Keysight M9420A VXT Vector Transceiver

KEYSIGHT TECHNOLOGIES

Programmer's Guide

Notices

Manual Part Number

M9420-90031

Published By

Keysight Technologies
No 116 Tianfu 4th Street
Hi-Tech Industrial Zone (South)
Chengdu, China

Edition

Edition 1, September 2015

Regulatory Compliance

This product has been designed and tested in accordance with accepted industry standards, and has been supplied in a safe condition. To review the Declaration of Conformity, go to http://www.keysight.com/go/conformity.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

The following safety precautions should be observed before using this product and any associated instrumentation.

This product is intended for use by qualified personnel who recognize shock hazards and are familiar with the

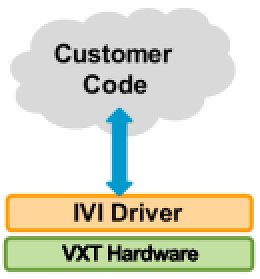
Contents

What You Will Learn in This Programmer's Guide

This programmer's guide is intended for individuals who write and run programs to control test-and-measurement instruments. Specifically, in this programmer's guide, you will learn how to use Visual Studio 2010 with the .NET Framework to write IVI-COM Console Applications in Visual C#. Knowledge of Visual Studio 2010 with the .NET Framework and knowledge of the programming syntax for Visual C# is required.

Our basic user programming model uses the IVI-COM driver directly and allows customer code to:

- Access the IVI-COM driver at the lowest level
- Access IQ Acquisition Mode, Power Acquisition Mode, and Spectrum Acquisition Mode
- Control the Keysight M9420A VXT Vector Transceiver while performing PA/FEM Power Measurement Production Tests
- Generate waveforms created by Signal Studio software (licenses are required)



- Example Program 1: How to Print Driver Properties, Check for Errors, and Close Driver Sessions
- Example Program 2: How to Perform a Channel Power Measurement Using Immediate Trigger
- Example Program 3: How to Perform a WCDMA Power Servo and ACPR Measurement

Related Websites

- Keysight Technologies PXI and AXIe Modular Products
  - M9420A VXT Vector Transceiver
- Keysight Technologies
  - IVI Drivers & Components Downloads
  - Keysight I/O Libraries Suite

  - GPIB, USB, & Instrument Control Products
  - Keysight VEE Pro
  - Technical Support, Manuals, & Downloads
  - Contact Keysight Test & Measurement
- IVI Foundation - Usage Guides, Specifications, Shared Components Downloads
- MSDN Online

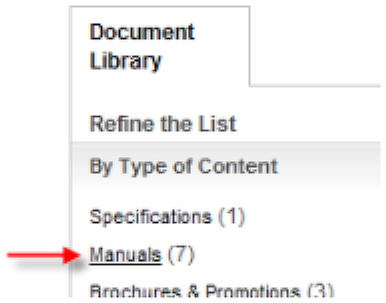
Related Documentation

To access documentation related to the Keysight M9420A VXT Vector Transceiver Programmer's Guide , use one of the following methods:

- If the product software is installed on your PC, the related documents are also available in the software installation directory.

Document	**Description**	**Format**
Getting Started Guide	Includes procedures to help you to unpack, inspect, install (software and hardware), perform instrument connections, verify operability, and troubleshoot your product.	*PDF*
IVI Driver reference (help system)	Provides detailed documentation of the IVI-COM and IVI-C driver API functions, as well as information to help you get started with using the IVI drivers in your application development environment.	*CHM (Microsoft Help Format)*
X-series Applications Programmer's Guide	Provides detailed documentation of the IVI-COM and IVI-C driver API functions, as well as information to help you get started with using the IVI drivers in your application development environment.	*PDF*
User's and Programming Reference	Describes the SCPI commands supported by the M9420A VXT Vector Transceiver.	*CHM (Microsoft Help Format)*

- The documentation listed above is also available on the product CD.
- To find the very latest versions of the user documentation, go to the product web site (www.keysight.com/find/M9420A) and download the files from the Manuals support page (go to **Document Library > Manuals**):



Overall Process Flow

Perform the following steps:

1. Write source code using Microsoft Visual Studio 2010 with .NET Visual C# running on Windows 7.
2. Compile source code using the .NET Framework Library.
3. Produce an Assembly.exe file – this file can run directly from Microsoft Windows without the need for any other programs.
   - When using the Visual Studio Integrated Development Environment (IDE), the Console Applications you write are stored in conceptual containers called **Solutions** and **Projects**.
   - You can view and access Solutions and Projects using the **Solution Explorer** window (View > Solution Explorer).

Installing Hardware, Software, and Licenses

Perform the following steps:

1. Unpack and inspect all hardware.
2. Verify the shipment contents.
3. Install the software. Note the following order when installing software.

   a. Install Microsoft Visual Studio 2010 with .NET Visual C# running on Windows 7.

      You can also use a free version of Visual Studio Express 2010 tools from: http://www.microsoft.com/visualstudio/eng/products/visual-studio-2010-express

      The following steps must be completed before programmatically controlling the M9420A hardware with the IVI driver.

   b. Install Keysight IO Libraries Suite (IOLS); this installation includes Keysight Connection Expert.

   c. Install the M9420A VXT software, Version 16.50 or newer.

      Driver software includes all IVI-COM and IVI-C Drivers and documentation. All of these items may be downloaded from the Keysight product websites:

      – http://www.keysight.com/find/iosuite > Select **Technical Support** > Select the **Drivers, Firmware & Software** tab > Download the **Keysight IO Libraries Suite Recommended**
      – http://www.keysight.com/find/m9420a > Select **Technical Support** > Select the **Drivers, Firmware & Software** tab > Download the **Instrument Driver**.
      – http://www.keysight.com/find/ivi - download other installers for Keysight IVI-COM drivers

4. Install the VXT modules and make cable connections. For detailed procedures, please refer to **M9420A Getting Started Guide**.

   The **M9300A PXIe Reference** must be included as part of the M9420A configurations. The M9300A PXIe Reference must be initialized first so that the other configurations that depend on the reference signal get the signal they are expecting. If the configuration of modules that is initialized first does not include the M9300A PXIe Reference, unlock errors will occur.

   Once the software and hardware are installed and Self-Test has been performed, they are ready to be programmatically controlled.

APIs for the M9420A VXT Vector Transceiver

The following IVI driver terminology may be used when describing the Application Programming Interfaces (APIs) for the M9420A VXT Vector Transceiver.

IVI[Interchangeable Virtual Instruments] - a standard instrument driver model defined by the IVI Foundation that enables engineers to exchange instruments made by different manufacturers without rewriting their code. www.ivifoundation.org

IVI Instrument Classes (Defined by the IVI Foundation)

Currently, there are 13 IVI Instrument Classes defined by the IVI Foundation. The M9420A VXT Vector Transceiver do not belong to any of these 13 IVI Instrument Classes and are therefore described as "NoClass" modules.

- DC Power Supply
- AC Power Supply
- DMM
- Function Generator
- Oscilloscope
- Power Meter
- RF Signal Generator
- Spectrum Analyzer
- Switch
- Upconverter
- Downconverter
- Digitizer
- Counter/Timer

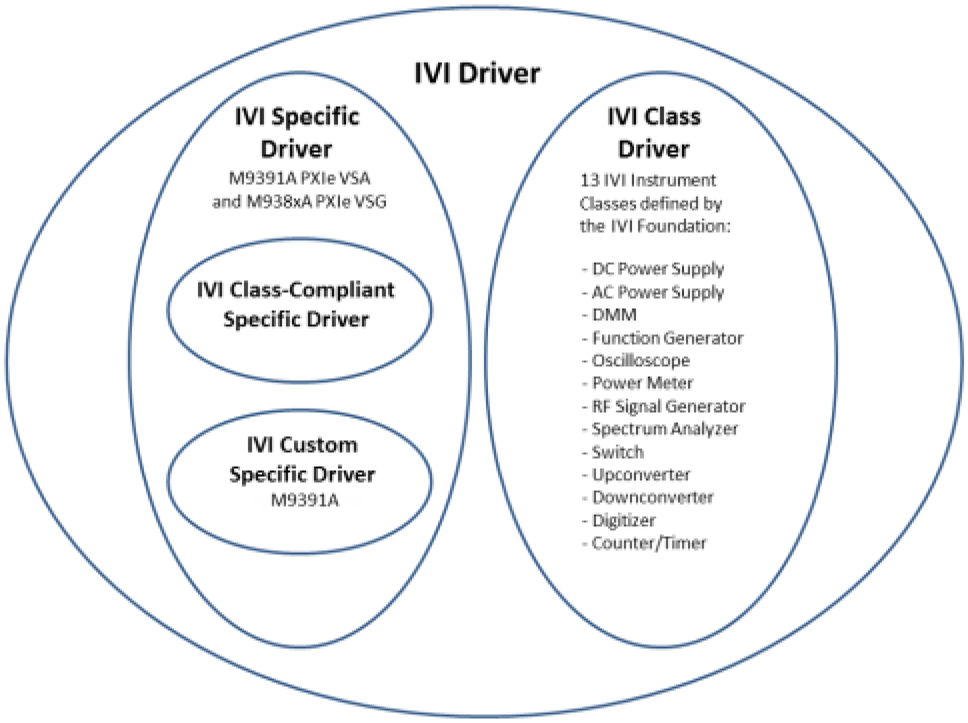
IVI Compliant or IVI Class Compliant

The M9420A VXT Vector Transceiver is IVI Compliant, but not IVI Class Compliant; none of these belongs to one of the 13 IVI Instrument Classes defined by the IVI Foundation.

- **IVI Compliant**– means that the IVI driver follows architectural specifications for these categories:
  - Installation
  - Inherent Capabilities
  - Cross Class Capabilities
  - Style
  - Custom Instrument API

- IVI Class Compliant– means that the IVI driver implements one of the 13 IVI Instrument Classes
  - If an instrument is IVI Class Compliant, it is also IVI Compliant
  - Provides one of the 13 IVI Instrument Class APIs in addition to a Custom API
  - Custom API may be omitted (unusual)
  - Simplifies exchanging instruments

IVI Driver Types



- IVI Driver
  - Implements the *Inherent Capabilities Specification*
  - Complies with all of the architecture specifications
  - May or may not comply with one of the 13 IVI Instrument Classes
  - Is either an IVI Specific Driver or an IVI Class Driver
- IVI Class Driver
  - Is an IVI Driver needed only for interchangeability in IVI-C environments
  - The IVI Class may be IVI-defined or customer-defined
- IVI Specific Driver
  - Is an IVI Driver that is written for a particular instrument such as the M9420A VXT Vector Transceiver

- IVI Class-Compliant Specific Driver
    - IVI Specific Driver that complies with one (or more) of the IVI defined class specifications
    - Used when hardware independence is desired
- IVI Custom Specific Driver
    - Is an IVI Specific Driver that is not compliant with any one of the 13 IVI defined class specifications
    - Not interchangeable

NOTE **This release is not binary compatible with prior releases of the IVI-C driver. Programs using the C/C++ IVI-C driver must be recompiled for this version of the driver. Similarly, programs compiled with this version of the driver will not be compatible with older versions of the IVI-C driver. This incompatibility is due to renumbering of attribute constants defined in the KtM9420.h include file.**

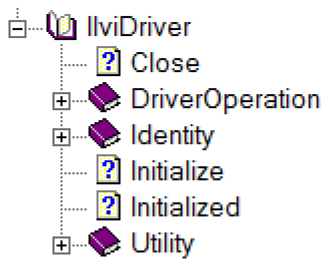
IVI Driver Hierarchy

When writing programs, you will be using the interfaces (APIs) available to the IVI-COM driver.

- The core of every IVI-COM driver is a single object with many interfaces.
- These interfaces are organized into two hierarchies: Class-Compliant Hierarchy and Instrument-Specific Hierarchy – and both include the IIviDriver interfaces.
    - Class-Compliant Hierarchy - Since the M9420A VXT Vector Transceiver does not belong to one of the 13 IVI Classes, there is noClass-Compliant Hierarchy in their IVI Driver.
    - Instrument-Specific Hierarchy
        - The M9420A VXT Vector Transceiver's instrument-specific hierarchy has IKtM9420 at the root (where KtM9420 is the driver name).
            - IKtM9420 is the root interface and contains references to child interfaces, which in turn contain references to other child interfaces. Collectively, these interfaces define the Instrument-Specific Hierarchy.
    - The IIviDriver interfaces are incorporated into both hierarchies: Class-Compliant Hierarchy and Instrument-Specific Hierarchy.

      The IIviDriver is the root interface for IVI Inherent Capabilities which are what the IVI Foundation has established as a set of functions and attributes that all IVI drivers must include – irrespective of which IVI instrument class the driver supports. These common functions and attributes are called IVI inherent capabilities and they are documented in

IVI-3.2 – Inherent Capabilities Specification. Drivers that do not support any IVI instrument class such as the M9420A VXT Vector Transceiver must still include these IVI inherent capabilities.



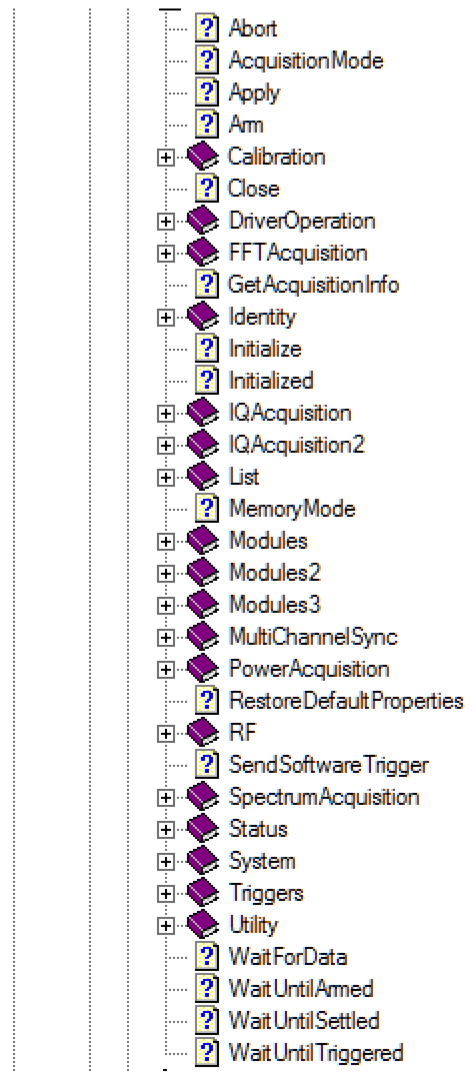
| | IiviDriver
Close
DriverOperation
Identity
Initialize
Initialized
Utility |

Instrument-Specific Hierarchies for M9420A

The following table lists the instrument-specific hierarchy interfaces for M9420A VXT Vector Transceiver.

Keysight M9420A VXT Instrument-Specific Hierarchy
KtM9420 is the driver name
IKtM9420Ex is the root interface

Keysight M9420A VXT Instrument-Specific Hierarchy

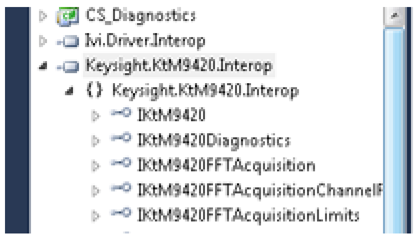

Abort
AcquisitionMode
Apply
Arm
Calibration
Close
DriverOperation
FFTAcquisition
GetAcquisitionInfo
Identity
Initialize
Initialized
IQAcquisition
IQAcquisition2
List
MemoryMode
Modules
Modules2
Modules3
MultiChannelSync
PowerAcquisition
RestoreDefaultProperties
RF
SendSoftwareTrigger
SpectrumAcquisition
Status
System
Triggers
Utility
WaitForData
WaitUntilArmed
WaitUntilSettled
WaitUntilTriggered

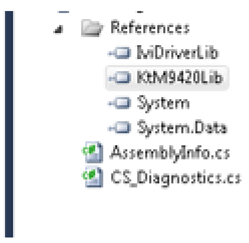
NOTE

All new code being created should use the IKtM9420Ex extended interfaces in place of the IKtM9420 interfaces. New functionalities have been added to the IKtM9420Ex extended interfaces. These new functionalities were not available in the original IKtM9420 interfaces, and have been left unchanged to support previously written code; this helps support backward code compatibility.

When Using Visual Studio

– To view interfaces available in the M9420 PXIe VXT, right-click KtM9420Lib library file, in the References folder, from the Solution Explorer window and select View in Object Browser.





Naming Conventions Used to Program IVI Drivers

General IVI Naming Conventions

- All instrument class names start with "Ivi"
  - Example: IviScope, IviDmm
- Function names
  - One or more words use PascalCasing
  - First word should be a verb

IVI-COM Naming Conventions

- Interface naming
  - Class compliant: Starts with "IIvi"
  - I<ClassName>
  - Example: IIviScope, IIviDmm
- Sub-interfaces add words to the base name that match the C hierarchy as close as possible
  - Examples: IIviFgenArbitrary, IIviFgenArbitraryWaveform
- Defined values
  - Enumerations and enum values are used to represent discrete values in IVI-COM
  - <ClassName><descriptive words>Enum
  - Example: IviScopeTriggerCouplingEnum
- Enum values don't end in "Enum" but use the last word to differentiate
  - Examples: IviScopeTriggerCouplingAC and IviScopeTriggerCouplingDC

Creating a Project with IVI-COM Using C-Sharp

This tutorial will walk through the various steps required to create a console application using Visual Studio and C#. It demonstrates how to instantiate two driver instances, set the resource names and various initialization values, initialize the two driver instances, print various driver properties to a console for each driver instance, check drivers for errors and report the errors if any occur, and close both drivers.

Step 1. – Create a "Console Application"
Step 2. – Add References
Step 3. – Add using Statements
Step 4. – Create an Instance
Step 5. – Initialize the Instance
Step 6. – Write the Program Steps (Create a Signal or Perform a Measurement)
Step 7. – Close the Instance

At the end of this tutorial is a complete example program that shows what the console application looks like if you follow all of these steps.

Step 1 - Create a Console Application

NOTE **Projects that use a Console Application do not show a Graphical User Interface (GUI) display.**

1. Launch Visual Studio and create a new Console Application in Visual C# by selecting: **File > New > Project** and select a Visual C# **Console Application**.
2. Enter "VxtProperties" as the **Name** of the project and click **OK**.

   NOTE **When you select New, Visual Studio will create an emptyProgram.csfile that includes some necessary code, including using statements. This code is required, so do not delete it.**

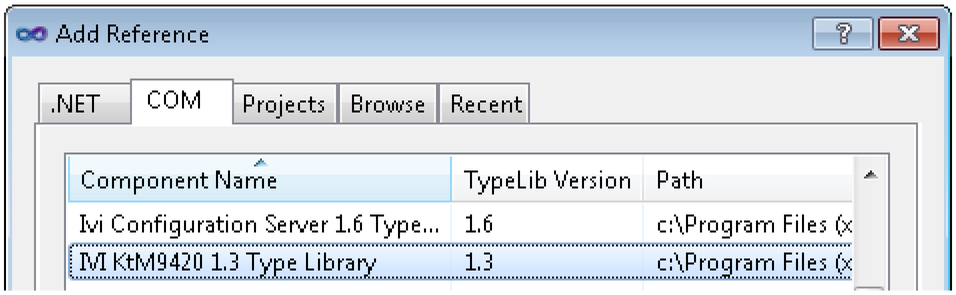
3. Select **Project** and click **Add Reference**. The Add Reference dialog appears. For this step, Solution Explorer must be visible (**View > Solution Explorer**) and the "Program.cs" editor window must be visible; select the **Program.cs** tab to bring it to the front view.

Step 2 - Add References

In order to access the M9420A VXT driver interfaces, references to their drivers (DLL) must be created.

1. In **Solution Explorer**, right-click on **References** and select **Add Reference**.
2. From the **Add Reference** dialog, select the **COM** tab.

Click on any of the type libraries under the "Component Name" heading and enter the letter "I".(All IVI drivers begin with IVI so this will move down the list of type libraries that begin with "I".)



Add Reference

Component Name	TypeLib Version	Path
Ivi Configuration Server 1.6 Type...	1.6	c:\Program Files (x
IVI KtM9420 1.3 Type Library	1.3	c:\Program Files (x

3.

NOTE If you have not installed the IVI driver for the M9420A VXT product (as listed in the previous section titled "Before Programming, Install Hardware, Software, and Software Licenses"), the IVI drivers will not appear in this list.

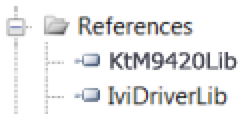
Also, the TypeLib Version that appears will depend on the version of the IVI driver that is installed. The version numbers change over time and typically increase as new drivers are released.
If the TypeLib Version that is displayed on your system is higher than the ones shown in this example, your system simply has newer versions – newer versions may have additional commands available.
To get the IVI drivers to appear in this list, you must close this Add Reference dialog, install the IVI drivers, and come back to this section and repeat "Step 2 – Add References".

Scroll to IVI section and, using **Shift-Ctrl**, select the following type libraries then select **OK**.
IVI KtM9420x 1.2 Type Library

4.

NOTE When any of the references for the KtM9420A are added, the IVIDriver 1.0 Type Library is also automatically added. This is visible as IviDriverLib under the project Reference; this reference houses the interface definitions for IVI inherent capabilities which are located in the file IviDriverTypeLib.dll (dynamically linked library).

These selected type libraries appear under the **References** node, in Solution Explorer, as:



5.

NOTE The program looks same as before you added the References, with the difference that the IVI drivers that are referenced are now available for use.

To allow your program to access the IVI drivers without specifying full path names of each interface or enum, you need to add using statements to your program.

Step 3 - Add Using Statements

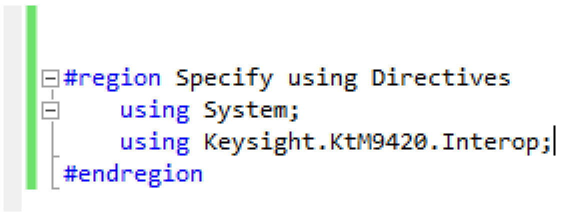
All data types (interfaces and enums) are contained within namespaces. (A namespace is a hierarchical naming scheme for grouping types into logical categories of related functionality. Design tools, such as Visual Studio, can use namespaces which makes it easier to browse and reference types in your code.)The C# using statement allows the type name to be used directly. Without the using statement, the complete namespace-qualified name must be used. To allow your program to access the IVI driver without having to type the full path of each interface or enum, type the following using statements immediately below the other using statements. The following example illustrates how to add using statements.

To Access the IVI Drivers Without Specifying or Typing The Full Path

These using statements should be added to your program:

```
using Ivi.Driver.Interop;
using Keysight.KtM9420.Interop;
```

NOTE You can create sections of code in your program that can be expanded and collapsed by surrounding the code with #region and #endregion keywords. Select – or + symbol to collapse or expand the region.



```
#region Specify using Directives
    using System;
    using Keysight.KtM9420.Interop;
#endregion
```

Step 4 - Create Instances of the IVI-COM Drivers

There are two ways to instantiate (create an instance of) the IVI-COM drivers:

– *Direct Instantiation*
– *COM Session Factory*

Since the M9420A VXT Vector Transceiver is considered NoClass module(because they do not belong to one of the 13 IVI Classes), the COM Session Factory is not used to create instances of their IVI-COM drivers. So, M9420A VXT Vector Transceiver IVI-COM driver uses direct instantiation. Because direct instantiation is used, their IVI-COM drivers may not be interchangeable with other modules.

To Create Driver Instances

The **new** operator is used in C# to create an instance of the driver.

```
IKtM9420 Driver = new KtM9420();
```

Step 5 - Initialize the Driver Instances

The `Initialize()` method is required when using any IVI driver. It establishes a communication link (an "I/O session") with an instrument and it must be called before the program can do anything with an instrument or work in simulation mode.

The `Initialize()` method has a number of options that can be defined. In this example, we prepare the `Initialize()` method by defining only a few of the parameters, then we call the `Initialize()` method with these parameters:

Resource Names

- If you are using Simulate Mode, you can set the Resource Name address string to:
  `string VxtResourceName = "%";`
- If you are actually establishing a communication link (an "I/O session") with an instrument, you need to determine the Resource Name address string (VISA address string) that is needed.You can use an IO application such as Agilent/Keysight Connection Expert, Agilent/Keysight Command Expert, National Instruments Measurement and Automation Explorer (MAX) to get the physical Resource Name string.

  In this guide, the example programs use the address below for M9420A VXT Vector Transceiver.

Module Name	Slot Number	VISA Address
M9420A VXT Vector Transceiver	2	PXI0::23-0.0::INSTR

Initialize() Parameters

NOTE **Although the `Initialize()` method has a number of options that can be defined (see *Initialize Options* below), we are showing this example with a minimum set of options to help minimize complexity.**

```
string VxtResourceName = "PXI0::23-0.0::INSTR;

bool IdQuery = true;
bool Reset = true;

string VxtOptionString = "QueryInstrStatus=true, Simulate=false, DriverSetup=
Model=VXT, Trace=false";

// Initialize the drivers
VxtDriver.Initialize(VxtResourceName, IdQuery, Reset, VxtOptionString);
Console.WriteLine("VXT Driver Initialized");
```

The above example shows how IntelliSense is invoked by simply rolling the cursor over the word "Initialize".

NOTE One of the key advantages of using C# in the Microsoft Visual Studio Integrated Development Environment (IDE) is IntelliSense. IntelliSense is a form of auto-completion for variable names and functions and a convenient way to access parameter lists and ensure correct syntax. This feature also enhances software development by reducing the amount of keyboard input required.

Initialize() Options

The following table describes options that are most commonly used with the `Initialize()` method.

Property Type and Example Value	**Description of Property**
string ResourceName = PXI[bus]::device [::function][::INSTR] string ResourceName = "PXI13::0::0::INSTR";	VxtResourceName – The driver is typically initialized using a physical resource name descriptor, often a VISA resource descriptor. See the procedure in the *Resource Names* section.
bool IdQuery = true;	Setting the **ID query** to false prevents the driver from verifying that the connected instrument is the one the driver was written for because if IdQuery is set to true, this will query the instrument model and fail initialization if the model is not supported by the driver.
bool Reset = true;	Setting **Reset** to true instructs the driver to initially reset the instrument.
string OptionString = "QueryInstrStatus=true,	OptionString - Setup the following initialization options:

Property Type and Example Value	Description of Property
Simulate=true,	– QueryInstrStatus=true (Specifies whether the IVI specific driver queries the instrument status at the end of each user operation.) – Simulate=true (Setting Simulate to true instructs the driver to not to attempt to connect to a physical instrument, but use a simulation of the instrument instead.) – Cache=false (Specifies whether or not to cache the value of properties.) – InterchangeCheck=false (Specifies whether the IVI specific driver performs interchangeability checking.) – RangeCheck=false (Specifies whether the IVI specific driver validates attribute values and function parameters.) – RecordCoercions=false (Specifies whether the IVI specific driver keeps a list of the value coercions it makes for ViInt32 and ViReal64 attributes.)
DriverSetup= ";	– DriverSetup= (This is used to specify settings that are supported by the driver, but not defined by IVI. If the Options String parameter (OptionString in this example) contains an assignment for the Driver Setup attribute, the Initialize function assumes that everything following 'DriverSetup=' is part of the assignment.)

If these drivers were installed, additional information can be found under *Initializing the IVI-COM Driver* from the following:

KtM9420x IVI Driver Reference
Start > All Programs > Keysight Instrument Drivers > IVI-COM-C Drivers > KtM9420 > KtM9420x IVI Driver Help

Step 6 - Write the Program

At this point, you can add program steps that use the driver instances to perform tasks.

In this example, perform the following steps:

Below is the corresponding code in C#:

```
// Set the output frequency to 1 GHz
Driver.Source.RF.Frequency = 1000000000;
// Set the output level to 0 dBm
```

```
Driver.Source.RF.Level = 0;
// Enables the RF Output
Driver.Source.RF.OutputPort = KtM9420PortEnum.KtM9420PortRFOutput;
// Apply all the setting above
Driver.Apply();
```

Step 7 - Close the Driver

Calling `Close()` at the end of the program is required by the IVI specification when using any IVI driver.

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.

```
{
  if(Driver!= null && Driver.Initialized)
   {
       // Close the driver
       Driver.Close();
       Console.WriteLine("");
          Console.WriteLine("driver Closed\n");
   }
}
```

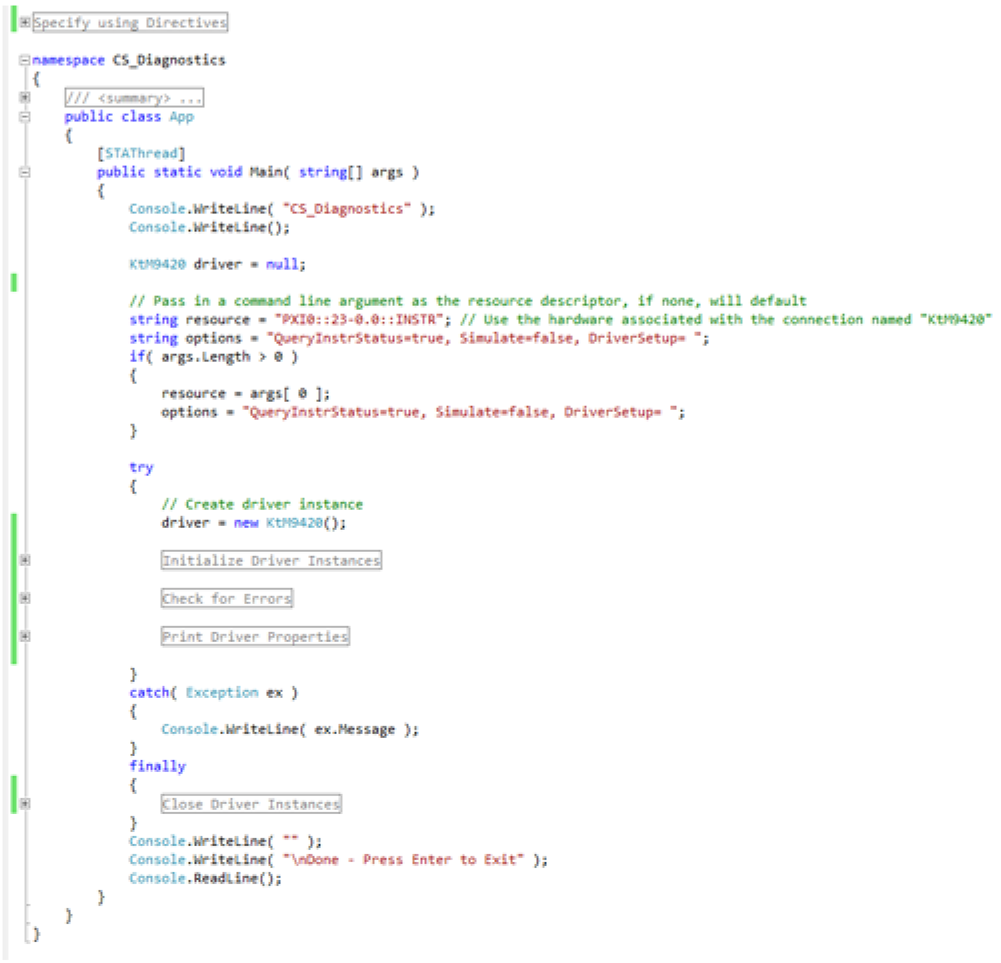
Step 8 - Building and Running a Complete Program Using Visual C-Sharp

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

1. Open the solution file **SolutionNameThatYouUsed.sln** in Visual Studio 2010.
2. Set the appropriate platform target for your project.
3. Choose Project > **ProjectNameThatYouUsed** Properties and select **Build | Rebuild Solution**.
   - **Tip**: You can also do the same thing from the Debug menu by clicking Start Debugging or pressing the **F5** key.

Example Program 1 – Code Structure

The following example code builds on the previously presented *Tutorial: Creating a Project with IVI-COM Using C#* and demonstrates how to instantiate two driver instances, set the resource names and various initialization values, initialize the two driver instances, print various driver properties for each driver instance, check drivers for errors and report the errors if any occur, and close the drivers.



```
Specify using Directives

namespace CS_Diagnostics
{
    /// <summary> ...
    public class App
    {
        [STAThread]
        public static void Main( string[] args )
        {
            Console.WriteLine( "CS_Diagnostics" );
            Console.WriteLine();

            KtM9420 driver = null;

            // Pass in a command line argument as the resource descriptor, if none, will default
            string resource = "PXI0::23-0.0::INSTR"; // Use the hardware associated with the connection named "KtM9420"
            string options = "QueryInstrStatus=true, Simulate=false, DriverSetup= ";
            if( args.Length > 0 )
            {
                resource = args[ 0 ];
                options = "QueryInstrStatus=true, Simulate=false, DriverSetup= ";
            }

            try
            {
                // Create driver instance
                driver = new KtM9420();

                Initialize Driver Instances

                Check for Errors

                Print Driver Properties

            }
            catch( Exception ex )
            {
                Console.WriteLine( ex.Message );
            }
            finally
            {
                Close Driver Instances
            }
            Console.WriteLine( "" );
            Console.WriteLine( "\nDone - Press Enter to Exit" );
            Console.ReadLine();
        }
    }
}
```

Example Program 1 – How to Print Driver Properties, Check for Errors, and Close Driver Sessions

```
// Copy the following example code and compile it as a C# Console Application
// Example__VxtProperties.cs
#region Specify using Directives
```

```
using System;
using Keysight.KtM9420.Interop;
#endregion

namespace CS_Diagnostics
{
    /// <summary>
    ///
    /// Keysight IVI-C Driver Example Program
    ///
    /// Initializes the driver, reads a few Identity interface
    /// properties, and initiates instrument specific functionality.
    /// Runs in simulation mode without an instrument.
    ///
    /// Requires a COM reference to the driver's type library.
    ///
    /// </summary>
    public class App
    {
        [STAThread]
        public static void Main( string[] args )
        {
            Console.WriteLine( "CS_Diagnostics" );
            Console.WriteLine();

            KtM9420 driver = null;

            // Pass in a command line argument as the resource descriptor, if
none, will default
            string resource = "PXI0::23-0.0::INSTR"; // Use the hardware
associated with the connection named "KtM9420"
            string options = "QueryInstrStatus=true, Simulate=false,
DriverSetup= ";
            if( args.Length > 0 )
            {
                resource = args[ 0 ];
                options = "QueryInstrStatus=true, Simulate=false,
DriverSetup= ";
            }

            try
            {
                // Create driver instance
                driver = new KtM9420();

                #region Initialize Driver Instances
                const bool idquery = true;
                const bool reset = true;

                // Initialize the driver.  See driver help topic
"Initializing the IVI-COM Driver" for additional information
                driver.Initialize( resource, idquery, reset, options );

                #endregion

                #region Check for Errors
                int errorcode = 0;
                string message = string.Empty;

                // Clear startup messages and warnings if any.
                do
                {
                    driver.Utility.ErrorQuery( ref errorcode, ref message );
                    if( errorcode != 0 )
                    {
                        Console.WriteLine( message );
                    }
                } while( errorcode != 0 );

                Console.WriteLine( "Driver Initialized" );

                #endregion

                #region Print Driver Properties

                Console.WriteLine("Identifier:  {0}",
driver.Identity.Identifier);
                Console.WriteLine("Revision:    {0}",
driver.Identity.Revision);
                Console.WriteLine("Vendor:      {0}",
driver.Identity.Vendor);
                Console.WriteLine("Description: {0}",
driver.Identity.Description);
                Console.WriteLine("Model:       {0}",
driver.Identity.InstrumentModel);
                Console.WriteLine("FirmwareRev: {0}",
driver.Identity.InstrumentFirmwareRevision);
                Console.WriteLine("Serial #:    {0}",
driver.System.SerialNumber);
                Console.WriteLine("Simulate:    {0}",
driver.DriverOperation.Simulate);
                Console.WriteLine();
                #endregion

            }
            catch( Exception ex )
            {
                Console.WriteLine( ex.Message );
            }
            finally
            {
                #region Close Driver Instances
                if ( driver != null && driver.Initialized )
                {
                    // Close the driver
                    driver.Close();
                    Console.WriteLine( "" );
                    Console.WriteLine( "Driver Closed" );
                }
                #endregion
            }
            Console.WriteLine( "" );
            Console.WriteLine( "\nDone - Press Enter to Exit" );
            Console.ReadLine();
        }
    }
}
```

Working with PA_FEM Measurements

The RF front end of a product includes all of the components between an antenna and the baseband device. The purpose of an RF front end is to upconvert a baseband signal to RF that can be used for transmission by an antenna. An RF front end can also be used to downconvert an RF signal that can be processed with ADC circuitry. As an example, the RF signal that is received by a cellular phone is the input into the front end circuitry and the output is a down-converted analog signal in the intermediate frequency (IF) range. This down-converted signal is the input to a baseband device, an ADC. For the transmit side, a DAC generates the signal to be up-converted, amplified, and sent to the antenna for transmission. Depending on whether the system is a Wi-Fi, GPS, or cellular radio will require different characteristics of the front end devices.

RF front end devices fall into a few major categories: RF Power Amplifiers, RF Filters and Switches, and FEMs [Front End Modules].

- RF Power Amplifiers and RF Filters and Switches typically require the following:
  - PA[Power Amplifier] – Production Tests which include:
    - Channel Power - Power Acquisition Mode is used to return one value back through the API.
    - ACPR [Adjacent Channel Power Ratio] – When making fast ACPR measurements, "Baseband Tuning" is used to digitally tune the center frequency in order to make channel power measurements, at multiple offsets, using the Power Acquisition interface.
    - Servo Loop- When measuring a power amplifier, one of the key measurements is performing a Servo Loop because when you measure a power amplifier:
      - it is typically specified at a specific output power
      - there is a need to adjust the source input level until you measure the exact power level - to do this, you will continually adjust the source until you achieve the specified output power then you make all of the ACPR and harmonic parametric measurements at that level.
- FEMs [Front End Modules] – which could be a combination of multiple front end functions in a single module or even a "Switch Matrix" that switches various radios (such as Wi-Fi, GSM, PCS, Bluetooth, etc.) to the antenna.

Test Challenges Faced by Power Amplifier Testing

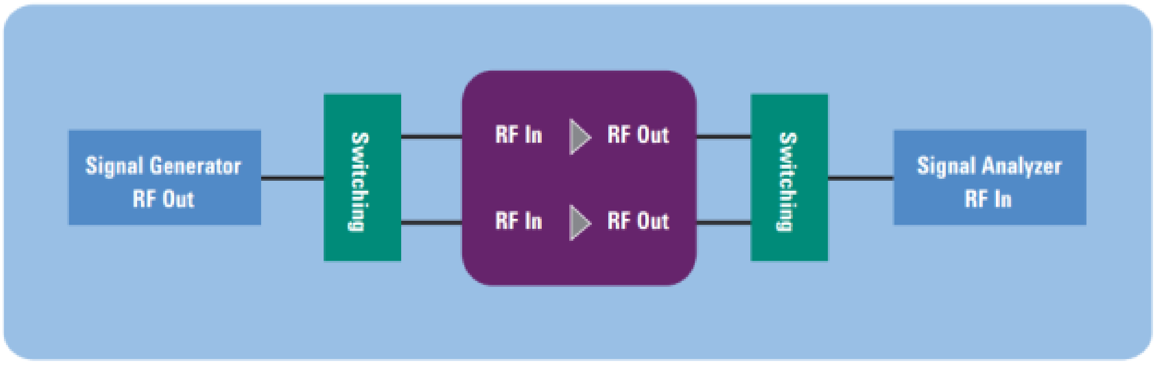
The following are the test challenges faced by Power Amplifier Testing:

- The need to quickly adjust power level inputs to the device under test (DUT).
- The need to assess modulation performance (i.e., ACPR and EVM) at high output power levels.

The figure below shows a simplified block diagram for the M9420 VXT Vector Transceiver in a typical PA / FEM test system.

Typical power amplifier modules require an input power level of 0 to + 5 dBm, digitally modulated according to communication standards such as WCDMA or LTE. The specified performance of the power amplifier or front end module is normally set at a specific output level of the DUT. If the devices have small variations in gain, it may be necessary to adjust the power level from the source to get the correct output level of the DUT. Only after the DUT output level is set at the correct value can the specified parameters be tested. The time spent adjusting the source to get the correct DUT output power can be a major contributor to the test time and the overall cost of test.

The source is connected to the DUT using a cable and switches. The switching may be used to support testing of multi-band modules or multi-site testing. The complexity of the switching depends on the number of bands in the devices and the number of test sites supported by the system. The DUTs are typically inserted into the test fixture using an automated part handler. In some cases, several feet of cable is required between the source and the input of the DUT.



The combination of the RF cables and the switching network can add several dB of loss between the output of the source and the input of the DUT, which requires higher output levels from the source. Since the tests are performed with a modulated signal, the source must also have adequate modulation performance at the higher power levels.

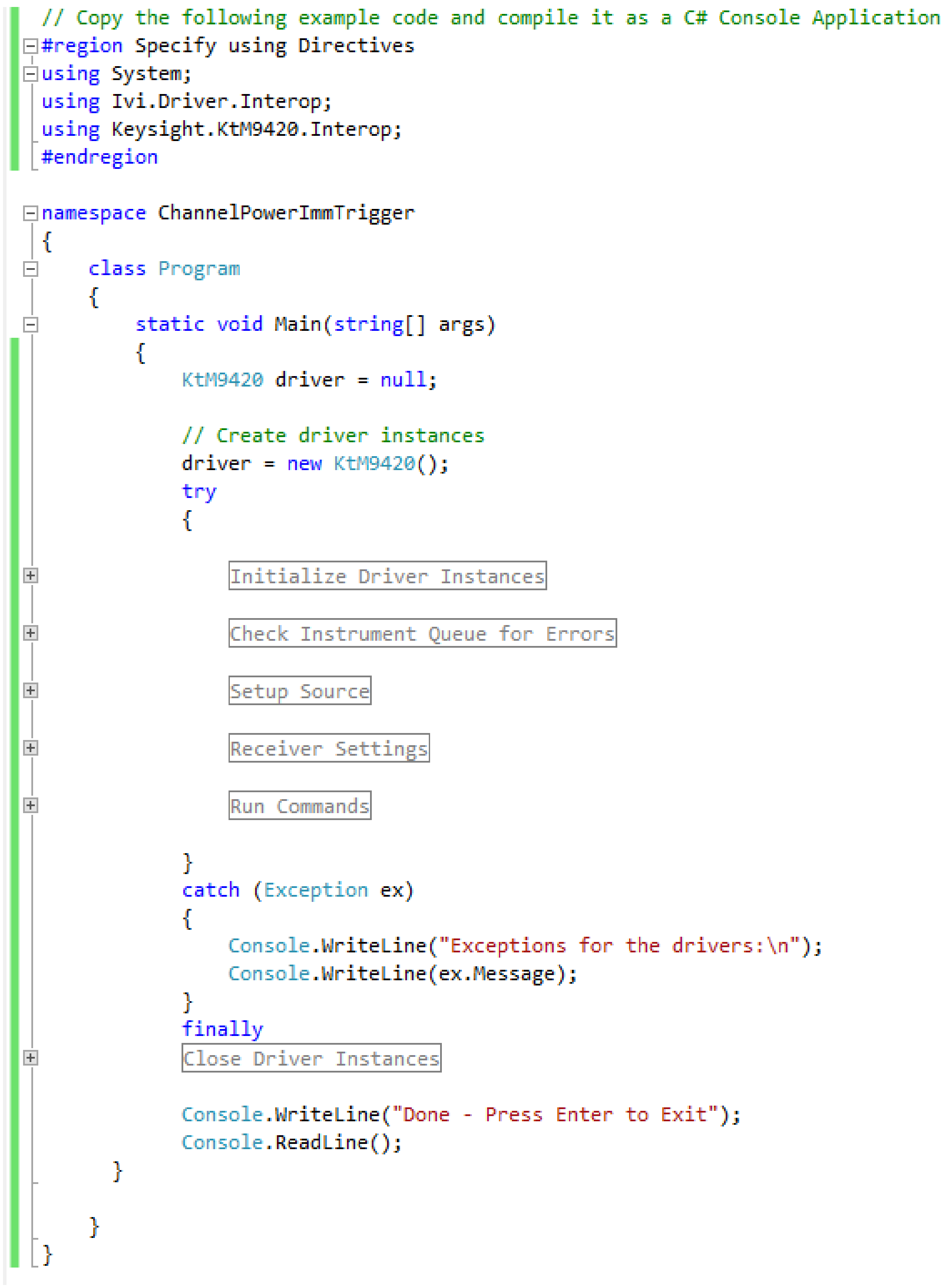
Performing a Channel Power Measurement, Using Immediate Trigger

Standard	**Sample Rate**	**Channel Filter Type**	**Channel Filter Parameter**	**Channel Filter Bandwidth**	**Channel Offsets**
WCDMA	5 MHz	RRC	0.22	3.84 MHz	5, 10 MHz
LTE 10 MHz FDD	11.25 MHz	Rectangular	N/A	9 MHz	10, 20 MHz
LTE 10 MHz TDD	11.25 MHz	Rectangular	N/A	9 MHz	10, 20 MHz
1xEV-DO	2 MHz	RRC	0.22	1.23 MHz	1.25, 2.5 MHz
TD-SCDMA	2 MHz	RRC	0.22	1.28 MHz	1.6, 3.2 MHz
GSM/EDGE Channel	1.25 MHz	Gaussian	0.3	271 kHz	
GSM/EDGEORFS	1.25 MHz	TBD	TBD	30 kHz	400, 600kHz

Example Program 2 – Code Structure

The following example code demonstrates how to instantiate a driver instance, set the resource name and various initialization values, initialize the driver instances, and perform other relevant tasks:

1. Send Source GeneratePowerRampArb, LoadWaveform and Modulation commands to the M9420A VXT driver.
2. Send Receiver RF and Power Acquisition commands to the M9420A VXT driver and Apply changes to hardware,
3. Check the instrument queue for errors.
4. Perform a Channel Power Measurement,
5. Report errors if any occur, and close the driver.



```
// Copy the following example code and compile it as a C# Console Application
#region Specify using Directives
using System;
using Ivi.Driver.Interop;
using Keysight.KtM9420.Interop;
#endregion

namespace ChannelPowerImmTrigger
{
    class Program
    {
        static void Main(string[] args)
        {
            KtM9420 driver = null;

            // Create driver instances
            driver = new KtM9420();
            try
            {

                Initialize Driver Instances

                Check Instrument Queue for Errors

                Setup Source

                Receiver Settings

                Run Commands

            }
            catch (Exception ex)
            {
                Console.WriteLine("Exceptions for the drivers:\n");
                Console.WriteLine(ex.Message);
            }
            finally
            Close Driver Instances

            Console.WriteLine("Done - Press Enter to Exit");
            Console.ReadLine();
        }

    }
}
```

Example Program 2 - Pseudo-code

Initialize Driver for VXT, Check for Errors

- Send RF Settings to VXT Driver:
  - Frequency
  - Level
  - Peak to Average Ratio
  - Conversion Mode
  - IF Bandwidth
  - Set Acquisition Mode to "Power"
- Send Power Acquisition Setting to VXT Driver:
  - Sample Rate
  - Duration
  - Channel Filter
- Apply Method to Send Changes to Hardware
  - Wait for Hardware to Settle
- Send Arm Method to VXT
- Send Read Power Method to VXT

Close Driver for VXT

Example Program 2 - Channel Power Measurement Using Immediate Trigger

```
// Copy the following example code and compile it as a C# Console Application
#region Specify using Directives
using System;
using Ivi.Driver.Interop;
using Keysight.KtM9420.Interop;
#endregion

namespace ChannelPowerImmTrigger
{
    class Program
    {
        static void Main(string[] args)
        {
            KtM9420 driver = null;

            // Create driver instances
            driver = new KtM9420();
            try
            {

                #region Initialize Driver Instances

                string ResourceName = "PXI0::23-0.0::INSTR";
                bool IdQuery = true;
                bool Reset = true;
                string OptionString = "QueryInstrStatus=true,
Simulate=false,DriverSetup= ";
                driver.Initialize(ResourceName, IdQuery, Reset,OptionString);
                Console.WriteLine("Driver Initialized\n");

                #endregion

                #region Check Instrument Queue for Errors

                int errorcode = 0;
                string message = string.Empty;
                // Check instrument for errors
                do
                {
                    driver.Utility.ErrorQuery( ref errorcode, ref message );
                    if( errorcode != 0 )
                    {
                        Console.WriteLine( message );
                    }
                } while( errorcode != 0 );

                #endregion

                #region Setup Source
                string testWaveform = "CW";
                driver.Source.GeneratePowerRampArb("CW", 0, 0, 1e-3, 25e6);
                string WaveformfilePath = "C:\\Program Files\\Keysight\\X-
Series\\MTRX\\Infrastructure\\Waveform";
                driver.Source.LoadWaveform(WaveformfilePath, testWaveform);
                driver.Source.Modulation.ArbPlayConfigure(
                        WaveformName: testWaveform,
                        ArbPlayMode:
KtM9420ArbPlayModeEnum.KtM9420ArbPlayModePlayArb,
                        ArbPlayDuration: 1e-4
                    );
                #endregion

                #region Receiver Settings
                // Receiver Settings
                double Frequency = 2000000000.0;
                double Level = 5;
                double RmsValue = 5;
                double ChannelTime = 0.0001;
                double MeasureBW = 5000000.0;
                KtM9420ChannelFilterShapeEnum FilterType =
KtM9420ChannelFilterShapeEnum.KtM9420ChannelFilterShapeRaisedCosine;
                double FilterAlpha = 0.22;
                double FilterBw = 3840000.0;
                double MeasuredPower = 0;
                bool Overload = true;
                #endregion

                #region Run Commands
                // Setup the RF Path in the Receiver
                driver.Receiver.RF.Frequency = Frequency;
                driver.Receiver.RF.Power = Level;
                driver.Receiver.RF.PeakerToAverage = RmsValue;
                               // Configure the Acquisition
                driver.AcquisitionMode =
KtM9420AcquisitionModeEnum.KtM9420AcquisitionModePower;
                driver.PowerAcquisition.Bandwidth = MeasureBW; // 5 MHz
                driver.PowerAcquisition.Duration = ChannelTime; // 100 us
                driver.PowerAcquisition.ChannelFilter.Configure
(FilterType,FilterAlpha, FilterBw);
                // Send Changes to hardware
                driver.Apply();

                string response = "y";
                while (string.Compare(response, "y") == 0)
                {
                    Console.WriteLine("Press Enter to Run Test");
                    Console.ReadLine();
                    driver.Arm();
                    driver.WaitForData(100);

                    #region Check for error
                    errorcode = 0;
                    message = string.Empty;
                    // Check instrument for errors
                    do
                    {
                        driver.Utility.ErrorQuery(ref errorcode, ref
message);
                        if (errorcode != 0)
                        {
                            Console.WriteLine(message);
                        }
                    } while (errorcode != 0);
                    #endregion

                    driver.PowerAcquisition.ReadPower(0, ref MeasuredPower,
ref Overload);
                    Console.WriteLine("Measured Power: " + MeasuredPower +
"dBm");
                    Console.WriteLine(String.Format("Overload = {0}",
Overload ? "true" : "false"));
                    Console.WriteLine("Repeat? y/n");
                    response = Console.ReadLine();
                }
                #endregion

            }
            catch (Exception ex)
            {
                Console.WriteLine("Exceptions for the drivers:\n");
                Console.WriteLine(ex.Message);
            }
            finally
            #region Close Driver Instances
            {
                if (driver != null && driver.Initialized)
                {
                // Close the driver
                driver.Close();
                Console.WriteLine("Driver Closed\n");
                }
            }
            #endregion

            Console.WriteLine("Done - Press Enter to Exit");
            Console.ReadLine();
        }

    }
}
```

Performing a WCDMA Power Servo and ACPR Measurement

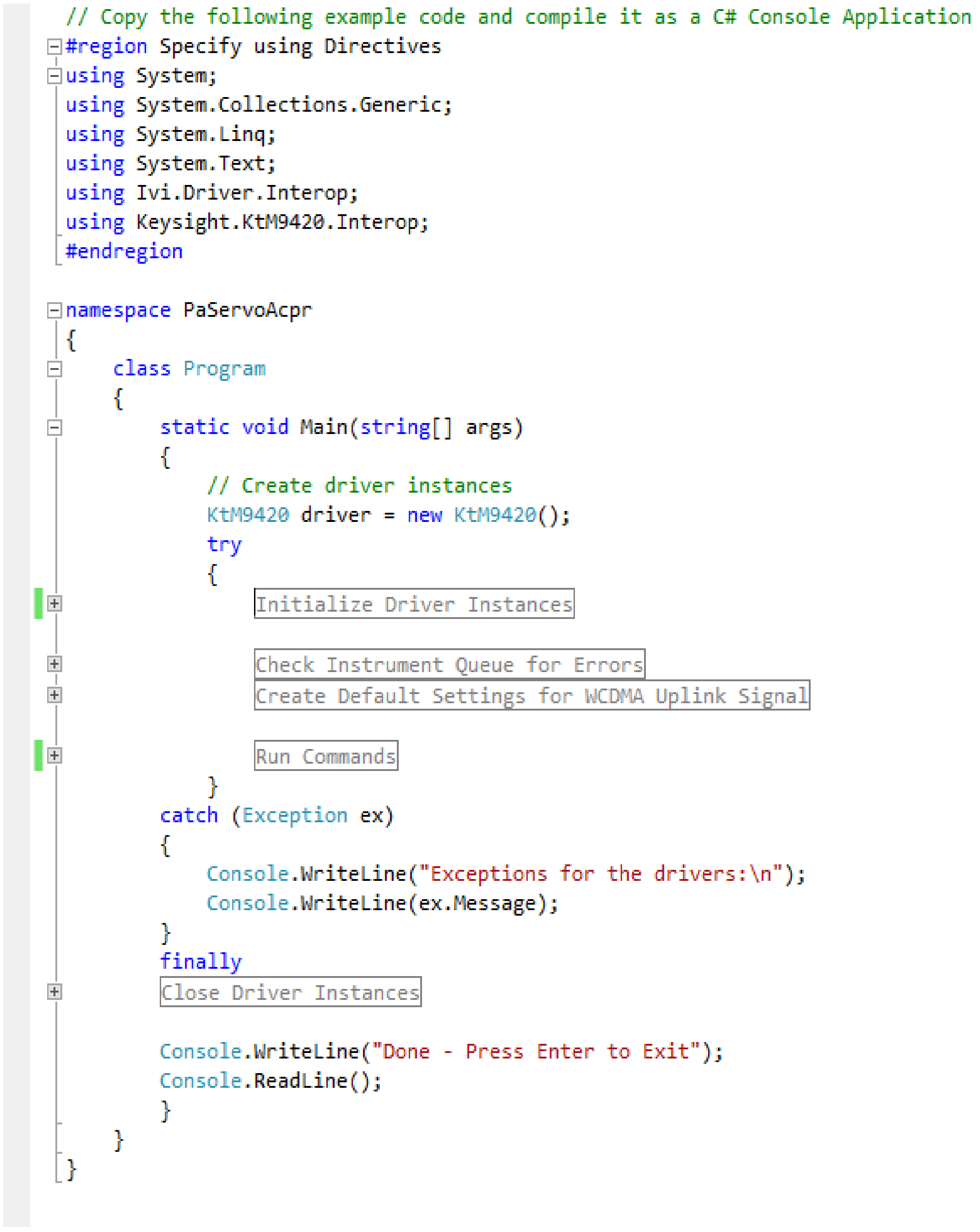
When making a WCDMA Power Servo and ACPR measurement, Servo is performed using "Baseband Tuning" to adjust the source amplitude and then "Baseband Tuning" is used to digitally tune the center frequency in order to make channel power measurements, at multiple offsets, using the Power Servo interface of the M9420A VXT.

Example Program 3 - Code Structure

The following example code demonstrates how to instantiate driver instances, set the resource names and various initialization values, initialize the driver instances, and perform the other relevant tasks:

1. Send Source RF and LoadWaveform commands to the M9420 VXT driver,
2. Send Receiver RF commands to the M9420 VXT driver,
3. Send Power Servo and ACPR configuration command to M9420 VXT driver,

4. Send Meaurement Process command to run a Servo Loop and ACPR measurementnt,
5. Read Power Servo and ACPR Measurement result,
6. Close the driver.



```
// Copy the following example code and compile it as a C# Console Application
#region Specify using Directives
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ivi.Driver.Interop;
using Keysight.KtM9420.Interop;
#endregion

namespace PaServoAcpr
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            KtM9420 driver = new KtM9420();
            try
            {
                Initialize Driver Instances

                Check Instrument Queue for Errors
                Create Default Settings for WCDMA Uplink Signal

                Run Commands
            }
        catch (Exception ex)
        {
            Console.WriteLine("Exceptions for the drivers:\n");
            Console.WriteLine(ex.Message);
        }
        finally
        Close Driver Instances

        Console.WriteLine("Done - Press Enter to Exit");
        Console.ReadLine();
        }
    }
}
```

Example Program 3 - Pseudo-code

Initialize Drivers for VXT and check for errors

- Configure Source RF Settings:
    - Frequency
    - RF Level
    - RF Output Port and Enable On
- Configure ARBPLAY Settings:
    - Load WCDMA Signal Studio File
    - Get RMS Value
    - Play ARB File
- Configure Receiver RF Settings:
    - Frequency
    - Level
    - Peak to Average Ratio
    - Input Port
- Configure Power Servo Settings
    - Enable Power Servo Measurement
    - Acquisition Mode
    - Acquisition Settings
    - Power Servo Settings
- Configure ACPR Settings
    - Enable ACPR Measurement
    - ACPR Measurement Settings
- Enable VXT Settings:
    - Source Settings
    - Receiver Settings
- Apply All Above Settings and Measurements
- Read Power Servo Results
    - Measured Power
    - Pass/Fail
    - Overload
    - Servo Count
- Read ACPR Results
    - ACPR Values
    - Overload

Example Program 3 - WCDMA Power Servo and ACPR Measurement

```
// Copy the following example code and compile it as a C# Console Application
#region Specify using Directives
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ivi.Driver.Interop;
using Keysight.KtM9420.Interop;
#endregion

namespace PaServoAcpr
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            KtM9420 driver = new KtM9420();
            try
            {
                #region Initialize Driver Instances

                string ResourceName = "PXI0::23-0.0::INSTR";
                bool IdQuery = true;
                bool Reset = true;
                string OptionString = "QueryInstrStatus=true,
Simulate=false,DriverSetup= ";
                driver.Initialize(ResourceName, IdQuery, Reset,OptionString);
                Console.WriteLine("Driver Initialized\n");

                #endregion

                #region Check Instrument Queue for Errors

                int errorcode = 0;
                string message = string.Empty;
                // Check instrument for errors
                do
                {
                    driver.Utility.ErrorQuery( ref errorcode, ref message );
                    if( errorcode != 0 )
                    {
                        Console.WriteLine( message );
                    }
                } while( errorcode != 0 );

                #endregion
                #region Create Default Settings for WCDMA Uplink Signal
                // Source Settings
                double Frequency = 1000000000.0;
                double Level = 3;
                double Gain = 0;
                double PowerOutMargin = 0.05;
                double ServoOverheadTime = 600e-6;
                // If a Signal Studio waveform file is used, it may require a
software license.
                string ExamplesFolder = "C:\\Program Files (x86)
\\Keysight\\M9420\\Example Waveforms\\";
                string WaveformFile = "WCDMA_UL_DPCHH_2DPDCH_1C.wfm";
                // Receiver Settings
                double ChannelTime = 0.0001;
                double AdjacentTime = 0.0005;
                double IfBandwidth = 40000000.0;
                double MeasureBW = 5000000.0;
                KtM9420ChannelFilterShapeEnum FilterType =
KtM9420ChannelFilterShapeEnum.KtM9420ChannelFilterShapeRaisedCosine;
                double FilterAlpha = 0.22;
                double FilterBw = 3840000.0;
                double AcprFliterBw = 3840000.0;
                double AcprFilterAlpha = 0.22;
                KtM9420ChannelFilterShapeEnum AcprFilterType =
KtM9420ChannelFilterShapeEnum.KtM9420ChannelFilterShapeRaisedCosine;
                double[] FreqOffset = new double[] {-5000000.0, 5000000.0, -
10000000.0, 10000000.0};
                double[] acprFilterAlpha = new double[4] {AcprFilterAlpha,
AcprFilterAlpha, AcprFilterAlpha, AcprFilterAlpha};
                double[] acprFilterBw = new double[4] {AcprFliterBw,
AcprFliterBw, AcprFliterBw, AcprFliterBw};
                KtM9420ChannelFilterShapeEnum[] acprFilterType = new
KtM9420ChannelFilterShapeEnum[4] {AcprFilterType, AcprFilterType,
AcprFilterType, AcprFilterType};
                double AcprSpan = 30.72e6 / 1.25;
                double AcprDuration = AdjacentTime;
                double[] acprSpan = new double[4]{AcprSpan, AcprSpan,
AcprSpan, AcprSpan};
                double[] acprDuration  = new double[4]{AcprDuration,
AcprDuration, AcprDuration, AcprDuration};
                double MeasuredPower = 0;
                               bool ServoPass = false;
                int ServoCount = 0;
                bool Overload = true;
                double[] MeasuredACPR = new double[4];
                bool[] MeasuredACPROverload = new bool[4];
                double RmsValue = 0;
                #endregion

                #region Run Commands
                //Setup Source
                driver.Source.RF.Frequency = Frequency;
                driver.Source.RF.Level = Level;
                driver.Source.RF.OutputPort =
KtM9420PortEnum.KtM9420PortRFOutput;
```

```
                driver.Source.RF.OutputEnable = true;
                driver.Source.LoadWaveform(ExamplesFolder, WaveformFile);
                RmsValue = driver.Source.Modulation.ArbRmsValue;
                driver.Source.Modulation.ArbPlayConfigure(
                        WaveformName: WaveformFile,
                        ArbPlayMode:
KtM9420ArbPlayModeEnum.KtM9420ArbPlayModePlayArb,
                        ArbPlayDuration: 1e-4
                    );

                // Setup Receiver
                driver.Receiver.RF.Frequency = Frequency;
                driver.Receiver.RF.Power = Level;
                driver.Receiver.RF.PeakerToAverage = RmsValue;
                driver.Receiver.RF.InputPort =
KtM9420PortEnum.KtM9420PortRFInput;

                // Configure PowerServo
                driver.Measurement.EnabledMeasurements = (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsPowerServo;
                driver.Measurement.PowerServo.AcqusitionMode =
KtM9420AcquisitionModeEnum.KtM9420AcquisitionModeFFT;

                driver.FFTAcquisition.SampleRate = MeasureBW*1.25;
                driver.FFTAcquisition.Length =
KtM9420FFTAcquisitionLengthEnum.KtM9420FFTAcquisitionLength_512;
                driver.FFTAcquisition.Duration = ChannelTime;
                driver.FFTAcquisition.ChannelFilter.Configure
(FilterType,FilterAlpha, FilterBw);

                driver.Measurement.PowerServo.InputPower = Level + Gain;
                driver.Measurement.PowerServo.OutputPower = Level;
                driver.Measurement.PowerServo.OutputPowerMargin =
PowerOutMargin;
                driver.Measurement.PowerServo.OverheadTime =
ServoOverheadTime;
                driver.Measurement.PowerServo.MaximumOutputPower = 20;

                //Configure Acpr
                driver.Measurement.EnabledMeasurements |= (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsAcpr;
                driver.Measurement.Acpr.AcquisitionMode =
KtM9420AcquisitionModeEnum.KtM9420AcquisitionModeFFT;
                driver.Measurement.Acpr.UseChanPwrForRef = true;
                driver.Measurement.Acpr.ConfigureFilter
(acprFilterType,acprFilterAlpha,acprFilterBw);
                driver.Measurement.Acpr.SetAcprParameter
(FreqOffset,acprSpan,acprDuration);

                //Setup all hardware in one time.
                driver.Measurement.EnabledMeasurements |= (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsSetupVsa;
                driver.Measurement.EnabledMeasurements |= (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsSetupVsaFrequency;
                driver.Measurement.EnabledMeasurements |= (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsSetupVsg;
                driver.Measurement.EnabledMeasurements |= (int)
KtM9420MeasurementsEnum.KtM9420MeasurementsSetupVsgFrequency;

                string response = "y";
                while (string.Compare(response, "y") == 0)
                {
                Console.WriteLine("Press Enter to Run Test");
                Console.ReadLine();

                //Process measurement
                driver.Measurement.Process();

                // Check instrument for errors
                do
                {
                    driver.Utility.ErrorQuery( ref errorcode, ref message );
                    if( errorcode != 0 )
                    {
                        Console.WriteLine( message );
                    }
                } while( errorcode != 0 );

                //Read PowerServo
                driver.Measurement.PowerServo.ReadPowerServo(ref
MeasuredPower,ref ServoPass, ref Overload, ref ServoCount);
                Console.WriteLine("Measured Power {0}dbm , Servo pass is {1},
Servo Count is {2}, Servo Overload is {3}",
                MeasuredPower, ServoPass, ServoCount, Overload);

                driver.Measurement.Acpr.ReadAcpr(ref MeasuredACPR, ref
MeasuredACPROverload);

                Console.WriteLine("ACPR1 L: {0} dBc, Overload is {1}",
MeasuredACPR[0], MeasuredACPROverload[0]);
                Console.WriteLine("ACPR1 U: {0} dBc, Overload is {1}",
MeasuredACPR[1], MeasuredACPROverload[1]);
                Console.WriteLine("ACPR2 L: {0} dBc, Overload is {1}",
MeasuredACPR[2], MeasuredACPROverload[2]);
                Console.WriteLine("ACPR2 U: {0} dBc, Overload is {1}",
MeasuredACPR[3], MeasuredACPROverload[3]);

                Console.WriteLine("Repeat? y/n");
                response = Console.ReadLine();
                }
                #endregion
            }
        catch (Exception ex)
        {
        Console.WriteLine("Exceptions for the drivers:\n");
        Console.WriteLine(ex.Message);
        }
        finally
        #region Close Driver Instances
        {
            if (driver != null && driver.Initialized)
            {
                // Close the driver
                driver.Close();
                Console.WriteLine("Driver Closed");
            }
        }
        #endregion

        Console.WriteLine("Done - Press Enter to Exit");
        Console.ReadLine();
        }
    }
}
```

References

- Understanding Drivers and Direct I/O, Application Note 1465-3 (Agilent Part Number: 5989-0110EN)
- Digital Baseband Tuning Technique Speeds Up Testing, by Bill Anklam, Victor Grothen and Doug Olney, Agilent Technologies, Santa Clara, CA, April 15, 2013, Microwave Journal
- Accelerate Development of Next Generation 802.11ac Wireless LAN Transmitters-Overview, Application Note (Agilent Part Number: 5990-9872EN)
- www.ivifoundation.org

Glossary

- **ADE** (application development environment) – An integrated suite of software development programs. ADEs may include a text editor, compiler, and debugger, as well as other tools used in creating, maintaining, and debugging application programs. Example: Microsoft Visual Studio.
- **API** (application programming interface) – An API is a well-defined set of set of software routines through which application program can access the functions and services provided by an underlying operating system or library. Example: IVI Drivers
- **C#** (pronounced "C sharp") – C-like, component-oriented language that eliminates much of the difficulty associated with C/C++.
- **Direct I/O** – commands sent directly to an instrument, without the benefit of, or interference from a driver. SCPI Example: SENSe:VOLTage:RANGe:AUTO Driver (or device driver) – a collection of functions resident on a computer and used to control a peripheral device.
- **DLL** (dynamic link library) – An executable program or data file bound to an application program and loaded only when needed, thereby reducing memory requirements. The functions or data in a DLL can be simultaneously shared by several applications.
- **Input/Output (I/O)** layer – The software that collects data from and issues commands to peripheral devices. The VISA function library is an example of an I/O layer that allows application programs and drivers to access peripheral instrumentation.
- **IVI** (Interchangeable Virtual Instruments) – a standard instrument driver model defined by the IVI Foundation that enables engineers to exchange instruments made by different manufacturers without rewriting their code. www.ivifoundation.org
- **IVI COM** drivers (also known as IVI Component drivers) – IVI COM presents the IVI driver as a COM object in Visual Basic. You get all the intelligence and all the benefits of the development environment because IVI COM does things in a smart way and presents an easier, more consistent way to send commands to an instrument. It is similar across multiple instruments.
- **Microsoft COM** (Component Object Model) – The concept of software components is analogous to that of hardware components: as long as components present the same interface and perform the same functions, they are interchangeable. Software components are the natural extension of DLLs. Microsoft developed the COM standard to allow software manufacturers to create new software components that can be used with an existing application program, without requiring that the application be rebuilt. It is this capability

that allows T&M instruments and their COM-based IVI-Component drivers to be interchanged.

- **.NET Framework** — The .NET Framework is an object-oriented API that simplifies application development in a Windows environment. The .NET Framework has two main components: the common language runtime and the .NET Framework class library.
- **VISA** (Virtual Instrument Software Architecture) — The VISA standard was created by the VXIplug&play Foundation. Drivers that conform to the VXIplug&play standards always perform I/O through the VISA library. Therefore if you are using Plug and Play drivers, you will need the VISA I/O library. The VISA standard was intended to provide a common set of function calls that are similar across physical interfaces. In practice, VISA libraries tend to be specific to the vendor's interface.
- **VISA-COM** — The VISA-COM library is a COM interface for I/O that was developed as a companion to the VISA specification. VISA-COM I/O provides the services of VISA in a COM-based API. VISA-COM includes some higher-level services that are not available in VISA, but in terms of low-level I/O communication capabilities, VISA-COM is a subset of VISA. Agilent VISA-COM is used by its IVI-Component drivers and requires that Agilent VISA also be installed.