Getting Test Programs Up and Running Quickly with Driver Tracing and I/O Monitoring

Application Note

Examining and troubleshooting I/O activity in IVI-COM drivers

The decision to use IVI-COM drivers may be influenced by a variety of factors. These drivers are easy to use in .NET languages, Visual Basic 6, and C++, and the integrated IntelliSense functionality provides built-in help for developers. IVI-COM drivers also enable software reuse and enhance test system portability. Although some IVI-COM drivers don't access all of an instrument's functionality, they are easy to extend with .NET languages (for more on this, please refer to the Agilent application note, *Using .NET to extend IVI-COM Instrument Drivers*, publication number 5990-3661EN).

These are compelling advantages if you are confident in the implementation and performance of the driver. In the search for that confidence, test engineers face a consistent problem: determining how the driver communicates with the instrument. Which Standard Commands for Programmable Instruments (SCPI) commands are used? Does the driver check the instrument for errors? How does it handle asynchronous communication? What happens inside the driver when there are errors or unexpected results? While many engineers look to driver source code for answers, the fundamental question is, what does the driver **actually do?**

To create a window into driver operation, Agilent provides two highly useful tools: driver tracing and IO Monitor. Driver tracing writes a detailed history of the specific calls made to the driver for later analysis. IO Monitor provides a real-time window into the calls made to Agilent VISA.

Driver tracing is built into nearly all Agilent IVI-COM drivers. When driver tracing is enabled, the driver writes an XML file that lists the driver methods and properties called by the test program, along with parameter values. The XML file can be easily viewed in your browser of choice after the test program has completed.

IO Monitor is an application that displays a system-wide, real-time list of input/output (I/O) calls to Agilent VISA, VISA-COM and SICL I/O libraries. It is included in the Agilent IO Libraries Suite.

These tools help test engineers in at least three ways:

- Learn more about how to program an instrument.
- Verify that the program is calling a driver as expected.
- Troubleshoot when the program is not working as expected.

The result is an enhanced ability to get test programs up and running – and producing the expected results – in less time. On the following pages, this note helps you create an example program and then shows you how to use driver tracing and IO Monitor to examine, verify and troubleshoot I/O activity in IVI-COM drivers.

1. *And many IVI-COM drivers from other vendors.*

Agilent Technologies

Essential Background

In building a test program, you need to assemble the correct set of drivers, which includes all instrument drivers as well as any required I/O drivers. For this application note, the examples use Agilent's VISA or VISA-COM I/O libraries, which provide the I/O drivers to manage the I/O device that connects the computer/operating system to the instrument. The examples also use the IVI-COM driver for instrument control.

Agilent's IO Libraries Suite provides the VISA and VISA-COM I/O libraries as well as several utilities, including IO Monitor, that are used later in this note. The driver-tracing capability is part of most Agilent IVI-COM drivers.

How Drivers Communicate

Agilent's IVI-COM drivers use either VISA or VISA-COM to communicate with instruments. VISA and VISA-COM are used to send SCPI commands to the instrument and read back the instrument's results independent of the interface—GPIB, LAN, USB and so on. The driver is notified of asynchronous I/O events by VISA or VISA-COM. With all of this activity, it's clear that visibility into VISA and VISA-COM calls is essential to understanding how a driver does its job.

It's worth noting that VISA and VISA-COM are industry standards that were originally created by the VXI*plug&play* Systems Alliance and are now maintained by the IVI Foundation. Several test and measurement vendors provide implementations of VISA and VISA-COM. Agilent's implementation, which is referred to as Agilent VISA or Agilent VISA-COM in this note, can be used by installing the Agilent IO Libraries Suite.

Preparing the Example Program

Throughout the rest of this note, we will examine the information that is delivered by driver tracing and IO Monitor in the context of a short example. Our starting point is a short program that first makes a connection to the instrument and then closes the connection. We will then extend the example with additional driver features and examine them through both driver traces and IO Monitor.

To follow along with the example programs, you'll need the following on your PC: Microsoft® Visual Studio® 2003, 2005, or 2008; the IVI Shared Components; a VISA-COM library such as Agilent IO Libraries Suite; and the Agilent 34410 IVI-COM instrument driver, v. 1.0.19.0.[1,2] To get a firsthand look at the I/O activity, we recommend that you run the examples with an Agilent 34410 instrument connected.[3]

To get things started, the example requires some upfront preparation of Visual Studio. The required steps are detailed in the following sections.

---

1. *Visual Studio 2008 was used to develop the example code in this note. Visual Studio is Microsoft's programming suite. If you buy a Microsoft compiler (C#, Visual Basic, etc.), the compiler is delivered with Visual Studio.*
2. *Agilent IO Libraries Suite installs the VISA-COM library and IVI Shared Components for you. The Agilent IO Libraries Suite is available at www.agilent.com/find/iolib*
3. *While you can use the 34410 IVI-COM driver in simulation mode, the driver trace information is less meaningful when simulating, and IO Monitor information is not meaningful at all.*

Creating the example solution and projects

To begin, open Visual Studio and create a new C# console application project called **Trace34410**. This project will hold the client code that is used to test the extended driver:

- **Select File | New... | Project** from the main menu. 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 "Trace34410" for the **Name**. We recommend that you check the **Create directory for solution** box. If needed, change the **Location** to a suitable directory for your PC. Click **OK**. This will create a new **Trace34410** 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 on **Program.cs** in the Solution Explorer. The properties for **Program.cs** will be displayed in the **Properties** pane. Change the **File Name** property to **"Trace34410.cs"**.

Adding assembly references

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 select **Add Reference**... from the dropdown menu. After a short delay, the **Add Reference** dialog will appear.
- Select the **COM** tab and add the references for IVI Agilent34410 1.0 Type Library. References to the dependent IviDriver and IviDmm type libraries will be added automatically.

Use the driver's namespace

Add the following line to the top of **Trace34410.cs**:

```
using Agilent.Agilent34410.Interop;
```

After completing these steps, you will have created the infrastructure necessary for the examples. To check for incidental errors, you can build the solution by selecting **Build | Rebuild Solution** from the main menu. If your program does not build, check your syntax and verify that you performed all of the above steps before proceeding.

Using Driver Traces

Most Agilent IVI-COM drivers include the ability to write an XML file of driver-trace messages. The trace includes messages that describe calls to driver functions, and also includes calls that the driver makes to the underlying I/O library, which is either VISA or VISA-COM.

Configuring the driver for tracing

Driver tracing is started when the user calls the driver's Initialize() method with tracing specified in the Option String. To see how this works, add the following lines to the Main method in **Trace34410.cs**:

```
Agilent34410 driver = new Agilent34410Class();

driver.Initialize("TCPIP0::156.140.113.215::inst0:
::INSTR",true, true,
@"DriverSetup=Trace=True,TraceName=C:TEMP\34410Tra
ce.xml");driver.Close();

Console.WriteLine("Press any key to end pro-
gram."); Console.ReadKey();
```

First, replace the resource descriptor in the sample code ("TCPIP0::156.140.113.215::inst0::INSTR") with the resource descriptor of your 34410 instrument. When the driver's initialize method is called, the OptionString parameter includes "DriverSetup=".[1] What follows is a series of name/value pairs that define driver-specific configuration settings. Setting "Trace=True" turns on tracing for the driver session, and setting "Tracename= C:\TEMP\34410Trace.xml" will configure tracing to write messages to the specified file.[2]

Creating the trace file

To create the trace file, build (debug or release) and run the program. Running it in the debugger will work well.

---

1. *If you are using simulation for this example, use the following value for the OptionString parameter: "Simulation=True, DriverSetup=Trace=True,TraceName=C:\TEMP\34410Trace.xml".*
2. *C:\TEMP exists on most Windows PCs. If it doesn't exist on your PC, either create it or modify the code to use the directory of your choice. Note that if you omit the "Tracename= C:\TEMP\34410Trace.xml" in the option string, trace data will be written to a file in the current directory. The default name for trace files created without the Tracename option includes the current date and time, and each time the program is executed, a new file is created.*

Reviewing the trace file

To review the trace file, open your browser and type the following URL:

```
File:\\C:\Temp\34410Trace.xml
```

The trace file references a formatting file, trace.xslt, that is used to format the output. You should see a browser screen that resembles Figure 1. In general, each numbered item in the list is a call to a driver method or property. If the method or property calls an I/O method, that method call is indented under the calling-driver method or property. The first line is the exception: Because tracing was not turned on when Initialize() was called, the Initialize function is not listed. Here is a line-by-line description of the trace file shown in Figure 1:

1. Initialize() was called with the IdQuery parameter = "true". As a result, Initialize() called the I/O method InstrPrintf() to send the SCPI command "*IDN?" to the instrument. In response to this command, the instrument returned an identification string, which is shown as the [Out] parameter bstrResult.

2. Initialize() was called with the Reset parameter = "true". As a result, Initialize() called another driver method, Reset(), which in turn called the I/O method InstrQuery() to send the SCPI command "*RST;*OPC?" to the instrument. This command initiated an instrument reset ("*RST") and waited for the operation to complete ("*OPC?") before returning.

3. Close() was called by the **Trace34410** program.

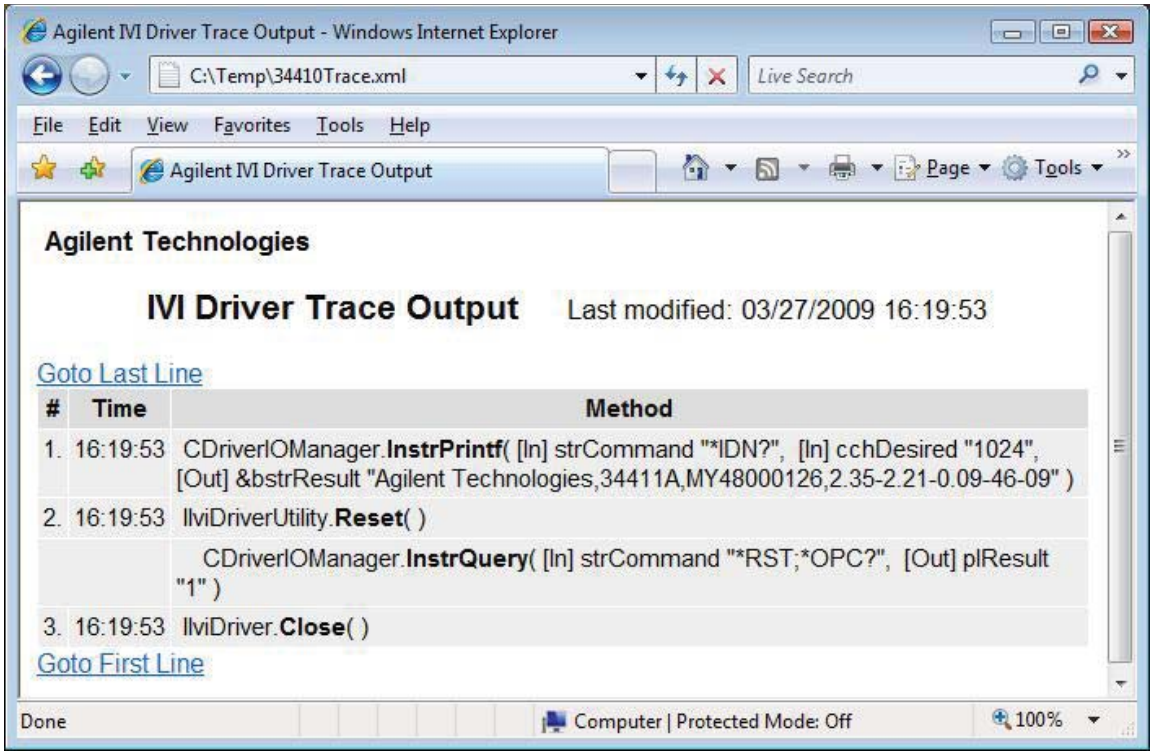

Figure 1. The trace output file provides a numbered sequence of driver I/O calls.

Exploring a complex example

The Agilent 34410 IVI-COM driver generally strikes a good balance between speed and resolution in instrument operations. For example, suppose a client program needs a DC volts measurement with the highest possible resolution. Sending the SCPI string "SENS:VOLT:DC:NPLC MAX" to the 34410 configures it to take a maximum-resolution measurement. Because this measurement takes about 3 to 3.5 seconds, the timeout must also be increased: 5000 ms is a safe amount of time to wait.

The driver has methods to configure and read a measurement. You might reasonably wonder if these methods configure and read a maximum resolution measurement—and the trace messages will answer the question. Here's how: Add the following lines to the **Trace34410** program between the Initialize() method and the Close() method.

```
driver.Voltage.DCVoltage.Configure(10,
Agilent34410ResolutionEnum.
Agilent34410ResolutionLeast);
driver.Trigger.SampleCount = 1;
driver.Trigger.TriggerSource =
Agilent34410TriggerSourceEnum.
Agilent34410TriggerSourceImmediate;
driver.System.TimeoutMilliseconds = 5000;
double[] reading = driver.Measurement.Read();
```

The Configure() method is used because it can set the DMM to take a DC volts measurement. The next parameter is the range and the last is resolution. It's logical to wonder which number to use to set the DC volts resolution to its maximum; however, there is no way to know, given the design of the driver.

Before you run the program, delete the existing trace file, or the new program will append to the old file. Once you have built and run the program, a quick look at the trace will verify that, as suspected, "SENS:VOLT:DC:NPLC MAX" is never sent to the instrument by any of the properties or methods in the line you added.

Line-by-line description

Here is a line-by-line description of the trace file shown in Figure 2.

1. Initialize() was called with the IdQuery parameter = “true”. As a result, Initialize() called the I/O method InstrPrintf() to send the SCPI command “*IDN?” to the instrument. In response to this command, the instrument returned an identification string, which is shown as the [Out] parameter bstrResult.

2. Initialize() was called with the Reset parameter = “true”. As a result, Initialize() called another driver method, Reset(), which in turn called the I/O method InstrQuery() to send the SCPI command “*RST;*OPC?” to the instrument. This command initiated an instrument reset (“*RST”) and waited for the operation to complete (“*OPC?”) before returning.

3. The driver contains an implementation of the IAgilent34410Voltage interface. The “Voltage” in driver.Voltage.DCVoltage.Configure() returns a reference to this interface. “Voltage” in this context is called an “interface reference property.” Interface reference properties are used to create a hierarchy of driver functionality for navigating the driver in Visual Studio using IntelliSense.

4. The Voltage interface contains an implementation of the IAgilent34410DCVoltage interface. The “DCVoltage” in driver.Voltage.DCVoltage.Configure() returns a reference to this interface.

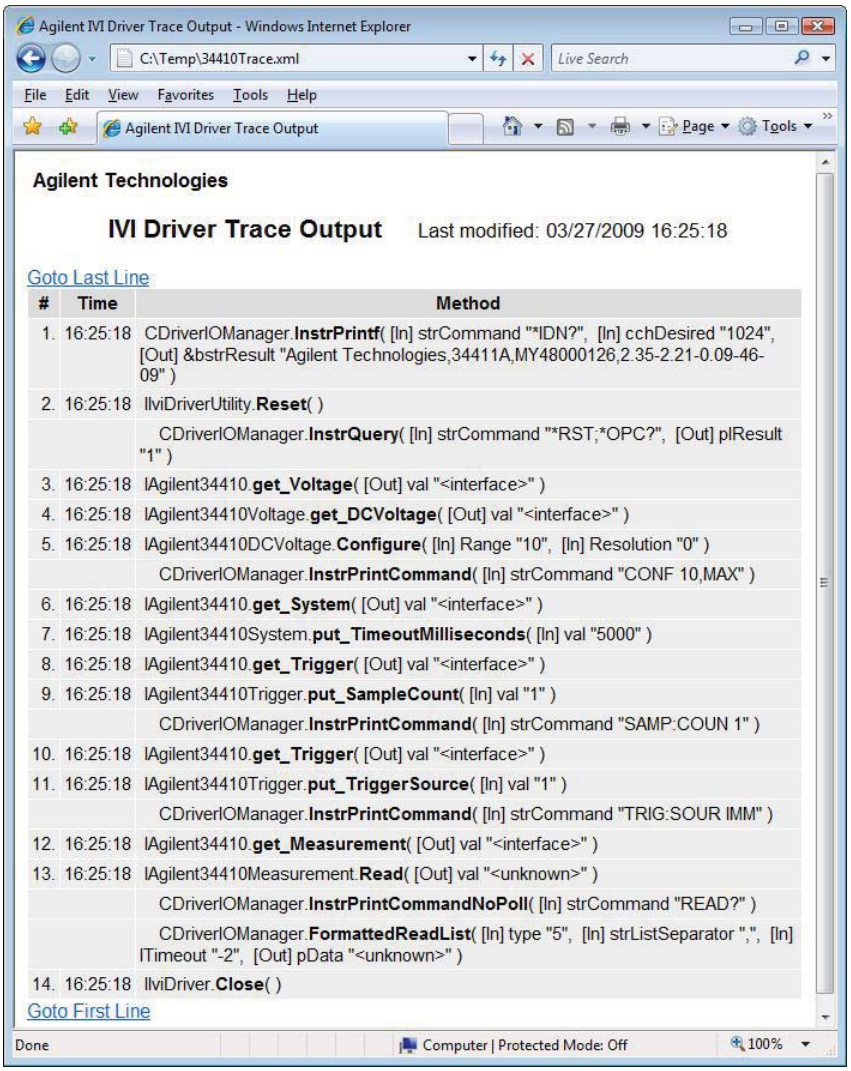

Figure 2. The trace output file for a 34410 measurement.

Line-by-line description, continued

5. Finally, the DCVoltage interface defines the Configure() method that we are calling to configure the 34410 for a DC volts measurement.

6. The driver contains an implementation of the IAgilent34410System interface. The "System" in driver.System.TimeoutMilliseconds returns a reference to this interface.

7. The System interface defines the TimeoutMilliseconds property. The program calls the property's put_TimeoutMilliseconds method and sets the timeout to 5000 milliseconds. This ensures that the driver has allowed enough time for a maximum-resolution measurement, if that's what the driver does when running our sample code.

8. The driver contains an implementation of the IAgilent34410Trigger interface. The "Trigger" in driver.Trigger.TriggerSource returns a reference to this interface.

9. The Trigger interface defines the TriggerSource property. The program calls the property's put_TriggerSource method to set the trigger source to "Immediate". This ensures that the measurement will be taken as soon as the driver starts the Read operation.

10. The driver contains an implementation of the IAgilent34410System interface. The "System" in driver.System.TimeoutMilliseconds returns a reference to this interface.

11. The System interface defines the TimeoutMilliseconds property. The program calls the property's put_TimeoutMilliseconds method to set the timeout to 5000 ms. Once the driver starts the read, the VISA will wait 5000 ms; if a result hasn't been received, it will return control to the test program. This ensures that errors don't make the program wait indefinitely.

12. The driver contains an implementation of the IAgilent34410Measurement interface. The "Measurement" in driver.Measurement.Read() returns a reference to this interface.

13. The Measurement interface defines the Read() method. The program calls the read method and queries the DMM for the measurement. (Note that the trace does not show numeric array results to instrument queries to avoid formatting and performance issues.)

14. Close() was called by the **Trace34410** program.

Understanding the limitations of driver tracing

For the examples shown above, analyzing driver traces can be useful. However, this method has an important limitation: Only methods and properties that are implemented as part of the driver will write messages to the trace file. This can be illustrated by using the driver's IO passthrough feature to send "SENS:VOLT:DC:NPLC MAX" to the instrument. Add the following line after the Configure() method in the example above:

```
driver.System.IO.WriteS
tring("SENS:VOLT:DC:NP
LC MAX", true);
```

After you build and run the program, note that the NPLC MAX command doesn't show up in the trace. What you will see is a message from IAgilent34410System.get_IO(), which corresponds to the "IO" in driver.IO.WriteString. In this case, "IO" refers to the IVI Foundation's VISA-COM Formatted IO class. This Formatted IO class implements the WriteString method. Because the Formatted IO class was developed by the IVI Foundation rather than the driver vendor, you won't see a trace message from WriteString, and hence "SENS:VOLT:DC:NPLC MAX" doesn't appear in the trace.

Using IO Monitor

Both Agilent VISA and Agilent VISA-COM are capable of sending "trace" messages to IO Monitor, which is also installed with Agilent IO Libraries Suite. IO Monitor is an application that displays these messages in a sequence that corresponds to the time they were generated.

Running IO Monitor

IO Monitor can be run from the start menu: Select **Agilent IO Libraries Suite | Utilities | IO Monitor**. Once you have started IO Monitor, select **Options**... from the toolbar. In the section titled **Monitor/Display Messages From Sources**, make sure that both the **Monitor** and **Display** check boxes for Agilent VISA are checked, then select **OK**. To keep the display as uncluttered as possible, make sure that no other boxes are checked. This configures IO Monitor to display only Agilent VISA messages.

IO Monitor capture

To capture IO trace messages, select **Start Capturing Messages** from the IO Monitor toolbar. Now run the sample program again. Agilent VISA will write messages to IO Monitor as it communicates with the 34410. After the program stops, note that IO Monitor displays a very complete picture of the VISA I/O to and from the instrument (Figure 3).

Reading an IO Monitor trace

As shown in Figure 3, the top pane in the IO Monitor window lists trace messages in a timestamp sequence. Each message represents a call to a VISA function. The Method call shows the VISA function that was called and the IO Data column shows the content of the first buffer parameter of the call. For the 34410 driver, SCPI command strings are normally shown as buffers for viWrite method calls and string-return values (which are often numbers represented as strings) from the instrument are normally shown as buffers for viRead method calls.

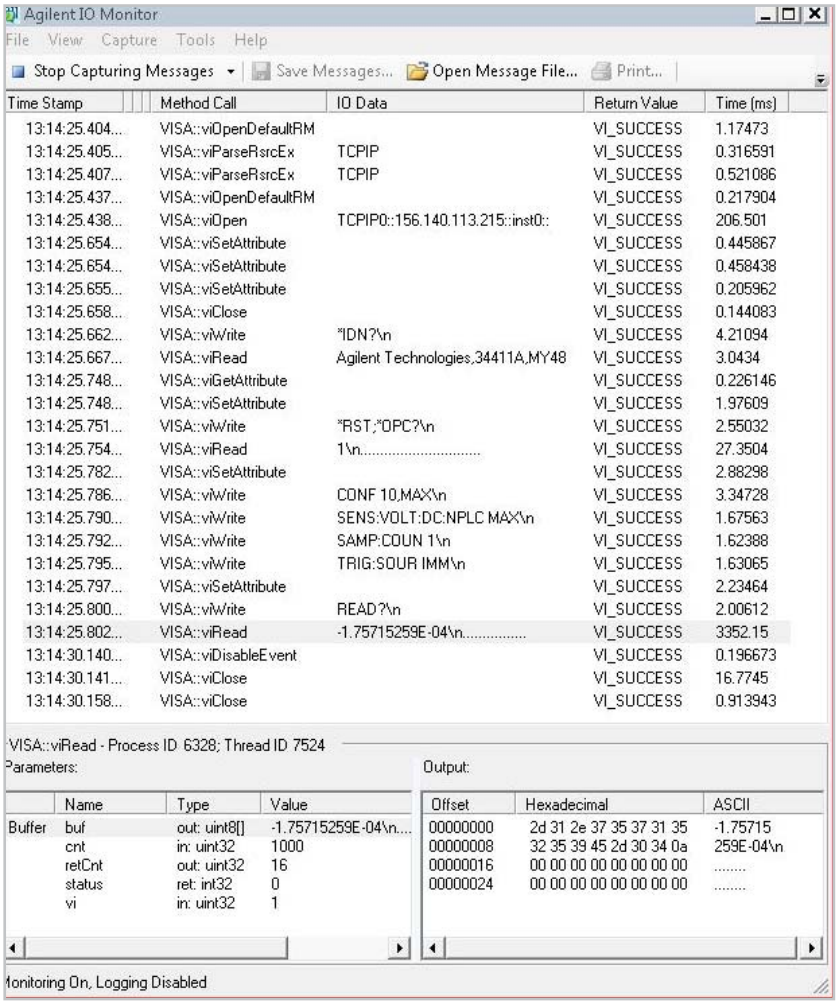

Figure 3. IO Monitor provides a detailed display of instrument communication.

IO monitor and general purpose I/O

IO Monitor is good for more than just the tracking of driver execution. In short, it's a general purpose I/O monitoring tool for any of the Agilent IO APIs (SICL, VISA and VISA-COM). Many developers prefer to program with SCPI directly, and test-system programs that use SICL, VISA or VISA-COM to send SCPI commands directly to an instrument can use IO Monitor to trace I/O calls.

To assist with troubleshooting, IO Monitor generates trace "message files." These are XML files that IO Monitor can write either dynamically as messages are being received or after an entire trace has been received. One caveat: Message files are not viewer friendly, but can be read into the **IO Monitor Log Viewer** application. The easiest way to compare two IO Monitor traces is to save them to message files and open each one in IO Monitor Log Viewer.

Finally, certain options in IO Monitor allow you to adapt it to a variety of I/O profiles. For example, memory use can be controlled by limiting the size of buffers (usually strings or arrays) returned by an instrument to a test program. On the other hand, if it is important to see the entire content of large strings or arrays, the buffer size can be increased. For programs with heavy I/O traffic, monitoring may affect execution speed. When performance is an issue, the IO Monitor display, which normally displays each trace message as it is acquired, can be configured to display the messages one window at a time to provide an incremental improvement in speed.

Trace message features

A quick review of the trace messages in Figure 3 reveals several interesting features:

- There is abundant I/O activity in the Initialize() call, which generated all of the trace messages from the beginning of the list to the message with the timestamp 13:14:25.782. This can be verified by debugging stepwise through the program (see next section).
- The SENS:VOLT:DC:NPLC MAX command is visible in the IO Monitor window (timestamp = 13:14:25.802).
- There is not an instrument SCPI command for setting the measurement timeout. In fact, the timeout is a VISA property. That is, VISA I/O will time out while waiting for a measurement response if the measurement takes too long. The VISA timeout is set by a VISA viSetAttribute() call (timestamp = 13:14:25.797).

Debugging with IO Monitor

Stepwise debugging can be used to correlate driver methods and properties from VISA I/O to the instrument. Start by setting a breakpoint on the first line of the client program. In the editor, click on the far left column next to the following line:

```
Agilent34410 driver = new Agilent34410Class();
```

Start IO Monitor and capture the message stream. Next, run the program with the debugger. Step through the first two lines – through the Initialize() call – and then look at the IO Monitor window. It should show only the lines from the Initialize() call. Step through the next line: IO Monitor should include an additional line showing that "CONF 10,MAX\n" was sent to the instrument. Keep stepping through the lines in the program, referring to IO Monitor between each line to see which lines have been added. Using this technique, it is relatively easy to see which commands are being sent to the instrument by each call the program makes to the driver.

Comparing Driver Tracing with I/O Monitor

Driver tracing and IO Monitor tracing are distinguished primarily by the fact than one is implemented in drivers and oriented towards driver operation, while the other is implemented in the Agilent IO Libraries Suite and is oriented toward I/O. Here is a summary of key differences, and some suggestions for choosing one or the other.

- **Name visibility:** Driver tracing shows the names of driver methods and properties; IO Monitor does not. Use driver tracing when it's important to see the driver calls that are being made.
- **Call visibility:** Driver tracing shows all calls to the driver, even those that don't perform I/O. IO Monitor shows all I/O activity from (generated by) the driver. Use IO Monitor if you need to see all of the I/O generated by a program, including calls made through the Formatted IO class (or any other means that bypasses the driver's implementation, as delivered by the vendor).
- **Viewing multiple drivers:** Driver trace files show activity from one instance of a driver at a time. If you want to see how two drivers interact, or even two instances of the same driver, you must create two driver-trace files with different names and manually compare them. IO Monitor shows all VISA I/O activity on a PC. This may include activity from multiple drivers and multiple test programs. While this makes it possible to get a bigger picture of how multiple instruments interact in a test program, it may also make it difficult to pick out what a specific instrument is doing.
- **Timestamp resolution**: Driver-tracing timestamps are stored in seconds; IO Monitor timestamps are in milliseconds. If you need more precise timestamps, use IO Monitor. If you need to manually correlate information from driver traces and IO Monitor, remember to account for this difference.
- **Performance:** The use of either driver tracing or IO Monitor may affect performance, especially for programs that do a lot of I/O very quickly. Driver tracing is typically faster than IO Monitor, but neither should be turned on for routine production use.
- **Dynamic Control:** Some recent IVI-COM drivers provide a System.Trace property that allows the test program to turn tracing on or off. This can be especially useful if you want to monitor only specific sections of your code.

Conclusion

In the process of getting test programs up and running in less time, driver tracing and IO Monitor each provide useful advantages. Fortunately, it isn't an either/or choice: You can use them together to maximize your understanding of how IVI-COM drivers communicate with the instruments in your system.

Being able to see what the driver actually does as it communicates with an instrument will go a long way towards building your confidence in IVI-COM drivers – and helping you maximize the benefits they offer. Using driver tracing and IO Monitor, you can quickly answer the questions posed at the beginning of this note: which SCPI commands are used by a driver, whether the driver checks the instrument for errors, and how the driver handles asynchronous communication.

Publication title	Pub number
Using .NET Methods to Add Functionality to IVI-COM Drivers	5990-3661EN
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

Microsoft is a U.S. registered trademark Microsoft Corporation.

Visual Studio is a registered trademark of Microsoft Corporation in the United States and/or other countries.

Remove all doubt

Our repair and calibration services will get your equipment back to you, performing like new, when promised. You will get full value out of your Agilent equipment through-out its lifetime. Your equipment will be serviced by Agilent-trained technicians using the latest factory calibration procedures, automated repair diagnostics and genuine parts. You will always have the utmost confidence in your measurements. For information regarding self maintenance of this product, please contact your Agilent office.

Agilent offers a wide range of additional expert test and measurement services for your equipment, including initial start-up assistance, onsite education and training, as well as design, system integration, and project management.

For more information on repair and calibration services, go to:

www.agilent.com/find/removealldoubt

www.agilent.com
www.agilent.com/find/open

For more information on Agilent Technologies' products, applications or services, please contact your local Agilent office. The complete list is available at:

www.agilent.com/find/contactus

Americas

Canada	(877) 894-4414
Latin America	305 269 7500
United States	(800) 829-4444

Asia Pacific

Australia	1 800 629 485
China	800 810 0189
Hong Kong	800 938 693
India	1 800 112 929
Japan	0120 (421) 345
Korea	080 769 0800
Malaysia	1 800 888 848
Singapore	1 800 375 8100
Taiwan	0800 047 866
Thailand	1 800 226 008

Europe & Middle East

Austria	01 36027 71571
Belgium	32 (0) 2 404 93 40
Denmark	45 70 13 15 15
Finland	358 (0) 10 855 2100
France	0825 010 700*
	*0.125 €/minute
Germany	07031 464 6333
Ireland	1890 924 204
Israel	972-3-9288-504/544
Italy	39 02 92 60 8484
Netherlands	31 (0) 20 547 2111
Spain	34 (91) 631 3300
Sweden	0200-88 22 55
Switzerland	0800 80 53 53
United Kingdom	44 (0) 118 9276201

Other European Countries:
www.agilent.com/find/contactus

Revised: March 24, 2009

Product specifications and descriptions in this document subject to change without notice.

© Agilent Technologies, Inc. 2009
Printed in USA, May 5, 2009
5990-4003EN

Agilent Technologies