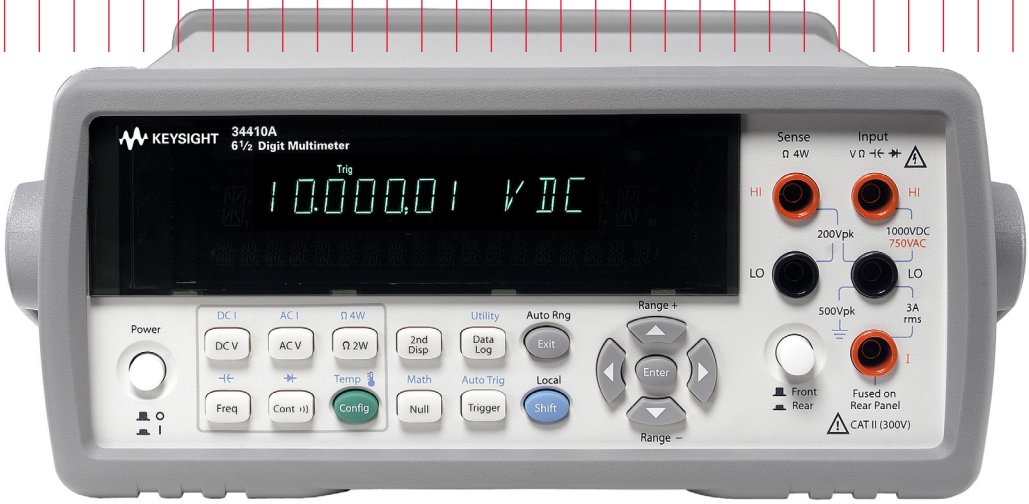
Keysight Technologies

Using .NET Methods to Add Functionality to IVI-COM Drivers

Access a deeper set of instrument functionality with minimal programming

Application Note



KEYSIGHT TECHNOLOGIES

Introduction

Today, most test equipment is supported with a driver, and all LXI instruments include IVI drivers. However, for practical reasons most drivers cover only a subset of an instrument's functionality and, in some cases, the omitted functionality is needed to accomplish a required measurement or set of tests. Although it's possible to add functionality by modifying a driver's source code, this requires advanced programming skills, and may consume more time than a project schedule allows.

An attractive alternative is .NET and the .NET application programming interface (API) that most IVI-COM drivers provide. (Keysight Technologies provides IVI-COM drivers for most newer instruments).

The API makes it possible to add functionality to the driver executable, and to create a set of "standard" additions that will meet your present and future needs. Through two methods—"extension" and "inheritance"—this approach makes it faster and easier to access more of an instrument's functionality during automated testing.

If software reuse and test-system portability are important to your organization, the use of drivers can be advantageous. This is especially true if you are migrating from GPIB to LAN and LXI. This application note will help you enhance these benefits by adding functionality to existing IVI-COM drivers through the extension and inheritance methods.

Looking inside IVI-COM drivers

IVI-COM drivers are based on the Microsoft Common Object Model (COM) and were standardized in 2002 by the IVI Foundation. Today, many test instruments—and all Keysight LXI devices—have an IVI-COM driver that can be used for programmatic control. These drivers include three key elements: source code, a set of examples for a variety of application development environments (ADEs), and a .NET Primary Interop Assembly (PIA) that allows test programs to easily call IVI-COM drivers. Let's take a closer look at each of these.

Source code: This is the ultimate reference for a driver's behavior. Vendors supply source code for three main reasons:

- To show users how the driver implements it functionality.
- To let users enhance the driver by adding functionality.
- To let users fix any defects they may discover in the driver. (Not recommended for most users due to the complexities involved).

The process of unraveling and understanding source code can be daunting. One solution is IO Monitor, a tool included with the Keysight IO Libraries Suite. IO Monitor lets you observe how the IVI-COM driver communicates with the instrument.

Example programs: These demonstrate how to use and access a driver's functionality in a variety of ADEs. For instance, the driver for the Keysight 34410A digital multimeter (DMM) includes examples for C++, C#, Visual Basic 6.0, Visual Basic .NET (VB.NET) and Keysight VEE. In this application note, we focus on C# and VB.NET as the basis for extending drivers.

.NET PIAs: These get their name for two reasons. First, they allow a language to interoperate with .NET. Second, they are "primary" because they are provided by the driver developer who, in theory, knows exactly how the interoperation should behave. PIAs are important because a program—C#, VB.NET or otherwise—will not automatically call IVI-COM drivers, which are created in C++ COM code. The PIA does the necessary translation, providing an easy interface to IVI-COM drivers and making it possible to add functionality through an easy-to-use language such as C# or VB.NET.

Evaluating an existing driver

The decision to expand driver functionality depends on the limitations of the existing driver:

- **Performance:** An IVI-COM driver may not take full advantage of instrument speed.
- **Functionality**: As noted earlier, few drivers implement an instrument's entire feature set.
- **Defects**: In some cases, the fastest way to remedy a defect is to create your own fix.
-

If any of these issues are affecting the performance or functionality of your test system, then it will be worthwhile to enhance the IVI-COM driver. This can be done through source code, but is much easier with .NET and IVI-COM PIAs along with the inheritance and extension methods. A few examples will help illustrate these techniques.

Preparing for the examples

As a starting point, this note assumes that you are using Visual Studio. More specifically, the inheritance example assumes that you are using Visual Studio 2008; however, this example can also be developed in other versions of Visual Studio, with slight modifications to account for differences between versions. The extension example requires Visual Studio 2008 or higher because Visual Studio 2008 was the first version to support extension methods.

The following items must also be installed on your PC: the IVI Shared Components, a VISA-COM library such as Keysight IO Libraries Suite, and the 34410A IVI-COM instrument driver. The IO Libraries Suite is available for Keysight customers at **www.Keysight.com/find/iolib**. Keysight instrument drivers are available at **www.Keysight.com/find/drivers**.

To run the example program with a real instrument attached, you will need a 34410 DMM with firmware revision 2.35 or higher. If an instrument is not available, you can specify "Simulate=true" in the Initialize call.[1]

Both the inheritance- and extension-method examples require some preparation in Visual Studio. The first step is to create a solution called App34410 and three projects within that solution:

- **App34410**: A C# console application that will let you try out the enhancements.
- **Inherit34410**: A C# class library that extends the 34410 driver using inheritance.
- **Extend34410**: A C# class library that extends the 34410 driver using extension methods.

After creating these projects, you will add references to the .NET assemblies to be used in each project, and make a few easy-to-implement changes to the driver source code.

Creating the example solution and projects

To begin, open Visual Studio 2008 and create a new C# console application project called **App34410**. This project will hold the client code that is used to test the enhanced driver.

- Select **File** > **New...** > **Project** from the main menu (Figure 1). This opens the **New Project** dialog.
- In the **New Project** dialog, select **Visual C#** > **Windows** under **Project types**, select **Console Application** under **Templates**, and type **App34410** for the **Name**.
- Note: We recommended that you check the **Create directory for solution** box. If needed, change the **Location** to a suitable directory of your choice. Click **OK**. This will create a new **App34410** project and solution, display the solution hierarchy in the **Solution Explorer** pane, and open the default class library file **Program.cs** in the C# code editor.
- Click **Program.cs** in the Solution Explorer, so the properties for **Program.cs** are displayed in the **Properties** pane. Change the **File Name** property to **App34410.cs.**

In the same solution, create a new C# class library project called.

Inherit34410.

- In the Solution Explorer pane, right click **Solution 'App34410.'** Select **Add** > **New Project...** from the drop down menu. This opens the **New Project** dialog.

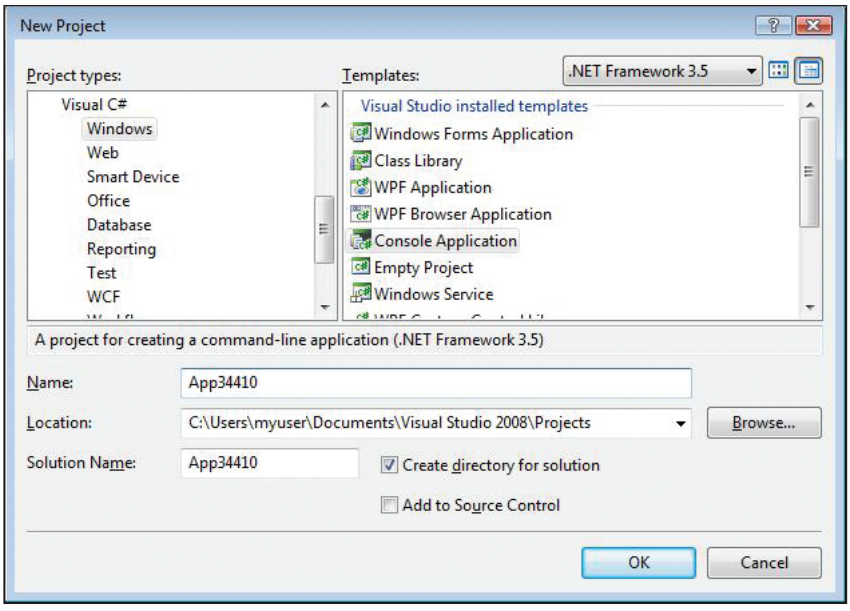


Figure 1.

1. The initialize calls are in App34410.cs in the Main method. Please see the code listings in the "Utilizing the new functionality" section of this note.

- In the **New Project** dialog, select **Visual C# > Windows** under **Project types**, select **Class Library** under **Templates**, and type **Inherit34410** for the **Name**. Check the **Create directory for solution** box and change the **Location** to a suitable directory. Click OK. This will create a new **Inherit34410** project, add the project to the solution hierarchy in the Solution Explorer pane, and open the default class library file **Class1.cs** in the C# code editor.
- In the **Properties** pane for **Class1.cs**, change the **File Name** property to **Inherit34410.cs**.

In the same solution, create a new C# class library project called **Extend34410** using the same procedure that you used to create the **Inherit34410** project. The **Inherit34410** and **Extend34410** projects extend the driver using inheritance and extension methods, respectively.

Adding the assembly references

The next step is to add references to the **Inherit34410** and **Extend34410** projects. In particular, you need references to the 34410 IVI-COM driver and two supporting IVI libraries, IviDriver and IviDmm.

- In the Solution Explorer pane under the **Inherit34410** project, right click on **References** and the select **Add Reference...** from the dropdown menu. After a short delay, the **Add Reference** dialog will appear.
- Select the **Browse** tab and then browse to the IVI Primary Interop Assemblies directory. The default location for this directory is C:\Program Files\IVI Foundation\IVI\Bin\Primary Interop Assemblies.
- Select **Keysight.Keysight34410.Interop.dll** and click **OK**. The project should now contain a reference to **Keysight.Keysight34410.Interop**.
- Repeat the previous step to add references to **Ivi.IviDriver.Interop.dll** and **Ivi.IviDmm.Interop.dll** to the project.
- Add a reference to **Ivi.Visa.Interop.dll**. The default location for the VISA-COM PIA directory is C:\Program Files\IVI Foundation\VISA\VisaCom\Primary Interop Assemblies. This will enable using the driver to send Standard Commands for Programmable Instruments (SCPI) commands and read query results directly from the instrument.
- Finally, add the same three references to the **Extend34410** project.

Next, add references to the **App34410** project. References to both extension class libraries are needed.

- In the Solution Explorer pane, under the **App34410** project, right click on **References** and the select **Add Reference...** from the dropdown menu. After a short delay, the **Add Reference** dialog will appear.
- Select the **Projects** tab, and click on the **Inherit34410** assembly, then click **OK**.
- Repeat for the **Extend34410** project.
  The **App34410** project should now contain references to both **Inherit34410** and **Extend34410**.
- Add the references for **Keysight.Keysight34410.Interop.dll**, **Ivi.IviDriver.Interop.dll** and **Ivi.IviDmm.Interop.dll** as described above.

Revising the source code

Add the following lines to the top of **App34410.cs**: (also see Figure 2)

```
using Inherit34410;
using Extend34410;
using Ivi.Dmm.Interop;
using Ivi.Driver.Interop;
using Keysight.Key-
  sight34410.Interop;
```

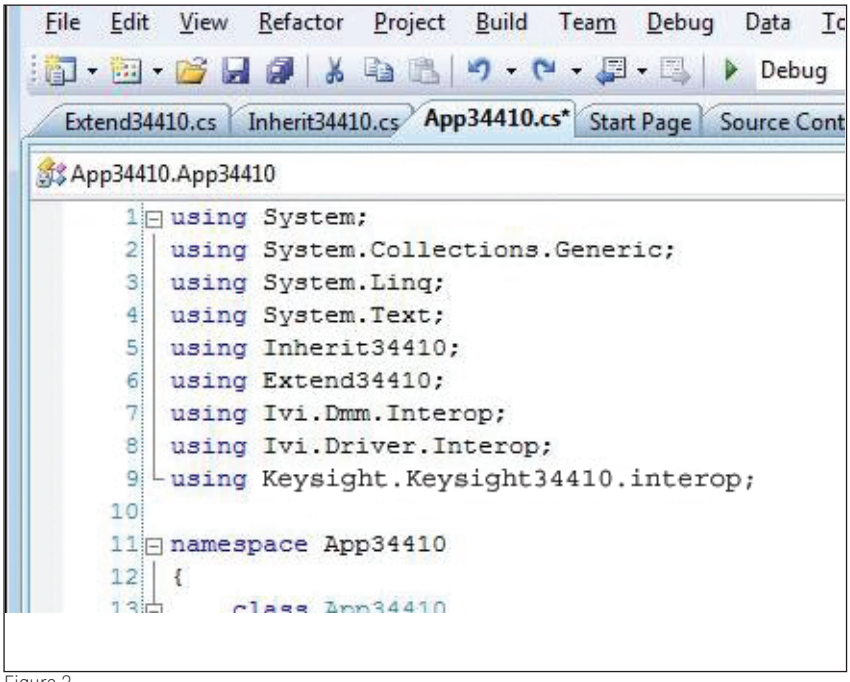


Figure 2.

Add the following lines to the top of both **Inherit34410.cs** and **Extend34410.cs**:

```
using Ivi.Visa.Interop;
using Ivi.Dmm.Interop;
using Ivi.Driver.Interop;
using Keysight.Key-
  sight34410.Interop;
```

After completing all of the steps described above, you will have created the infrastructure necessary to proceed with the examples. To check for incidental errors, you can build the solution by selecting **Build > Rebuild Solution** from the main menu.

Choosing inheritance or extension

Before exploring the examples, it will be worthwhile to take a closer look at inheritance and extension. The choice of one method versus the other depends on a few key characteristics of each. For example, inheritance has three distinguishing characteristics:

- A new class is created, and it has a new name. From this, it is always clear that you are not using the original driver; however, source code written to use the original driver must be revised to access the inheriting driver.
- Properties can be overridden using inheritance.
- Creating a series of new capabilities is awkward with inheritance. For example, if two people create a set of extensions, they will each create a new driver that inherits from the original. The two new drivers cannot be used simultaneously. Instead, you must either choose one or create a third enhanced driver that combines the two.

Extension methods also have three distinguishing characteristics:

- Because no new class is created, there is no need to change the source code for instantiating the driver or calling other driver methods. The drawback: It may not always be clear when you are using methods implemented in the original driver versus those you've added.
- Extension methods must be actual methods, meaning there are no "extension properties." Properties must be implemented as "set" or "get" methods.
- Adding a series of new capabilities is straightforward with extension methods. If two people create independent sets of methods, these can be combined as long as there are no conflicts in their names or signatures. If each is created as a new assembly of methods that extend the original driver, both sets of extensions can be used simultaneously by simply adding references to both assemblies in your .NET code.

Definitions of key terms

To ensure clarity in the explanations and examples, here are definitions of nine key terms from object-oriented programming (OOP):

- **assembly**: in the context of .NET, this is a partially compiled code library containing one or more modules (files)
- **class**: represents a programmatic concept and encapsulates the state (through variables) and behavior (through methods) of that concept
- **extension**: expands the executable form of a class without modifying its underlying code
- **inheritance**: creates a new class based on a previously defined class; the new one inherits the attributes and behavior of the original
- **instance**: an individual object that is created based on a specific class
- **instantiation**: the creation of an object (instance) based on a class
- **method**: a subroutine that performs a specific action; associated with a specific class or object
- **object**: another term for a variable in a computer program; each individual object is a specific instantiation of a class
- **property**: a class value that can be retrieved or set (programmatically or from user input)

Applying the inheritance method

The concept of inheritance allows you to create a new class by "inheriting" an existing class and then adding new functionality. This example extends the 34410 IVI-COM driver by creating a new C# class called **Inherit34410**, which inherits from the root 34410 driver class, and then adding a few new functions.

In **Inherit34410.cs**, the Inherit34401 class must inherit from Keysight-34410Class, the root class of the 34410 driver. Modify the class statement in **Inherit34410.cs** as follows:

```
public class Inherit34410Class : Keysight34410Class
```

For clarity, error handling has been omitted in the examples that follow.

Improving performance with inheritance

The 34410 IVI-COM driver strikes a good balance between speed and resolution. However, the driver does not default to a maximum-resolution measurement. In some cases it may be useful to provide a method that forces high-resolution measurements. Add the following method to Inherit34410Class in **Inherit34410.cs** to get the highest possible resolution measurement of DC voltage:

```
public double ReadHighResDCVolts()
{
    // Configure for DC volts
    this.MeasurementFunction =
        Keysight34410MeasurementFunctionEnum.
  Keysight34410MeasurementFunctionDCVoltage;

    // Configure for maximum NPLC. Note that NPLC MAX is
  not explicitly
    //   supported by the driver, so driver IO is used.
    this.IO.WriteString("SENS:VOLT:DC:NPLC MAX", true);

    // Set trigger count to 1, sample count to 1, trig-
  ger source to immediate,
    //   sample timer to 0 - essentially reset the
  trigger.
    this.MultiPoint.Configure(1, 1,
        IviDmmSampleTriggerEnum.IviDmmSampleTriggerIm-
  mediate, 0);

    // Read a measurement. (Maximum NPLC takes a long
  time & large timeout.)
    double reading = this.Read(10000);

    return reading;
}
```

Fixing a defect with inheritance

The 34410 IVI-COM driver is robust: Tests by Keysight have not revealed any defects that would cause the driver to behave improperly (i.e., it behaves according to reasonable design assumptions). To simulate a defect, pretend that the TriggerDelay command uses the dimensions of milliseconds rather than seconds. To program this, modify Inherit-34410Class in **Inherit34410.cs** with the following property, which performs conversion and then sets the value correctly:

```
public double MyTriggerDelay
{
    set { this.Trigger.TriggerDelay = value * 1000; }
    get { return this.Trigger.TriggerDelay / 1000; }
}
```

Adding measurement-specific functionality with inheritance

Users of the 34410 IVI-COM driver may choose to build a library of functions that extends the driver to meet specific measurement requirements. One highly useful addition would be to digitize a waveform from the 34410. To do this, add the following method to Inherit34410Class in **Inherit34410.cs**:

```
public double[] DigitizeDCWaveform(double range, int points)
{
    // Configure for DC volts, set manual range, and fastest
  resolution
    this.DCVoltage.Configure(range,
        Keysight34410ResolutionEnum.
  Keysight34410ResolutionLeast);

    // Set trigger count to 1, sample count to 'points', trig-
  ger source to
    //   immediate, sample timer to 0
    this.MultiPoint.Configure(1, points,
        IviDmmSampleTriggerEnum.IviDmmSampleTriggerImmediate,
  0);

    // Turn off autozero
    this.DCVoltage.AutoZero =
        Keysight34410AutoZeroEnum.Keysight34410AutoZeroOff;

    // Set the trigger delay to 0
    this.Trigger.TriggerDelay = 0;

    // Format the data for 32-bit binary
    this.DataFormat.DataFormat =
        Keysight34410DataFormatEnum.
  Keysight34410DataFormatReal32;

    // Initialize the instrument
    this.Initiate();

    // Wait for the measurements to complete
    this.WaitForOperationComplete(points / 10);

    //Remove Data
    double[] readings = this.RemoveReadings(points);

    return readings;
}
```

Applying the extension method

Extension methods allow a developer to extend an existing class by adding methods that appear to be methods on the original class. No new class is required. In this example, we will extend the Keysight 34410 IVI-COM driver by creating a few new extension methods on the root Keysight 34410 driver class.

In **Extend34410.cs**, the class Extend34410Class must be static. This is a requirement for classes that implement new extension methods. Modify the class statement in **Extend34410.cs** as follows:

```
public static class Extend34410Class
```

Adding missing functionality with extension

This example replicates the two methods added through inheritance, but with two key differences in the code. One is in the first parameter of each method, which identifies the extended class. The declaration of the first parameter consists of three key items: the keyword “this”; the type that the method is extending (it must be a class or structure, in this case **Keysight34410Class**); and the name (“driver”) to be used in the code for the instance of the class. The other key difference is in the keyword “this,” which isn’t used in the bodies of the methods. Instead, it is replaced with the name of the instance of the class being extended, in this case “driver.”

To implement the missing instrument commands, add the following two methods to Extend34410Class in **Extend34410.cs**:

```
public static void SetToRemote(this Keysight34410Class driver)
{
    driver.IO.WriteString(“DIAG:REM”, true);
}

public static void SetToLocal(this Keysight34410Class driver)
{
    driver.IO.WriteString(“DIAG:LOCAL”, true);
}
```

Improving performance with extension

As above, this example achieves the same basic functionality as that added using inheritance. Notice the differences in the code. To perform a DC voltage measurement with the highest possible resolution, add the following method to **Extend34410.cs**:

```
public static double ReadHighResDCVolts(this Keysight34410Class driver)
{
    // Configure for DC volts
    driver.MeasurementFunction =
        Keysight34410MeasurementFunctionEnum.Keysight34410MeasurementFunctionDCVoltage;

    // Configure for maximum NPLC. Note that NPLC MAX is not explicitly
    //   supported by the driver, so driver I/O is used.
    driver.IO.WriteString("SENS:VOLT:DC:NPLC MAX", true);

    // Set trigger count to 1, sample count to 1, trigger source to immediate,
    // and sample timer to 0, which, in effect, resets the trigger
    driver.MultiPoint.Configure(1, 1,
        IviDmmSampleTriggerEnum.IviDmmSampleTriggerImmediate, 0);

    // Read a measurement (maximum NPLC takes a long time & large timeout)
    double reading = driver.Read(10000);

    return reading;
}
```

Fixing a defect with extension

This adds the same basic functionality as in the inheritance example. (Recall that this is not a known defect in the driver; it is simply used for illustration).

Here, the extension must be implemented using methods because there is a required parameter (the trigger delay). Because of this, it is not possible to create a property to change the implementation. Instead, it requires the use of two methods, SetMyTriggerDelay() and GetMyTriggerDelay(), that do the same thing. Add the following methods to Extend34410Class in **Extend34410.cs** to "correct" the trigger delay:

```
public static void SetMyTriggerDelay(this Keysight-
  34410Class driver, double value)
{
    driver.Trigger.TriggerDelay = value * 1000;
}

public static double GetMyTriggerDelay(this Keysight-
  34410Class driver)
{
    return driver.Trigger.TriggerDelay / 1000;
}
```

Adding measurement-specific functionality with extension

This example adds the same basic functionality added through inheritance (once again, note the differences in the code). Add the following method to **Extend34410.cs**:

```
public static double[] DigitizeDCWaveform(this Keysight34410Class driver,
                        double range, int points)
{
    // Configure for DC volts, set manual range and fastest resolution
    driver.DCVoltage.Configure(range,
        Keysight34410ResolutionEnum.Keysight34410ResolutionLeast);

    // Set trigger count to 1, sample count to 'points', trigger source to
    //   immediate and sample timer to 0
    driver.MultiPoint.Configure(1, points,
        IviDmmSampleTriggerEnum.IviDmmSampleTriggerImmediate, 0);

    // Turn off autozero
    driver.DCVoltage.AutoZero =
        Keysight34410AutoZeroEnum.Keysight34410AutoZeroOff;

    // Set the trigger delay to 0
    driver.Trigger.TriggerDelay = 0;

    // Format the data for 32-bit binary
    driver.DataFormat.DataFormat =
        Keysight34410DataFormatEnum.Keysight34410DataFormatReal32;

    // Initialize the instrument
    driver.Initiate();

    // Wait for the measurments to complete
    driver.WaitForOperationComplete(points);

    //Remove data
    double[] readings = driver.RemoveReadings(points);

    return readings;
}
```

Using the new functionality

App34410.cs will use the driver and the extensions to access the instrument. First, add two lines to the end of the Main method: These will keep the content of the console window visible until you hit a key to indicate that you are done.

```
Console.WriteLine("Press any key to end program.");
Console.ReadKey();
```

Next, add the code to access the instrument using the new driver created through inheritance from the Keysight34410 driver. This code goes in the Main method:

```
Inherit34410Class iDriver = new Inherit34410Class();

iDriver.Initialize("TCPIP0::A-34411A-00126.lvld.keysight.com::inst0::INSTR",
                   false, true, "");
Console.WriteLine("Set to local");
iDriver.SetToLocal();
Console.WriteLine("Set to remote");
iDriver.SetToRemote();
Console.WriteLine("High resolution DC Voltage Reading: " +
                  iDriver.ReadHighResDCVolts().ToString());
iDriver.MyTriggerDelay = 0.02;
Console.WriteLine("My Trigger Delay: " + iDriver.MyTriggerDelay.ToString());

double[] readings = iDriver.DigitizeDCWaveform(10, 50);
Console.WriteLine("DC Waveform (points 1-10)");
for (int idx = 0; idx <= 10; idx++)
{
    Console.WriteLine("   " + readings[idx].ToString());
}

iDriver.Close();
```

Note that the instantiated class is Inherit34410Class, not Keysight34410Class. Inherit-34410Class inherits the entire API of the Keysight34410 and has the methods and properties added above as well. You should be able to run this code and see output that looks something like this:

```
Set to local
Set to remote
High resolution DC Voltage Reading: 1.23456+E2
My Trigger Delay: 0.019999
DC Waveform (points 1-10)
0.012345
…
0.012345
```

Add the code to access the instrument using the new driver created by adding extension methods to the Keysight34410 driver. This code is also added to the main method:

```
Keysight34410Class eDriver = new Keysight34410Class();

eDriver.Initialize("TCPIP0::A-34411A-00126.lvld.keysight.com::inst0::INSTR",
                   false, true, "");
Console.WriteLine("Set to local");
eDriver.SetToLocal();
Console.WriteLine("Set to remote");
eDriver.SetToRemote();
Console.WriteLine("High resolution DC Voltage Reading: " +
                  eDriver.ReadHighResDCVolts().ToString());
eDriver.SetMyTriggerDelay(0.02);
Console.WriteLine("My Trigger Delay: " + eDriver.GetMyTriggerDelay().ToString());

double[] readings2 = eDriver.DigitizeDCWaveform(10, 50);
Console.WriteLine("DC Waveform (points 1-10)");
for (int idx = 0; idx <= 10; idx++)
{
    Console.WriteLine("   " + readings2[idx].ToString());
}

eDriver.Close();
```

Notice that the only differences between the output of this code and the previous example are in the readings. The only difference in the code itself, aside from a few variable names, is in the direct instantiation of the Keysight34410Class driver. While this is truly the case, the extension methods are associated with the driver through the "using Extend34410" declaration at the top of the file.

Additional considerations and limitations

Adding functionality to a driver using either extension or inheritance is limited to amendments to what the driver executable already does. Because neither method involves editing or rebuilding the driver source code, it isn't possible to change the underlying way the driver operates. As a result, there are some limitations on what can be done to enhance a driver.

One key limitation is associated with drivers that utilize state caching (rare with Keysight IVI-COM drivers).[2] In such a case, the tracked state is invisible and there is no way to access or change it when adding functionality to the driver. As a best practice when using a driver that supports state caching, you should invalidate the driver's cache at the end of any method, whether it is based on inheritance or extension. Another limitation comes from error handling. Instrument drivers perform some level of error detection in their properties and methods, but they rarely query the instrument after every I/O call to check the instrument state. If the level of error handling implemented in a driver is affecting performance in your application, note that it may not be possible to work around this fundamental behavior by simply amending the driver.

As a caveat, leaving driver functionality as-is may produce the best result. For example, it is not advised to use inheritance or extension methods to replace the Initialize() method because it performs so many unseen "housekeeping" activities. If you find it absolutely necessary to amend the Initialize() method, be sure to call the driver's Initialize() method from the new one that is expanding its functionality.

Conclusion

As shown throughout this note, it is possible to add functionality to IVI-COM instrument drivers without resorting to advanced, time-consuming programming techniques. The inheritance and extension methods provide attractive alternatives that will allow you to achieve small or moderate improvements in driver functionality and performance, and to work around driver defects.

2. For instruments of any complexity, state caching is normally impractical for two reasons: relationships between state variables in the firmware are too complex to accurately track in the driver; and some states are dependent on instrument physics so are impossible to track in the driver.

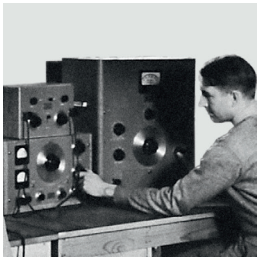
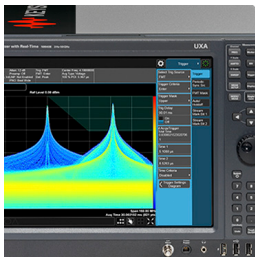
Related Keysight Literature

Table 5.

Publication title	Pub number
Assessing the use of IVI drivers in your test system: Determining when IVI is the right choice	5990-3186EN
Building Hybrid Test Systems, Part 1: Laying the groundwork for a successful transition	5989-8175EN
Building Hybrid Test Systems, Part 2: Ensuring success in two common scenarios	5989-8176EN
Using Linux in Your Test Systems: Linux Basics	5989-6715EN
Using Linux to Control LXI Instruments Through VXI-11	5989-6716EN
Using Linux to Control LXI Instruments Through TCP	5989-6717EN
Using Linux to Control USB Instruments	5989-6718EN
Tips for Optimizing Test System Performance in Linux Soft Real-Time Applications	5989-6719EN
LXI: Going Beyond GPIB, PXI and VXI, Overcoming the major challenges of testing	5989-4371EN

Evolving Since 1939

Our unique combination of hardware, software, services, and people can help you reach your next breakthrough. We are unlocking the future of technology.
From Hewlett-Packard to Agilent to Keysight.


myKeysight

myKeysight
www.keysight.com/find/mykeysight
A personalized view into the information most relevant to you.

http://www.keysight.com/find/emt_product_registration
Register your products to get up-to-date product information and find warranty information.

KEYSIGHT SERVICES
Accelerate Technology Adoption.
Lower costs.

Keysight Services
www.keysight.com/find/service
Keysight Services can help from acquisition to renewal across your instrument's lifecycle. Our comprehensive service offerings—one-stop calibration, repair, asset management, technology refresh, consulting, training and more—helps you improve product quality and lower costs.

CALIBRATION UP TO 10 YEARS WARRANTY
ASSURANCE PLANS

Keysight Assurance Plans
www.keysight.com/find/AssurancePlans
Up to ten years of protection and no budgetary surprises to ensure your instruments are operating to specification, so you can rely on accurate measurements.

Keysight Channel Partners
www.keysight.com/find/channelpartners
Get the best of both worlds: Keysight's measurement expertise and product breadth, combined with channel partner convenience.

www.keysight.com/find/open

For more information on Keysight Technologies' products, applications or services, please contact your local Keysight office. The complete list is available at:
www.keysight.com/find/contactus

Americas

Canada	(877) 894 4414
Brazil	55 11 3351 7010
Mexico	001 800 254 2440
United States	(800) 829 4444

Asia Pacific

Australia	1 800 629 485
China	800 810 0189
Hong Kong	800 938 693
India	1 800 11 2626
Japan	0120 (421) 345
Korea	080 769 0800
Malaysia	1 800 888 848
Singapore	1 800 375 8100
Taiwan	0800 047 866
Other AP Countries	(65) 6375 8100

Europe & Middle East

Austria	0800 001122
Belgium	0800 58580
Finland	0800 523252
France	0805 980333
Germany	0800 6270999
Ireland	1800 832700
Israel	1 809 343051
Italy	800 599100
Luxembourg	+32 800 58580
Netherlands	0800 0233200
Russia	8800 5009286
Spain	800 000154
Sweden	0200 882255
Switzerland	0800 805353 Opt. 1 (DE) Opt. 2 (FR) Opt. 3 (IT)
United Kingdom	0800 0260637

For other unlisted countries:
www.keysight.com/find/contactus
(BP-9-7-17)

DEKRA Certified
ISO9001 Quality Management System

www.keysight.com/go/quality
Keysight Technologies, Inc.
DEKRA Certified ISO 9001:2015
Quality Management System

KEYSIGHT TECHNOLOGIES

Published in USA, December 2, 2017
5990-3661EN
www.keysight.com