



Getting Started with IVI Drivers

**Your Guide to Using IVI with
Visual C++**

Version 1.1

**© Copyright IVI Foundation, 2011
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.....	5
	Purpose	5
	Why Use an Instrument Driver?	5
	Why IVI?.....	6
	Why Use an IVI Driver?	8
	Flavors of IVI Drivers	9
	Shared Components	9
	Download and Install IVI Drivers	9
	Familiarizing Yourself with the Driver	10
	Examples	10
Chapter 2	Using IVI with Visual C++.....	12
	The Environment	12
	Example Requirements	12
	Download and Install the Driver	12
	Using IVI-COM in C++	12
	Create a New Project and Import the Driver Type Libraries.....	12
	Import COM Type Libraries	13
	Initialize COM.....	14
	Create an Instance of the Driver	14
	Initialize the Instrument	14
	Configure the Instrument	14
	Set the Trigger Delay	15
	Set the Reading Timeout/Display the Reading	15
	Error Checking	15
	Close the Session	15
	View the Results	15

Complete Source Code	16
Build and Run the Application	17
Using IVI-C in Visual C++.	18
Create a New Project and Import the Driver Type Libraries.	18
Include Driver Header	19
Declare Variables.	19
Define Error Checking	19
Initialize the Instrument	20
Configure the Instrument	20
Set the Trigger and Trigger Delay	20
Set the Reading Timeout/Display the Reading	21
Close the Session	21
View the Results	21
Complete Source Code	21
Build and Run the Application	21



Chapter 1

Introduction

• • •

Purpose

Welcome to ***Getting Started with IVI Drivers: Your Guide to Using IVI with Visual C++***. 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, 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, 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 an Agilent DMM. For example, the instrument driver interface for one may be `ke2000_read` while another may be `hp34401_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:

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 signal generator	IviRFSigGen
Upconverter	IviUpconverter
Downconverter	IviDownconverter
Digitizer	IviDigitizer
Counter/timer	IviCounter

IVI Class Compliant drivers usually also include capability that is not part of the IVI Class. It is common for instruments that are part of a class to have 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 conform to and are documented according to the IVI specifications and usually display the standard IVI 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 drivers 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 an 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). 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.

All IVI drivers communicate to the instrument through an I/O Library. Our examples use the Virtual Instrument Software Architecture (VISA), a widely used standard library for communicating with instruments from a personal computer.

Shared Components

To make it easier for you 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 your hardware or software vendor's web site or by linking to them from the IVI Foundation web site.

To see the list of drivers registered with the IVI Foundation, go to <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 Agilent Connection Expert.
- Reference the driver or load driver files – For the examples in the IVI guides, the driver is the **IVI-COM/IVI-C Version 1.2.2.0 for 34401A, October 2008 (from Agilent Technologies)** ... or the **Agilent 34401A IVI-C driver, Version 4.4, July 2010 (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:
 - 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. We 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 – We set a range of **1.5 volts** and a resolution of **0.001 volts (1 millivolt)**.
- Access an instrument property – We set the trigger delay to **0.01 seconds**.
- Set the reading timeout – We 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++

•••

The Environment

Microsoft Visual C++ is a software development environment for the C++ programming language and is available as part of Microsoft Visual Studio. Visual C++ allows you to create, debug, and execute conventional applications as well as applications that target the .NET Framework.

Example Requirements

- Visual C++
- Microsoft Visual Studio 2010
- IVI-COM: Agilent 34401A IVI-COM, Version 1.2.2.0, October 2008 (from Agilent Technologies); or
- IVI-C: Agilent 34401A IVI-C, Version 4.4, July 2010 (from National Instruments)
- Agilent IO Libraries Suite 16.1
- National Instruments IVI Compliance Package version 4.0 or later

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.

Since Visual C++ supports both IVI-COM and IVI-C drivers, this example is written two ways, first to show how to use an IVI-COM driver in Visual C++, and second to show how to use an IVI-C driver in Visual C++.

Note: *If you do not install the appropriate instrument driver, the project will not build because the referenced files are not included in the program. If you need to download and install a driver, you do not need to exit Visual Studio. Install the driver and continue with your program.*

Using IVI-COM in C++

The following sections show how to get started with an IVI-COM driver in Visual C++.

Create a New Project and Import the Driver Type Libraries

To use an IVI Driver in a Visual C++ program, you must provide the path to the type libraries it uses.

- 1 Launch Visual Studio 2010 and create a Visual C++ Win32 Console Application with the name "IviDemo". Use the default settings.

Note: *The program already includes some required code, including the standard header file:*

```
#include "stdafx.h"
```

- 2 In Solution Explorer, right click on the "IviDemo" project node and click on "Properties". This will open the "IviDemo Property Pages" dialog.
- 3 In the tree view on the left of the dialog, expand "Configuration Properties", then click on "VC++ Directories".
- 4 Locate the "Include Directories" row in the right hand pane and click on the drop down icon in the column that contains the directory paths. Click on "<Edit...>".
- 5 Add the following two entries to your path.

The first entry will point to the default directory for IVI drivers. On 32-bit Windows, use:

```
"C:\Program Files\IVI Foundation\IVI\Bin"
```

On 64-bit Windows, use:

```
"C:\Program Files (x86)\IVI Foundation\IVI\Bin"
```

The second entry points to the VISA DLL that many drivers require:

```
"$(VXIPNPPATH)VisaCom"
```

Note: *The second entry will point to the correct VISA COM directory regardless of whether you are operating with 32-bit or 64-bit Windows.*

- 6 Click OK twice to save changes and exit the "IviDemo Property Pages" dialog.

Import COM Type Libraries

COM type libraries must be imported before they can be accessed. To import the type libraries, type the following statements following the header file reference:

```
#import <IviDriverTypeLib.dll> no_namespace  
#import <IviDmmTypeLib.dll> no_namespace  
#import <GlobMgr.dll> no_namespace
```

```
#import <Ag34401.dll> no_namespace
```

Note: The `#import` statements access the driver type libraries used by the Agilent 34401 DMM. The `no_namespace` attribute allows the code to access the interfaces in the type libraries from the global namespace.

At this point the Visual C++ editor may flag the `#import` statements as errors. To fix the errors, select “Rebuild Solution” from the Build menu.

Initialize COM

- 1 Initialize the COM library, and check for errors. Add the following lines at the beginning of the `_tmain` function (immediately before the `return` statement):

```
HRESULT hr = ::CoInitialize(NULL);  
if (FAILED(hr)) exit(1);
```

- 2 To close the COM library before exiting, type the following line at the end of your code, right before the return line:

```
::CoUninitialize();
```

Create an Instance of the Driver

To create an instance of the driver, type

```
{  
    IIVI DmmPtr dmm(__uuidof(Agilent34401));  
}
```

Note: This creates a smart pointer that provides easy access to the COM object.

You are now ready to write the program to control the simulated instrument.

Initialize the Instrument

You can now write the main constructs for your program.

Below the smart pointer statement, type

```
dmm->Initialize("GPIB::23", false, true, "simulate=true");
```

Note: As soon as you type `->`, Intellisense displays options and helps ensure you use correct syntax and values.

Configure the Instrument

To set the range to 1.5 volts and resolution to 0.001 volts, type

```
dmm->Configure(IIVI DmmFunctionDCVolts, 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, make a reading with a timeout of 1 second (1000 milliseconds), and display the result to the console:

```
double reading = dmm->Measurement->Read(1000);  
wprintf(L"Reading: %g\n", reading);
```

Error Checking

To catch errors in the code, activate error checking.

- 1 Surround the preceding statements with a try block. Add the following lines before the call to the Initialize method:

```
try  
{
```

- 2 Process errors in a catch block. Add the following lines after the call to the wprintf method that follows the Read method:

```
}  
catch (_com_error e)  
{  
    wprintf(L"Error: %s", e.ErrorMessage());  
}
```

Close the Session

Close out the instance of the driver and free resources. Add the following line after the closing bracket of the catch block:

```
dmm->Close();
```

View the Results

Prompt the user to press any key to continue. Without these lines, the console window would immediately close before the user could view the information that was written to it. Add the following lines immediately before the return statement:

```
printf("\nDone - Press any key to exit");  
getchar();
```

Complete Source Code

The complete source code for the IviDemo.cpp file is shown below:

```
// IviDemo.cpp : Defines the entry point for the console
// application.
//
#include "stdafx.h"

#import <IviDriverTypeLib.dll> no_namespace
#import <IviDmmTypeLib.dll> no_namespace
#import <GlobMgr.dll> no_namespace
#import <Ag34401.dll> no_namespace

int _tmain(int argc, _TCHAR* argv[])
{
    HRESULT hr = ::CoInitialize(NULL);
    if (FAILED(hr)) exit(1);

    {
        IIVIvDmmPtr dmm(__uuidof(Agilent34401));

        try
        {
            dmm->Initialize("GPIB::23", false, true,
"simulate=true");
            dmm->Configure(IviDmmFunctionDCVolts, 1.5,
0.001);
            dmm->Trigger->Delay = 0.01;
            double reading = dmm->Measurement->Read(1000);
            wprintf(L"Reading: %g\n", reading);
        }
        catch (_com_error e)
        {
            wprintf(L"Error: %s", e.ErrorMessage());
        }
        dmm->Close();
    }

    ::CoUninitialize();

    printf("\nDone - Press any key to exit");
}
```

```

        getchar();

        return 0;
    }

{
    HRESULT hr;
    hr = CoInitialize(NULL);
    if(FAILED(hr))
        exit(1);
    {
        IAgilent34401Ptr dmm(__uuidof(Agilent34401));
        try{
dmm->Initialize("GPIB::23", false, true, "simulate=true");
            dmm->DCVoltage->Configure(1.5, 0.001);
            dmm->Trigger->Delay = 0.01;
            double reading = dmm->Measurement->Read(1000);
            wprintf(L"Reading: %g\n", reading);
        }
        catch(_com_error e){
            wprintf(L"Error: %s\n", e.ErrorMessage);
        }
        dmm->Close();
    }
    CoUninitialize();
    return 0;
}

```

Build and Run the Application

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

- 1 From the Build menu, select “Build”, and click “Rebuild Solution”.
- 2 From the Debug menu, select “StartDebugging” to run the application.

Using IVI-C in Visual C++

The following sections show to get started with IVI-C in Visual C++.

Create a New Project and Import the Driver Type Libraries

To use an IVI-C Driver in a Visual C++ program, you must provide paths to the header files and libraries it uses.

- 1 Launch Visual Studio 2010 and create a Visual C++ Win32 Console Application with the name “IviDemo2”. Use the default settings.

Note: *The program already includes some required code, including the standard header file:*

```
#include "stdafx.h"
```

- 2 In Solution Explorer, right click on the “IviDemo2” project node and click on “Properties”. This will open the “IviDemo2 Property Pages” dialog.
- 3 In the tree view on the left of the dialog, expand “Configuration Properties”, then click on “VC++ Directories”.
- 4 Locate the “Include Directories” row in the right hand pane and click on the drop down icon in the column that contains the directory paths. Click on “<Edit...>”.
- 5 Add the following two entries to your path. The first entry will point to the default directory for IVI drivers.

On 32-bit Windows, use:

```
"C:\Program Files\IVI Foundation\IVI\Include"
```

On 64-bit Windows, use:

```
"C:\Program Files (x86)\IVI Foundation\IVI\Include"
```

The second entry points to the VISA DLL that many drivers require:

```
"$(VXIPNPPATH)WinNT\include"
```

Note: *The second entry will point to the correct VISA directory regardless of whether you are operating with 32-bit or 64-bit Windows.*

- 6 Locate the “Library Directories” row in the right hand pane and click on the drop down icon in the column that contains the directory paths. Click on “<Edit...>”.
- 7 Add the following two entries to your path. The first entry will point to the default directory for IVI drivers.

On 32-bit Windows, use:

```
"C:\Program Files\IVI Foundation\IVI\Lib\msc"
```

On 64-bit Windows, use:

```
"C:\Program Files (x86)\IVI Foundation\IVI\Lib\msc"
```

The second entry points to the VISA DLL that many drivers require:

```
"$(VXIPNPPATH)WinNT\lib\msc"
```

Note: The second entry will point to the correct VISA directory regardless of whether you are operating with 32-bit or 64-bit Windows.

- 8 Next, expand "Linker" in the tree view on the left of the "IviDemo2 Property Pages" dialog, then click on "Input".
- 9 Locate the "Additional Dependencies" row in the right hand pane and click on the drop down icon in the column that contains the list of .lib files. Click on "<Edit...>".
- 10 Add the following library file to the list:

```
"hp34401a.lib"
```
- 11 Click OK twice to save changes and exit the "IviDemo2 Property Pages" dialog.

Include Driver Header

To add the hp34401a instrument driver header file to your program, type the following statement following the existing header file reference:

```
#include "hp34401a.h"
```

Select "Rebuild Solution" from the Build menu.

Declare Variables

Declare the program variables. Add the following lines at the beginning of the `_main` function (immediately before the `return` statement):

```
ViSession session;  
ViStatus error = VI_SUCCESS;  
ViReal64 reading;
```

Define Error Checking

Next define error checking for your program. First you will define a macro to catch the errors. It is better to define it once at the beginning of the program that to add the logic to each of your program statements. After the `#include` statements, type the following lines:

```
#ifndef checkErr  
#define checkErr(fCall) \  
if (error = (fCall), (error = (error < 0) ? error :  
VI_SUCCESS)) \  
{goto Error;} else error = error  
#endif
```

Next add code to handle any errors that occur. Add the following lines before the return statement:

```
Error:
    if (error != VI_SUCCESS)
    {
        ViChar errStr[2048];
        hp34401a_GetError (session, &error, 2048,
errStr);
        printf ("Error!", errStr);
    }
```

Note: Including error handling in your programs is good practice. This code checks for errors in your program.

Initialize the Instrument

To initialize the instrument, add the following Initialize with Options function right after the variable declarations you added in the previous section:

```
checkErr( hp34401a_InitWithOptions
("GPIB::23::INSTR",VI_FALSE, VI_TRUE,
"Simulate = 1",
&session));
```

This initializes the instrument with the following parameters:

- GPIB0::23::INSTR is the Resource Name (instrument at GPIB address 23).
- VI_FALSE indicates that an ID Query should not be performed by this function.
- VI_TRUE resets the device .
- Simulate=1 is the Options parameter that sets the driver to simulation mode.
- &session assigns the Instrument Handle to the variable “session” defined above. Configure the Instrument

Configure the Instrument

To set the range to 1.5 volts and resolution to 0.001 millivolts, type:

```
checkErr( hp34401a_ConfigureMeasurement (session,
HP34401A_VAL_DC_VOLTS, 1.5, 0.001));
```

Set the Trigger and Trigger Delay

To set the trigger source to immediate and the trigger delay to 0.01 seconds, type:

```
checkErr( hp34401a_ConfigureTrigger (session,
HP34401A_VAL_IMMEDIATE,
```

```
0.01));
```

Set the Reading Timeout/Display the Reading

To take a reading from the instrument and to set the reading timeout to 1 second (1000 ms) type, and display the result using the printf function:

```
checkErr( hp34401a_Read (session, 1000, &reading);  
printf ("Reading = %f", reading);
```

Note: The Read function takes a reading from the instrument and assigns the result to the variable “reading” defined above.

Close the Session

To close out the instance of the driver and free resources, add the following lines immediately before the return statement:

```
If (session)  
hp34401a_Close(session);
```

View the Results

Prompt the user to press any key to continue. Without these lines, the console window would immediately close before the user could view the information that was written to it. Add the following lines immediately before the return statement:

```
printf("\nDone - Press any key to exit");  
getchar();
```

Complete Source Code

The complete source code for the IviDemo2.cpp file is shown below:

```
// IviDemo2.cpp : Defines the entry point for the console  
// application.  
//  
#include "stdafx.h"  
#include <hp34401a.h>  
  
#ifndef checkErr  
#define checkErr(fCall) \  
if (error = (fCall), (error = (error < 0) ? error :  
VI_SUCCESS)) \  

```

```

{goto Error;} else error = error
#endif

int _tmain(int argc, _TCHAR* argv[])
{
    ViSession session;
    ViStatus error = VI_SUCCESS;
    ViReal64 reading;
    checkErr( hp34401a_InitWithOptions ("GPIB::23::INSTR",
    VI_FALSE, VI_TRUE,
    "Simulate=1", &session));
    checkErr( hp34401a_ConfigureMeasurement (session,
    HP34401A_VAL_DC_VOLTS,
    1.5, 0.0001));
    checkErr( hp34401a_ConfigureTrigger (session,
    HP34401A_VAL_IMMEDIATE, 0.01));
    checkErr( hp34401a_Read (session, 1000, &reading));
    printf ("Reading = %f", reading);

    Error:
    if (error != VI_SUCCESS)
    {
        ViChar errStr[2048];
        hp34401a_GetError (session, &error, 2048, errStr);
        printf ("Error!", errStr);
    }

    if (session)
        hp34401a_close (session);

    printf("\nDone - Press any key to exit");
    getchar();

    return 0;
}

```

Build and Run the Application

- Build your application and run it to verify it works properly.
- 1 From the Build menu, select “Build”, and click “Rebuild Solution”.
 - 2 From the Debug menu, select “StartDebugging” to run the application.

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