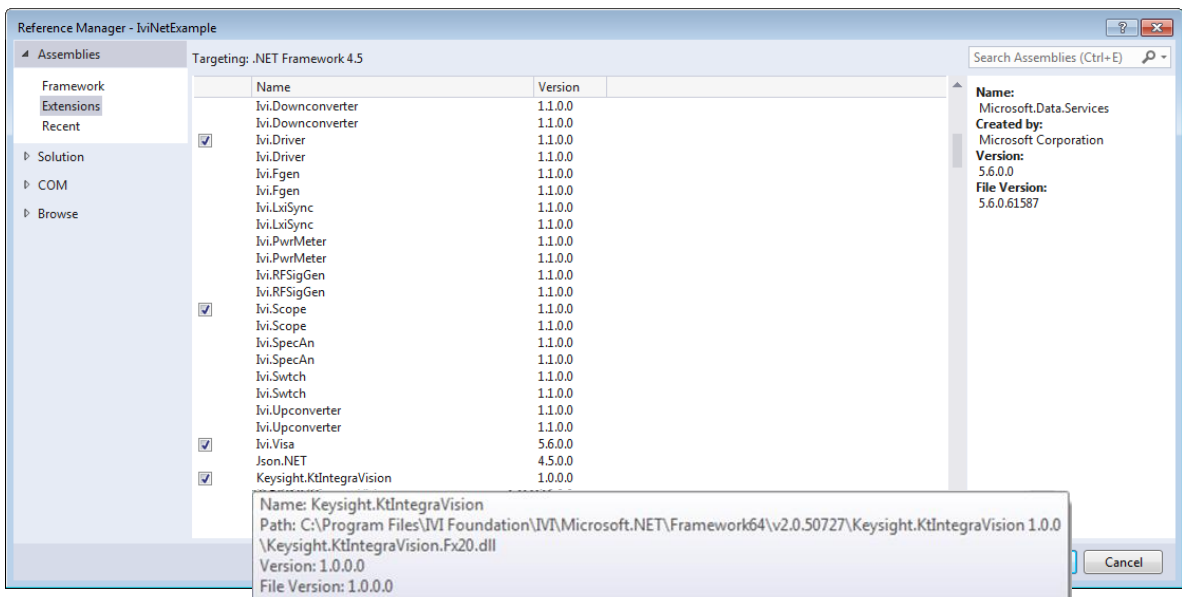
Note: When you select new, Visual Studio will create an empty program that includes some necessary code, including *using* statements. Keep this required code as the basis of the example.

For the next steps you will need to ensure that the "Program.cs" editor window is visible and the **Solution Explorer** is visible.

- 2 Right-click **References** in the **Solution Explorer** and then select **Add Reference**. When the Add Reference dialog appears, select **Extensions** under **Assemblies**.
- 3 The following image shows which items to select: Shared Components Ivi.Driver, Ivi.Scope, and Ivi.Visa. Select Keysight.KtIntegraVision.

Note: If you have not installed the IVI driver, it will not appear in this list. You must close the Add Reference dialog, install the driver, and select Add Reference again for the driver to appear.

Also Note: Two entries appear for most assemblies – representing both 32-bit and 64-bit versions. Hover over the assembly to determine its type and where it is actually located. Select the 64-bit versions.

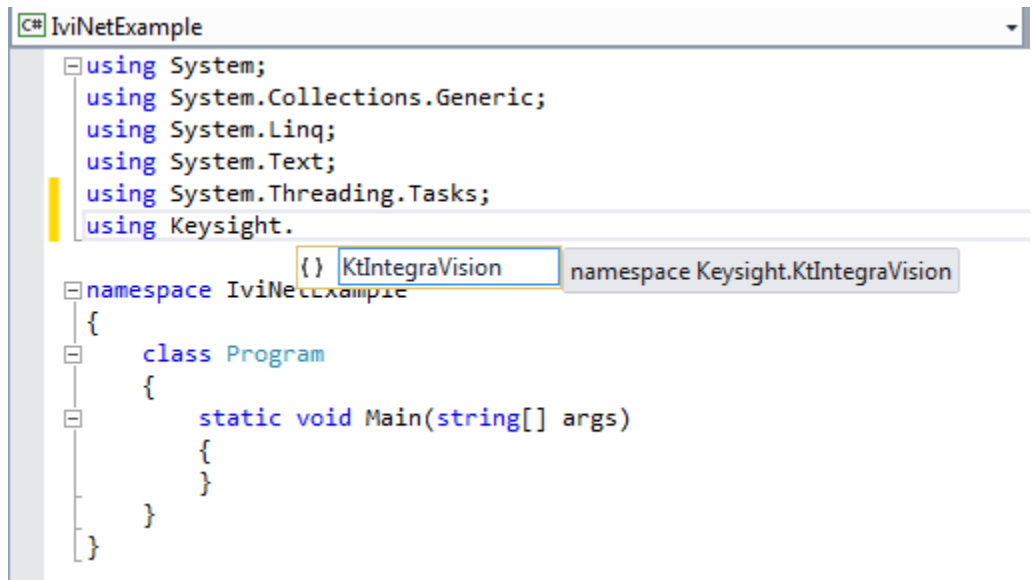


Note: Adding the reference changes nothing in the default program created by Visual Studio, but the driver references are now available for use. To see the reference, select View and click **Solution Explorer**. **Solution Explorer** appears and lists the reference.

Create an Instance of the Driver

Before creating the instance of the driver, the default program should be modified for easy access of the previously referenced driver. To allow your program to access the driver without specifying the full namespace for every program entry, type the following line immediately below the other default using statements. See figure below:

Note: As soon as you type the *Keysight*, **IntelliSense** lists the valid inputs.



```
C# IviNetExample
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Keysight.

namespace IviNetExample
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

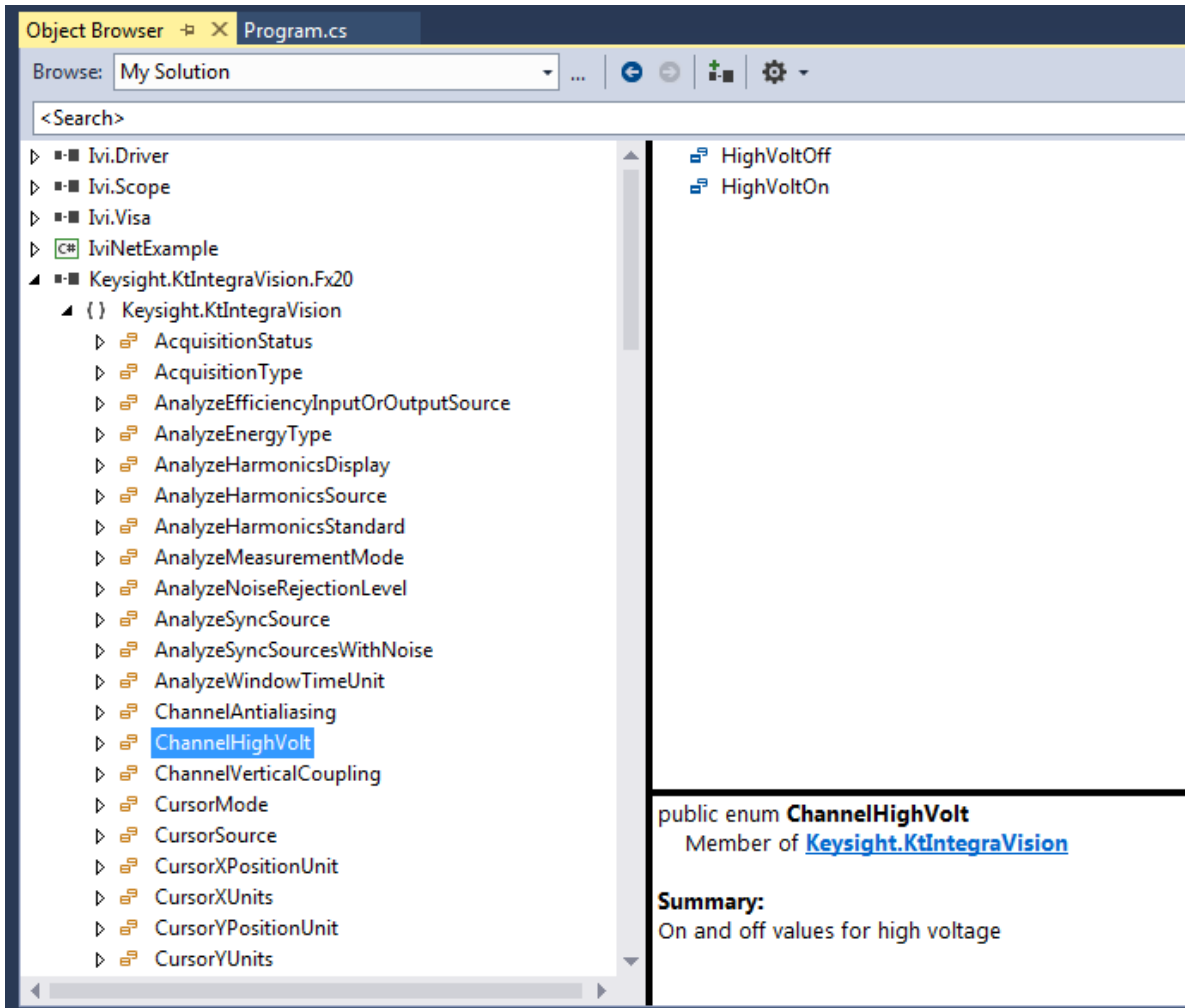
When finished adding the Keysight reference, add “`using Ivi.Driver;`” This gives access to Shared Components such as common Driver Exceptions, which is the basis for trapping error conditions in the .NET environment.

Congratulations! Your program will now compile and you can write the rest of the program to control the simulated instrument.

Use the Object Browser for Help

Now that you have created the initial project, you may need help finding the correct properties and functions to call when programming the instrument.

Microsoft's **Object Browser** provides a visual and hierarchical representation of the IVI driver functionality. If the **Object Browser** is not already visible, find the **VIEW** tab at the top of Visual Studio and select **Object Browser**.



Now continue to add to the C# program as this example progresses.

The following program snippet shows creation of the IVI.NET driver object. Since any failure here should be caught immediately, this operation has its own [try-catch](#) structure, and there will be a new [try-catch-finally](#) structure for the rest of the program example.

Optionally using .NET **Named Parameters** makes the code self-documenting, which this example uses extensively to make it easier to read. Note that **options** uses **Simulate=true**, so you don't actually have to have the instrument present to write the program. With simulation, the **resourceName** can be anything, since it will not actually be used if you run the program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Keysight.KtIntegraVision;
using Ivi.Driver;

namespace IviNetExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var driver = (KtIntegraVision)null;

            try
            {
                driver = new KtIntegraVision(resourceName: "MyVisaAddress",
                                             idQuery: true,
                                             reset: true,
                                             options: "Simulate=true");
            }
            catch (Ivi.Driver.IOException ex)
            {
                Console.WriteLine("Error connecting to Instrument: " + ex.Message);
                Console.ReadKey();
            }
        }
    }
}
```

Note: IntelliSense helps ensure you use the correct syntax and values. You will experience this every time you begin to type in something that is a part of the [using](#) statements at the top of the program.

Configure the Instrument

All instruments have some sort of configuration for a particular measurement or output setup. A DMM might have fewer steps to get towards making a measurement than a scope-type instrument, but with an IVI driver, all setup is relatively small. The example below shows the configuration of an instrument that has multiple channels. In this case, the IVI Driver **Repeated Capabilities** feature allows the same setup to be used for each channel, but you provide a name for each channel ("Channel1"). Note that this particular driver mixes both the "Channel1" designator and also uses an integer value, which is not a recommended driver implementation practice. However, you may see it in other drivers, both IVI.NET and IVI-COM.drivers. Each of the steps below are illustrated in the program snippet image that follows.

- Use [PrecisionTimeSpan](#) to set timeout operations with instrument. This shared component helps you to always pick the correct time units.
- Enable all measurement functions for Channel1
- Set the coupling to DC using a property setting and enumerated type
- Auto-setup the channel.
- Configure the Trigger Subsystem for a measurement. Note that [TriggerSlope](#) is also a part of the IVI Shared Components.
- Select the desired measurements as a list to be returned when querying the results.

```
// Limit how long to wait for instrument operations
driver.System.IOTimeout = PrecisionTimeSpan.FromSeconds(10);

// Enable all measurements: Voltage, Current, and Power on Channel 1
driver.Channels["Channel1"].EnableAll();

// Autoseup and then set trigger source to Current on Channel 1
driver.RunControl.AutoSetup();
driver.Trigger.Configure(source: Keysight.KtIntegraVision.TriggerSource.Current,
    channelNumber: 1,
    level: 0.1,
    slope: TriggerSlope.Positive);

// Set the measurement source to power on channel 1
driver.Measure.SetSource(source: MeasureSource.Power, channelNumber: 1);

// Add the desired measurements and read results
driver.Measure.AddMeasurement(measurementType: MeasurementAddType.PeakToPeak);
driver.Measure.AddMeasurement(measurementType: MeasurementAddType.Min);
driver.Measure.AddMeasurement(measurementType: MeasurementAddType.Max);
```

Display the Results

For this instrument, the `AutoSetup()` and `Trigger.Configure` creates the environment for actually measuring the applied signal, very much like having the instrument in a free-running bench operation. All that is needed is to acquire the measurement result from the instrument using the following:

```
var PowerResults = driver.Measure.GetResults();
```

Check for Errors

When using the .NET programming environment, Exceptions are the preferred method to trap error conditions. The IVI .NET Shared Components provide common error exceptions that cover a wide range of errors and warnings. The following is an example of using two exceptions.

```
try
{
    // Program the Instrument
}
catch (Ivi.Driver.InstrumentStatusException ex)
{
    // Handle errors flagged by instrument
}
catch (Ivi.Driver.IOException ex)
{
    // Handle general IO errors with instrument
}
```

With message-based instruments, the IVI .NET Shared Components provide a single call to query both the error number and error string using [ErrorQueryResult](#). This may be necessary when revisiting an instrument where some condition has occurred since you last programmed it. Otherwise, any programming error condition will be caught with an Exception.

For example, losing a 10MHz Reference signal causes an instrument Warning or Error. This might generate quite a few queued error conditions in the instrument's error queue. All queued errors can be read from the instrument when looping using the following code:

```
// Check instrument for errors
ErrorQueryResult result;
do
{
    result = driver.Utility.ErrorQuery();
    Console.WriteLine("ErrorQuery: {0}, {1}", result.Code, result.Message);
} while (result.Code != 0);
```

The Exception may be a single Event, and you can then use the above example to read any remaining errors in the queue.

Close the Session

It is good programming practice to call **Close()** even if .NET will eventually dispose of the driver object. Performing the close operation controls *when* the object is closed. The following is added to the [try-catch](#) discussed in **Checking for Errors** and creates an overall a [try-catch-finally](#), where the [finally](#) is always the last thing called. This is where we put the closing of the IVI Driver.

```
finally
{
    driver.Close();
    Console.WriteLine("\n\nPress any key to end program...");
    Console.ReadKey();
}
```

Complete program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Keysight.KtIntegraVision;
using Ivi.Driver;

namespace IviNetExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var driver = (KtIntegraVision)null;

            try
            {
                driver = new KtIntegraVision(resourceName: "MyVisaAddress",
                    idQuery: true,
                    reset: true,
                    options: "Simulate=true");
            }
            catch (Ivi.Driver.IOException ex)
            {
                Console.WriteLine("Error connecting to Instrument: " + ex.Message);
                Console.ReadKey();
            }

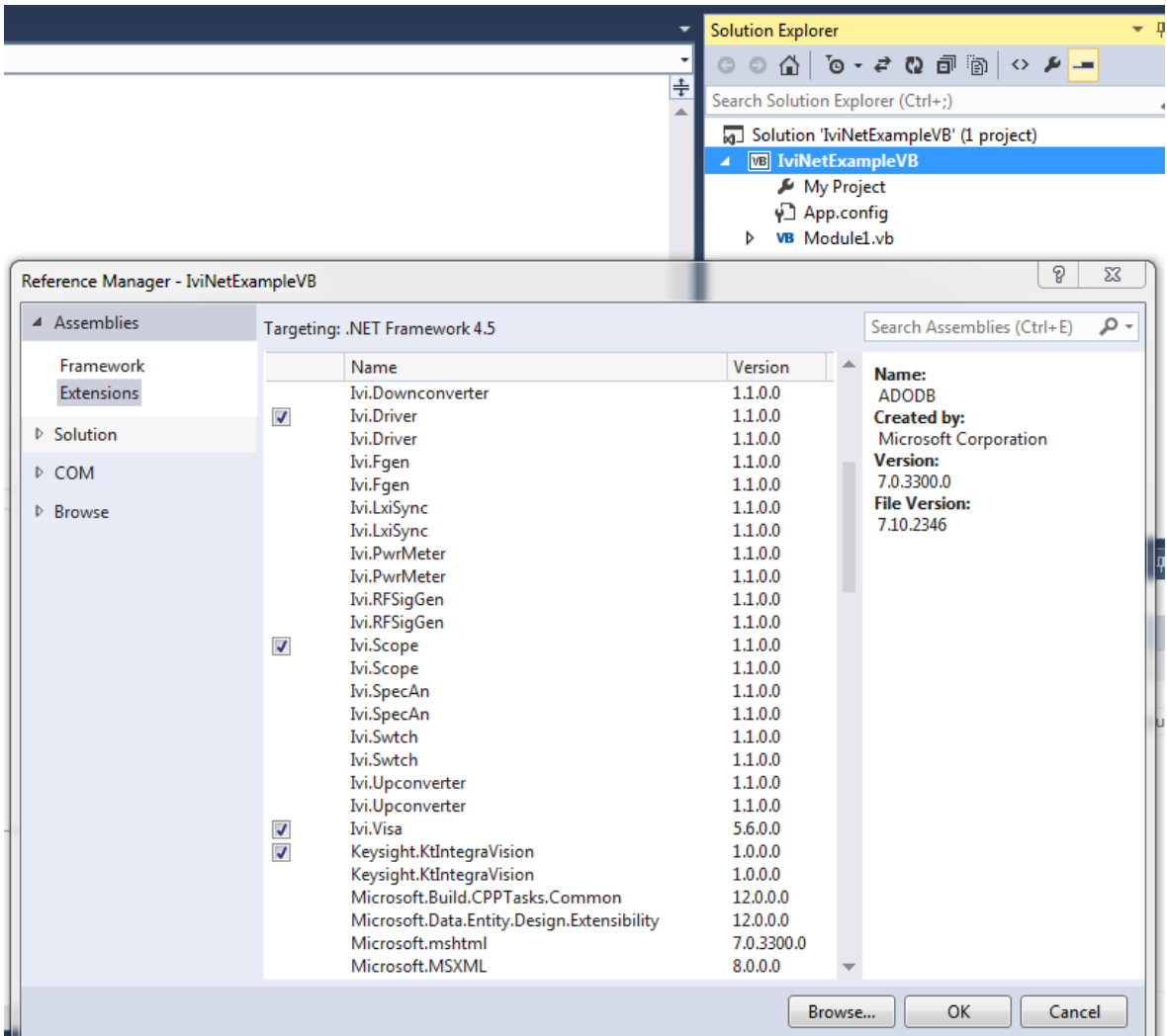
            try
            {
                // Limit how long to wait for instrument operations
                driver.SYSTEM.IOTimeout = PrecisionTimeSpan.FromSeconds(10);

                // Enable all measurements: Voltage, Current, and Power on Channel 1
                driver.Channels["Channell"].EnableAll();

                // Autoseup and then set trigger source to Current on Channel 1
                driver.RunControl.AutoSetup();
                driver.Trigger.Configure(source: Keysight.KtIntegraVision.TriggerSource.Current,
                    channelNumber: 1,
                    level: 0.1,
                    slope: TriggerSlope.Positive);
            }
        }
    }
}
```


Visual Basic.NET Code

Virtually all of the steps above are the same when using Visual Basic.NET. To add the Reference for IVI Driver and IVI Shared Components, right-click the IviNetExampleVB project as illustrated in the figure below, Add Reference, choose Extensions, and select the same references as detailed under the C# example: Ivi.Driver, Ivi.Scope, Ivi.Visa, and Keysight.KtIntegraVision, the 64-bit versions.



Here is the equivalent code example:

```
Imports Ivi.Driver
Imports Keysight.KtIntegraVision

Module Module1

    Sub Main()

        Dim driver As KtIntegraVision

        Try
            driver = New KtIntegraVision(resourceName:="MyVisaAddress",
                idQuery:=True,
                reset:=True,
                options:="Simulate=true")

        Catch ex As Ivi.Driver.IOException
            Console.WriteLine("Error connecting to Instrument: " + ex.Message)
            Console.ReadKey()
        End Try

        Try
            ' Limit how long to wait for instrument operations
            driver.System.IOTimeout = PrecisionTimeSpan.FromSeconds(10)

            ' Enable all measurements: Voltage, Current, and Power on Channel 1
            driver.Channels("Channel1").EnableAll()

            ' Autoseup and then set trigger source to Current on Channel 1
            driver.RunControl.AutoSetup()
            driver.Trigger.Configure(source:=Keysight.KtIntegraVision.TriggerSource.Current,
                channelNumber:=1,
                level:=0.1,
                slope:=TriggerSlope.Positive)

            ' Set the measurement source to power on channel 1
            driver.Measure.SetSource(source:=MeasureSource.Power, channelNumber:=1)

            ' Add the desired measurements and read results
            driver.Measure.AddMeasurement(measurementType:=MeasurementAddType.PeakToPeak)
            driver.Measure.AddMeasurement(measurementType:=MeasurementAddType.Min)
            driver.Measure.AddMeasurement(measurementType:=MeasurementAddType.Max)
            Dim PowerResults = driver.Measure.GetResults()
        Catch ex As Ivi.Driver.InstrumentStatusException
            ' Handle errors flagged by instrument
        Catch ex As Ivi.Driver.IOException
            ' Handle general IO errors with instrument
        Finally
            driver.Close()
            Console.WriteLine("\n\nPress any key to end program...")
            Console.ReadKey()
        End Try
    End Sub

End Module
```

Further Information

- Learn more about Visual C# at <http://msdn.microsoft.com/vcsharp/>.
- Learn more about Visual Basic at <http://msdn.microsoft.com/vbasic/>.
- IVI Foundation forums at <http://forums.ivifoundation.org/>
- IVI Foundation resources at <http://ivifoundation.org/resources/default.aspx>

Microsoft® and Visual Studio® are registered trademarks of Microsoft Corporation in the United States and/or other countries.



Chapter 3

IVI.NET vs. IVI-COM



Overview

Consistency in the development of IVI drivers has been a high priority for the IVI Foundation, and any differences between test system development tools and environments have reasons for best performance and usability. This section provides insight into differences between IVI.NET and IVI-COM drivers.

Here is a little history and purpose for the two driver types:

IVI-COM COM (Common Object Model) is a language independent object oriented interface standard for components introduced by Microsoft in 1993. COM is supported by many different environments. IVI-COM drivers provide a COM type library and a .NET interop assembly or wrapper. IVI-COM drivers are easily accessed from .NET environments and can also be accessed from other environments such as National Instruments LabVIEW and Microsoft VBA.

IVI.NET IVI.NET drivers provide a native .NET interface to the instrumentation and provide the best end-user experience for those working in a .NET environment and using languages such as C# and VB.NET

Although the IVI-COM interface is very good in .NET, the IVI.NET drivers add native .NET characteristics allowing application programs to more easily deal with things like collections, enumerated types, and complex data types found in using waveforms.

The IVI Foundation made a deliberate decision to design the required IVI.NET API to best match what IVI.NET users expect, and this inevitably means taking advantage of .NET-only features, which cannot be represented in COM.

IVI Driver Source Code

Many IVI drivers provide the option to include source code for the driver during installation. However, IVI-COM drivers are quite complicated internally. IVI.NET driver source is easier to understand and modify by a programmer and could provide a means to enhance or fix the driver. Source code is located in the same locations as the Driver, Help and Example directories:

For IVI-COM:

Program Files / IVI Foundation / IVI / Drivers /...

Program Files (x86) / IVI Foundation / IVI / Drivers /...

For IVI.NET:

Program Files / IVI Foundation / IVI / Microsoft.NET / Framework64 /...

Program Files (x86) / IVI Foundation / IVI / Microsoft.NET / Framework32 /...

A generic Visual Studio project file is typically provided to build in the 32-bit or 64-bit environments.

Side-by-side Deployment of IVI Drivers

Only a single IVI-COM driver can be installed at a time for a particular instrument; whereas, IVI.NET allows multiple versions of a driver to be installed.

For IVI-COM, you add a single reference once in the .NET environment, since newer versions replace older versions with the same DLL reference and location. For IVI.NET, there is a different version number for each release of the driver and different programs can continue to run previous versions of the driver.

IVI.NET's Richer Type System

IVI-COM has a limited set of data types – Integers, Floats, Booleans, and Strings. Here is an example showing an IVI.NET special added type:

```
Channels[ ].Measurement.FetchWaveform) IWaveform<Double> waveform);
```

Other data types added with IVI.NET besides [IWaveform](#) include [PrecisionDateTime](#) and [PrecisionTimeSpan](#) (see below). With these later types, there is no longer confusion about time units, such as milliseconds vs. seconds, or UTC time, etc.

IVI-COM drivers provide a means to return variables, arrays, and strings or they can be passed references to variables, arrays, and strings.

IVI.NET can return objects and structures.

PrecisionTimeSpan

IVI-COM drivers represent time in seconds; whereas, IVI.NET drivers present units of time implicit in the definition of [PrecisionTimeSpan](#) and [PrecisionDateTime](#), providing days, hours, microseconds, nanoseconds, etc. These IVI shared components provide more resolution than the corresponding .NET [DateTime](#) and [TimeSpan](#) classes.

Initializing the Driver

IVI-COM creates the driver object and then initializes the driver.

```
IAgMSwitch driver = new IAgMSwitch();  
driver.Initialize(visaAdd, true, true, "");
```

IVI.NET performs the initialization when creating the driver object:

```
KtIntegraVision driver = new KtIntegraVision(visaAdd, true, true, "");
```

Error Handling

IVI-COM Error handling depends upon the programming environment. When programming from .NET, every error generated in the instrument driver is transmitted to the test program as a .NET COMException. These driver errors can be trapped with [try-catch-finally](#) capability in .NET. For message based devices, the following can be used to query error conditions directly from the instrument.

```
string ErrorMessage = null;
int ErrorCode = 0;

driver.Utility.ErrorQuery(ref ErrorCode, ref ErrorMessage);
```

IVI.NET Most errors are returned as exceptions and should be trapped using the [try-catch-finally](#) capabilities in .NET. The example program in Chapter 2 illustrates a couple of error conditions that might occur from various interactions with the driver.

There may be cases where an error or warning may occur in an instrument since the last time you tried to program it. This might occur if an instrument loses its high precision frequency locking signal, for example. If desired, you can use this secondary mechanism to query the instrument directly with the following. Note the new type established for the Error results as compared to IVI-COM drivers.

```
ErrorQueryResult result;
result = driver.Utility.ErrorQuery();
Console.WriteLine("{0} : {1}", result.Code, result.Message);
```

Events from the Driver

IVI-COM drivers almost never expose events, and exposing something commonly needed as an SRQ event from an instrument is difficult and requires special knowledge and programming. In general, IVI-COM exceptions are primarily oriented around I/O errors with the instrument.

IVI.NET provides a standard mechanism for exposing events and requires no special programming. To use events, the programmer simply subscribes to or unsubscribes from the event.

Syntax for Enumerations and Repeated Capabilities

IVI.NET provides a simpler syntax for referring to enumerated values and also repeated capabilities.

In the example below, the detection level on Channel 3 is being set to the High level. Both the reference to the repeated capability (“[Channel3](#)”) and the enumeration (“[ArmSourceDetection.High](#)”) are somewhat simpler in .NET.

IVI-COM

```
digitizer.Arm.Source.get_Item("Channel3").Detection =  
IviLxiSyncArmSourceDetectionEnum.IviLxiSyncArmSourceDetectionHigh;
```

IVI.NET

```
digitizer.Arm.Source["Channel3"].Detection =  
ArmSourceDetection.High;
```