



# ***Getting Started with IVI Drivers***

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

**Version 1.2**

**© Copyright IVI Foundation, 2015  
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.*



# Contents



- Chapter 1 Introduction.....4**
  - Purpose.....4
  - Why Use an Instrument Driver? ..... 4
  - Why IVI?..... 5
  - Why Use an IVI Driver? ..... 7
  - Flavors of IVI Drivers ..... 8
  - Shared Components .....8
  - Download and Install IVI Drivers ..... 8
  - Familiarizing Yourself with the Driver ..... 9
  - Examples .....9
  
- Chapter 2 Using IVI with Visual C# and Visual Basic .NET ..... 12**
  - The Environment .....12
  - Example Requirements.....12
  - Download and Install the Driver ..... 12
  - Create a New Project and Reference the Driver ..... 13
  - Create an Instance of the Driver ..... 15
  - Initialize the Instrument.....16
  - Configure the Instrument .....17
  - Set the Trigger Delay .....17
  - Set the Reading Timeout/Display the Reading.....17
  - Close the Session ..... 18
  - Build and Run the Application ..... 19
  - Tips. ....19
  - Further Information .....20



# Chapter 1

## Introduction

•••

### Purpose

Welcome to ***Getting Started with IVI Drivers: Your Guide to Using IVI with Visual C# and Visual Basic .NET***. This guide introduces key concepts about IVI drivers and shows you how to create a short program to perform a measurement. 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, VXI *plug&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, 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. By using a driver, you can access the instrument by calling a subroutine 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 instrument the syntax “MEASURE : VOLTAGE : DC?”, then read back a string, and build it into a variable. With a driver you can merely call a function called `MeasureDCVoltage()` and pass it a variable to return the measured voltage.

Although you still need to be syntactically correct in your calls to the instrument driver, making calls to a subroutine in your programming language is less error prone. If you have been programming to 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.

## 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 `ag34401_get` or something even farther afield. Without consistency across instruments manufactured by different vendors, many programmers still spent a lot of time learning each individual driver.

To carry *VXIplug&play* drivers a step (or two) further, 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. If you look at the membership, it's clear that many of the foundation members are competitors. But all agreed on the need to promote specifications for programming test instruments that provide better performance, reduce the cost of program development and maintenance, and simplify interchangeability.

For example, for any IVI driver developed for a DMM, the measurement command is *IviDmmMeasurement.Read*, regardless of the vendor. Once you learn how to program the commands specified by IVI for the instrument class, you can use any vendor's instrument and not need to relearn the commands. Also commands that are common to all drivers, such as *Initialize* and *Close*, are identical regardless of

the type of instrument. This commonality lets you spend less time browsing through the help files in order to program an instrument, leaving more time to get your job done.

That was the motivation behind the development of IVI 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.

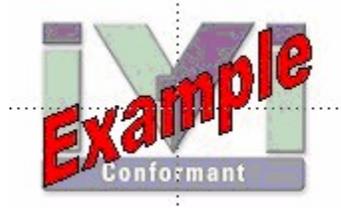
The IVI Foundation has created IVI class specifications that define the capabilities for drivers for the following thirteen instrument classes:

<b>Class</b>	<b>IVI Driver</b>
Digital multimeter (DMM)	IviDmm
Oscilloscope	IviScope
Arbitrary waveform/function generator	IviFgen
DC power supply	IviDCPwr
AC power supply	IviACPwr
Switch	IviSwth
Power meter	IviPwrMeter
Spectrum analyzer	IviSpecAn
RF signal generator	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 or IVI-COM (Component Object Model) API. Driver developers may provide either or both 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-C drivers extend the *VXIplug&play* driver specification and their usage is similar. IVI-COM drivers provide easy access to instrument functionality through methods and properties.

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.

## 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.

Important! **You must install the IVI Shared Components before an IVI driver can be installed.**

The IVI Shared Components can be downloaded from vendors' web sites as well as from the IVI Foundation Web site.

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 Shared Components.
- 3 Choose the IVI Shared Components msi file for the Microsoft Windows Installer package or the IVI Shared Components exe for the executable installer.

## Download and Install IVI Drivers

After you've installed Shared Components, you're 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 the display the IVI conformant logo. To see the list of drivers registered with the IVI Foundation, go to the registration section of the IVI web site at <http://www.ivifoundation.org>.

## **Familiarizing Yourself with the Driver**

Although the examples in *Getting Started with IVI Drivers* use a DMM driver, you will likely employ a variety of IVI drivers to develop test programs. To jumpstart that task, you'll want to familiarize yourself quickly with drivers you haven't used before. Most ADEs provide a way to explore IVI drivers to learn their functionality. In each IVI guide, where applicable, we add a note explaining how to view the available functions. In addition, browsing an IVI driver's help file often proves an excellent way to learn its functionality.

## Examples

As we noted above, 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 IVI driver— covered in the Download and Install IVI Drivers section above.
- Determine the VISA address string – Examples in **Getting Started with IVI Drivers** use the simulate mode, so we chose the address string **GPIB0::23::INSTR**, often shown as GPIB::23. If you need to determine the VISA address string for your instrument and the ADE does not provide it automatically, use an IO application, such as National Instruments Measurement and Automation Explorer (MAX) or Keysight Connection Expert.
- Reference the driver or load driver files – For the examples in this guide, the driver is the **IVI-COM/IVI-C Version 1.2.5.0 for 34401A, April 2013 (from Keysight Technologies)** . . . or the **Keysight 34401A IVI-C driver, Version 4.5, January 2015 (from National Instruments)**.
- Create an instance of the driver in ADEs that use COM – For the examples in the IVI guides, the driver is the **Agilent 34401A (IVI-COM) or HP 34401 (IVI-C)**.
- Write the program. The programs in this series all perform the following steps:
  - Initialize the instrument – Initialize is required when using any IVI driver. Initialize establishes a communication link with the instrument and must be called before the program can do anything with the instrument. The examples set reset to **true**, ID query to **false**, and simulate to **true**.

Setting reset to true tells the driver to initially reset the instrument. Setting the ID query to false prevents the driver from verifying that the connected instrument is the one the driver was written for. Finally, setting simulate to true tells the driver that it should not attempt to connect to a physical instrument, but use a simulation of the instrument.

- Configure the instrument – The examples set a range of **1.5 volts** and a resolution of **0.001 volts (1 millivolt)**.
- Access an instrument property – The examples set the trigger delay to **0.01 seconds**.
- Set the reading timeout – The examples set the reading timeout to **1000 milliseconds (1 second)**.
- Take a reading
- Close the instrument – This step is required when using any IVI driver, unless the ADE explicitly does not require it. We close the session to free resources.

**Important! Close may be the most commonly missed step when using an IVI driver. Failing to do this could mean that system resources are not freed up and your program may behave unexpectedly on subsequent executions.**

- Check the driver for any errors.
- Display the reading.

**Note:** *Examples that use a console application do not show the display.*

Now that you understand the logic behind IVI drivers, let's see how to get started.



## Chapter 2

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

• • •

### The Environment

C# and Visual Basic 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 the code for Visual Basic. NET. If you are using Visual Basic 6.0, we recommend another guide in this series, **Getting Started with IVI Drivers: Your Guide to Using IVI with Visual Basic 6**.

**Note:** *One of the key advantages of using C# and Visual Basic in the Microsoft® Visual Studio® Integrated Development Environment is IntelliSense™. IntelliSense is a form of autocompletion for variable names and functions and a convenient way to access parameter lists and ensure correct syntax. The feature also enhances software development by reducing the amount of keyboard input required.*

### Example Requirements

- Visual C#
- Microsoft Visual Studio 2010
- Agilent 34401A IVI-COM, Version 1.2.2.0, October 2008 (from Agilent Technologies)
- Agilent IO Libraries Suite 16.1

### Download and Install the Driver

If you have not already installed the driver, go to the vendor Web site and follow the instructions to download and install it. You can also refer to Chapter 1, Download and Install IVI Drivers, for instructions.

This example uses an IVI-COM driver. IVI-COM is the preferred driver for C#, but IVI-C is also supported.

## Create a New Project and Reference the Driver

Begin by creating a new project, and add a reference to the IVI Driver.

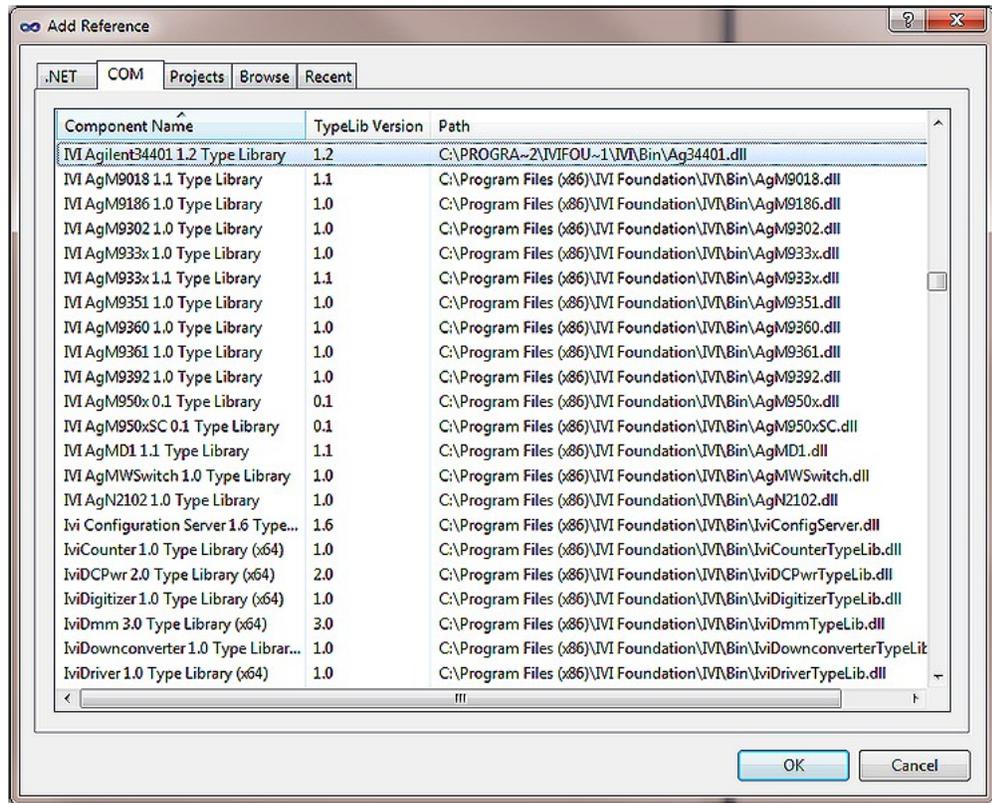
- 1 Launch Visual Studio and create a new Console Application in Visual C# by selecting File -> New -> Project and selecting a Visual C# Console Application.

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

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

- 2 Select Project and click Add Reference. The Add Reference dialog appears.
- 3 Select the COM tab. All IVI drivers begin with IVI. Scroll to the IVI section and select IVI Agilent 34401 (Agilent Technologies) 1.2 Type Library. Click OK.

**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.

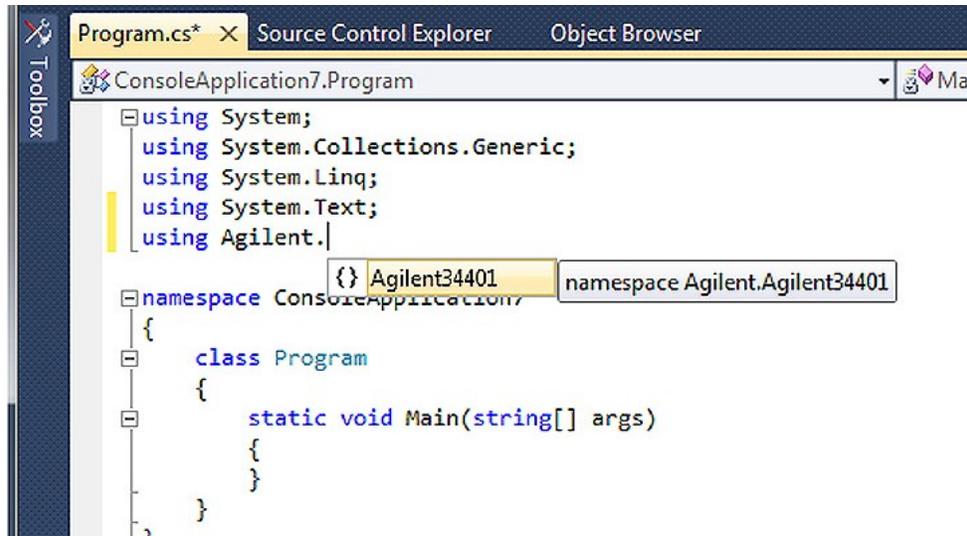


**Note:** The program looks the same as it did before you added the reference, but the driver is now available for use. To see the reference, select View and click Solution Explorer. Solution Explorer appears and lists the reference.

## Create an Instance of the Driver

To allow your program to access the driver without specifying the full path, type the following line immediately below the other `using` statements:  
`using Agilent.Agilent34401.Interop;`

**Note:** As soon as you type the A for Agilent, IntelliSense lists the valid inputs.



The screenshot shows the Visual Studio IDE with a C# file named `Program.cs` open. The code contains several `using` statements: `using System;`, `using System.Collections.Generic;`, `using System.Linq;`, `using System.Text;`, and `using Agilent.`. Below these, there is a `namespace ConsoleApplication7` containing a `class Program` with a `static void Main(string[] args)` method. An IntelliSense popup is visible, showing a list of namespaces under the `Agilent` prefix, with `Agilent34401` selected. The popup also shows the full namespace path: `namespace Agilent.Agilent34401`.

Congratulations! You may now write the program to control the simulated instrument.

**Note:** To view the functions and parameters available in the instrument driver, right-click the library in the References folder in Solution Explorer and select View in Object Browser.

The screenshot displays the Visual Studio IDE with two panes. The left pane shows the 'Agilent.Agilent34401.Interop' namespace tree, with 'IAgent34401' selected. The right pane shows the 'Initialize(string, bool, bool, string)' method signature and its summary.

**Initialize(string, bool, bool, string)**

AC  
ACCurrent  
ACVoltage  
Advanced  
Calibration  
DCCurrent  
DCVoltage  
DCVoltageRatio  
Display  
DriverOperation  
Frequency  
Function  
Identity  
Initialized  
IviDmm  
Math  
Measurement  
Resistance  
Status  
System  
Trigger  
Utility

**void Initialize**(string ResourceName, bool IdQuery, bool Reset, [string OptionString = null])  
Member of [Agilent.Agilent34401.Interop.IAgent34401](#)

**Summary:**  
Opens the I/O session to the instrument. Driver methods and properties that

## Initialize the Instrument

You can now write the main constructs for your program. Create a variable to represent your instrument and set the Initialization parameters.

- 1 Type `Agilent34401 dmm = new Agilent34401();`
- 2 Type `dmm.Initialize ("GPIB::23", false, true, "simulate=true");`

**Note:** IntelliSense helps ensure you use correct syntax and values.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Agilent.Agilent34401.Interop;

namespace ConsoleApplication7
{
    class Program
    {
        static void Main(string[] args)
        {
            Agilent34401 dmm = new Agilent34401();
            dmm.Initialize(
                void IAgilent34401.Initialize(string ResourceName, bool IdQuery, bool Reset, [string...])
                Opens the I/O session to the instrument. Driver methods and properties that access
                ResourceName: An IVI logical name or an instrument specific string that identifies
            )
        }
    }
}
```

### Configure the Instrument

To set the range to 1.5 volts and the resolution to 1 millivolt, type

```
dmm.DCVoltage.Configure(1.5, 0.001);
```

### Set the Trigger Delay

To set the trigger delay to 0.01 seconds, type

```
dmm.Trigger.Delay = 0.01;
```

### Set the Reading Timeout/Display the Reading

Create a variable to represent the reading and display the reading:

- 1 Type `double reading;`

- 2 To trigger the multimeter and take a reading with a timeout of 1 second, type  
`reading = dmm.Measurement.Read(1000);`
- 3 Type `Console.WriteLine("The measurement is {0}", reading);`
- 4 Type `Console.ReadLine();`

## Close the Session

To close out the instance of the driver to free resources, type

```
dmm.Close();
```

Your final program should contain the code below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Agilent.Agilent34401.Interop;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Agilent34401 dmm = new Agilent34401();
            dmm.Initialize("GPIB::23", false, true, "simulate=true");
            dmm.DCVoltage.Configure(1.5, 0.001);
            dmm.Trigger.Delay = 0.01;
            double reading;
            reading = dmm.Measurement.Read(1000);
            Console.WriteLine("The measurement is {0}",
reading);
            Console.ReadLine();
            dmm.Close();
        }
    }
}
```

## Build and Run the Application

Build your application and run it to verify it works properly.

- 1 From the Build menu, click the name of your Console Application.
- 2 From the Debug menu, click Start Debugging.

## Tips

The code for a Visual Basic console application in Visual Studio 2010 is almost identical to the C# application:

```
Option Explicit On
Imports Agilent.Agilent34401.Interop
Module Module1
    Sub Main()
        Dim dmm As New Agilent34401
        dmm.Initialize("GPIB::23", False, True,
"simulate=true")
        dmm.Function =
Agilent34401FunctionEnum.Agilent34401FunctionDCVolts
        dmm.DCVoltage.Configure(1.5, 0.001)
        dmm.Trigger.Delay = 0.01
        Dim reading As New Double
        reading = dmm.Measurement.Read(1000)
        dmm.Close()
        Console.WriteLine("The reading is {0}", reading)
        Console.ReadLine()
    End Sub
End Module
```

The main differences include the following:

- To use Visual Basic, select Visual Basic in Project Types.
- To enforce type checking, insert a line at the start of the code. Type `Option Explicit On`
- This example also shows how to set an enumerated property. This property assignment sets the DMM function to Voltage: Type

```
dmm.Function =
Agilent34401FunctionEnum.Agilent34401FunctionDCVolts
```

- To dimension a variable for the instrument and reading, use `Dim dmm` and `Dim reading`.

### Further Information

- Learn more about Visual C# at <http://msdn.microsoft.com/vcsharp/>.
- Learn more about Visual Basic at <http://msdn.microsoft.com/vbasic/>.

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