Programming Guide

Keysight M9391A PXIe Vector Signal Analyzer & M9381A PXIe Vector Signal Generator

Notice: This document contains references to Agilent. Please note that Agilent's Test and Measurement business has become Keysight Technologies. For more information, go to www.keysight.com.

KEYSIGHT TECHNOLOGIES

Notices

Copyright Notice

Manual Part Number

M9300-90080

Published By

Keysight Technologies
Ground Floor and Second Floor, CP-11
Sector-8, IMT Manesar – 122051
Gurgaon, Haryana, India

Edition

Edition 2.1, July, 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.

Warranty

Technology Licenses

U.S. Government Rights

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 safety precautions required to avoid possible injury. Read and follow all installation, operation, and maintenance information carefully before using the product.

WARNING

If this product is not used as specified, the protection provided by the equipment could be impaired. This product must be used in a normal condition (in which all means for protection are intact) only.

The types of product users are:

- Responsible body is the individual or group responsible for the use and maintenance of equipment, for ensuring that the equipment is operated within its specifications and operating limits, and for ensuring operators are adequately trained.
- Operators use the product for its intended function. They must be trained in electrical safety procedures and proper use of the instrument. They must be protected from electric shock and contact with hazardous live circuits.
- Maintenance personnel perform routine procedures on the product to keep it operating properly (for example, setting the line voltage or replacing consumable materials). Maintenance procedures are described in the user documentation. The procedures explicitly state if the operator may perform them. Otherwise, they should be performed only by service personnel.
- Service personnel are trained to work on live circuits, perform safe installations, and repair products. Only properly trained service personnel may perform installation and service procedures.

WARNING

Operator is responsible to maintain safe operating conditions. To ensure safe operating conditions, modules should not be operated beyond the full temperature range specified in the Environmental and physical specification. Exceeding safe operating conditions can result in shorter lifespans, improper module performance and user safety issues. When the modules are in use and operation within the specified full temperature range is not maintained, module surface temperatures may exceed safe handling conditions which can cause discomfort or burns if touched. In the event of a module exceeding the full temperature range, always allow the module to cool before touching or removing modules from chassis.

Keysight products are designed for use with electrical signals that are rated Measurement Category I and Measurement Category II, as described in the International Electrotechnical Commission (IEC) Standard IEC 60664. Most measurement, control, and data I/O signals are Measurement Category I and must not be directly connected to mains voltage or to voltage sources with high transient over-voltages. Measurement Category II connections require protection for high transient over-voltages often associated with local AC mains connections. Assume all measurement, control, and data I/O connections are for connection to Category I sources unless otherwise marked or described in the user documentation.

Exercise extreme caution when a shock hazard is present. Lethal voltage may be present on cable connector jacks or test fixtures. The American National Standards Institute (ANSI) states that a shock hazard exists when voltage levels greater than 30V RMS, 42.4V peak, or 60VDC are present. A good safety practice is to expect that hazardous voltage is present in any unknown circuit before measuring.

Operators of this product must be protected from electric shock at all times. The responsible body must ensure that operators are prevented access and/or insulated from every connection point. In some cases, connections must be exposed to potential human contact. Product operators in these circumstances must be trained to protect themselves from the risk of electric shock. If the circuit is capable of operating at or above 1000V, no conductive part of the circuit may be exposed.

Do not connect switching cards directly to unlimited power circuits. They are intended to be used with impedance-limited sources. NEVER connect switching cards directly to AC mains. When connecting sources to switching cards, install protective devices to limit fault current and voltage to the card.

Before operating an instrument, ensure that the line cord is connected to a properly-grounded power receptacle. Inspect the connecting cables, test leads, and jumpers for possible wear, cracks, or breaks before each use.

When installing equipment where access to the main power cord is restricted, such as rack mounting, a separate main input power disconnect device must be provided in close proximity to the equipment and within easy reach of the operator.

For maximum safety, do not touch the product, test cables, or any other instruments while power is applied to the circuit under test. ALWAYS remove power from the entire test system and discharge any capacitors before: connecting or disconnecting cables or jumpers, installing or removing switching cards, or making internal changes, such as installing or removing jumpers.

Do not touch any object that could provide a current path to the common side of the circuit under test or power line (earth) ground. Always make measurements with dry hands while standing on a dry, insulated surface capable of withstanding the voltage being measured.

The instrument and accessories must be used in accordance with its specifications and operating instructions, or the safety of the equipment may be impaired.

Do not exceed the maximum signal levels of the instruments and accessories, as defined in the specifications and operating information, and as shown on the instrument or test fixture panels, or switching card.

When fuses are used in a product, replace with the same type and rating for continued protection against fire hazard.

Chassis connections must only be used as shield connections for measuring circuits, NOT as safety earth ground connections.

If you are using a test fixture, keep the lid closed while power is applied to the device under test. Safe operation requires the use of a lid interlock.

Instrumentation and accessories shall not be connected to humans.

Before performing any maintenance, disconnect the line cord and all test cables.

To maintain protection from electric shock and fire, replacement components in mains circuits – including the power transformer, test leads, and input jacks – must be purchased from Keysight. Standard fuses with applicable national safety approvals may be used if the rating and type are the same. Other components that are not safety-related may be purchased from other suppliers as long as they are equivalent to the original component (note that selected parts should be purchased only through Keysight to maintain accuracy and functionality of the product). If you are unsure about the applicability of a replacement component, call an Keysight office for information.

WARNING

No operator serviceable parts inside. Refer servicing to qualified personnel. To prevent electrical shock do not remove covers. For continued protection against fire hazard, replace fuse with same type and rating.

PRODUCT MARKINGS:

The CE mark is a registered trademark of the European Community.

Australian Communication and Media Authority mark to indicate regulatory compliance as a registered supplier.

ICES/NMB-001
ISM GRP.1 CLASS A

This symbol indicates product compliance with the Canadian Interference-Causing Equipment Standard (ICES-001). It also identifies the product is an Industrial Scientific and Medical Group 1 Class A product (CISPR 11, Clause 4).

KCC-REM-KST-BLMxxxxx

South Korean Class A EMC Declaration. This equipment is Class A suitable for professional use and is for use in electromagnetic environments outside of the home. A 급 기기 (업무용 방송통신기자재) 이 기기는 업무용 (A 급) 전자파적합기기로서 판 매자 또는 사용자는 이 점을 주 의하시기 바라 며 , 가정외의 지역에서 사용하는 것을 목적으 로 합니다 .

This product complies with the WEEE Directive marketing requirement. The affixed product label (above) indicates that you must not discard this electrical/electronic product in domestic household waste. **Product Category**: With reference to the equipment types in the WEEE directive Annex 1, this product is classified as "Monitoring and Control instrumentation" product. Do not dispose in domestic household waste. To return unwanted products, contact your local Keysight office, or for more information see http://about.keysight.com/en/companyinfo/environment/takeback.shtml.

This symbol indicates the instrument is sensitive to electrostatic discharge (ESD). ESD can damage the highly sensitive components in your instrument. ESD damage is most likely to occur as the module is being installed or when cables are connected or disconnected. Protect the circuits from ESD damage by wearing a grounding strap that provides a high resistance path to ground. Alternatively, ground yourself to discharge any built-up static charge by touching the outer shell of any grounded instrument chassis before touching the port connectors.

This symbol on an instrument means caution, risk of danger. You should refer to the operating instructions located in the user documentation in all cases where the symbol is marked on the instrument.

This symbol indicates the time period during which no hazardous or toxic substance elements are expected to leak or deteriorate during normal use. Forty years is the expected useful life of the product.

CLEANING PRECAUTIONS:

WARNING

To prevent electrical shock, disconnect the Keysight Technologies instrument from mains before cleaning. Use a dry cloth or one slightly dampened with water to clean the external case parts. Do not attempt to clean internally. To clean the connectors, use alcohol in a well-ventilated area. Allow all residual alcohol moisture to evaporate, and the fumes to dissipate prior to energizing the instrument.

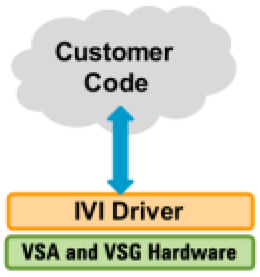
Contents

What You Will Learn in This Programming Guide

This programming guide is intended for individuals who write and run programs to control test-and-measurement instruments. Specifically, in this programming 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 M9391A PXIe Vector Signal Analyzer (VSA) and Keysight M9381A PXIe Vector Signal Generator (VSG) while performing PA/FEM Power Measurement Production Tests
- Generate waveforms created by Signal Studio software (licenses are required)



The Working with 802.11ac MIMO RnD and DVT Tests (page 61) section focuses on 802.11ac MIMO R&D/DVT Tests related to Rx/Tx PHY Layer characterization. This section shows how IVI-COM Console Applications are used to route backplane triggers on the M9018A PXIe Chassis for the M9381A PXIe VSGs, M9300A PXIe References, and M9391A PXIe VSAs. It then sets the controls for multiple M9381A PXIe VSGs and starts them playing a waveform file. The results of these waveform files are analyzed with Keysight 89600 VSA Software that is used to control M9391A PXIe VSAs while performing transmitter tests for PHY Layer characterization.

The Using Shared LO for Phase-Coherent Signal Generation and Signal Acquisition (page 97) section focuses on creating programs for multichannel operations for M9381A PXIe VSGs and M391A PXIe VSAs using shared local oscillator (LO) for each set of transmitters (M9381A) and receivers (M9391A).

- 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
- Example Program 4: How to Perform Transmitter Tests with 89600 VSA Software,(Playing Waveforms on M9381A PXIe VSGs Using External Trigger)
- Example Program 5: How to Perform Multi-Channel Synchronous Modulated Signal Generation Using Shared LO
- Example Program 6: How to Perform Multi-Channel IQ Acquisition Using Shared LO

Related Websites

- Keysight Technologies PXI and AXIe Modular Products
  - M9391A PXIe Vector Signal Analyzer
  - M9381A PXIe Vector Signal Generator
- 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 M9391A PXIe Vector Signal Analyzer and M9381A PXIe Vector Signal Generator Programming 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**	**Default Location on 64-bit Windows system**	**Format**
Startup Guide	Includes procedures to help you to unpack,	For M9381: *C:/Program Files (x86) /Agilent/M938x/Help\M9391_and_ M9381_StartupGuide.pdf* **For M9391A:***C:\Program Files (x86)*	*PDF*

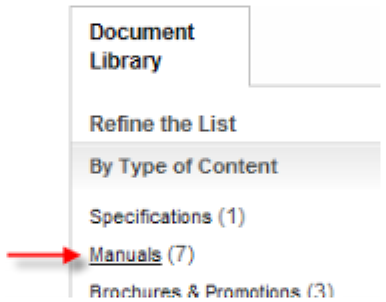
Document	**Description**	**Default Location on 64-bit Windows system**	**Format**
	inspect, install (software and hardware), perform instrument connections, verify operability, and troubleshoot your product. Also includes an annotated block diagram.	*\Agilent\M9391\Help\M9391_and_ M9381_StartupGuide.pdf*	
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.	For M9381: *C:/Program Files (x86) /Agilent/M938x/Help\AgM938x.chm* **For M9391A:***C:\Program Files (x86) \Agilent\M9391\Help\AgM9391.chm*	*CHM (Microsoft Help Format)*
Data Sheet	In addition to a detailed product introduction, the data sheet supplies full product specifications.	For M9381: *C:/Program Files (x86) /Agilent/M938x/Help\M9381_ DataSheet_5991-0279EN.pdf* **For M9391A:***C:\Program Files (x86) \Agilent\M9391\Help\M9391_ DataSheet_5991-2603EN.pdf*	*PDF*
LabVIEW Driver	Provides detailed	For M9381: *C:/Program Files (x86) /Agilent/M938x/Help\AgM938x_*	*CHM (Microsoft*

Document	Description	Default Location on 64-bit Windows system	Format
Reference	documentation of the LabVIEW G Driver API functions.	*LabVIEW_Help.chm* For M9391A:*C:\Program Files (x86) \Agilent\M9391\Help\AgM9391_ LabVIEW_Help.chm*	*Help Format)*
SCPI Reference	Describes the SCPI commands supported by the M9381A PXIe Vector Signal Generator.	*C:/Program Files (x86) /Agilent/M938x/Help\M938x_SCPI_ Reference.chm*	*CHM (Microsoft Help Format)*
Software Release Notes	Includes recent changes, enhancements, and bug fixesin the current release.	For M9381: *C:/Program Files (x86) /Agilent/M938x/Help\M938x_ SoftwareReleaseNotes.pdf* For M9391A:*C:\Program Files (x86) \Agilent\M9391\Help\M9391_ SoftwareReleaseNotes.pdf*	*PDF*

NOTE Alternatively, you can find these documents under:

- *Start > All Programs > Keysight > M938x.*
- *Start* > All Programs > Keysight > M9391.

- The documentation listed above is also available on the product CD.
- To understand the available user documentation in context to your workflow, Documentation Map (page 16).
- To find the very latest versions of the user documentation, go to the product web site (www.keysight.com/find/M9381A or www.keysight.com/find/M9391A) and download the files from the Manuals support page (go to **Document Library > Manuals**):

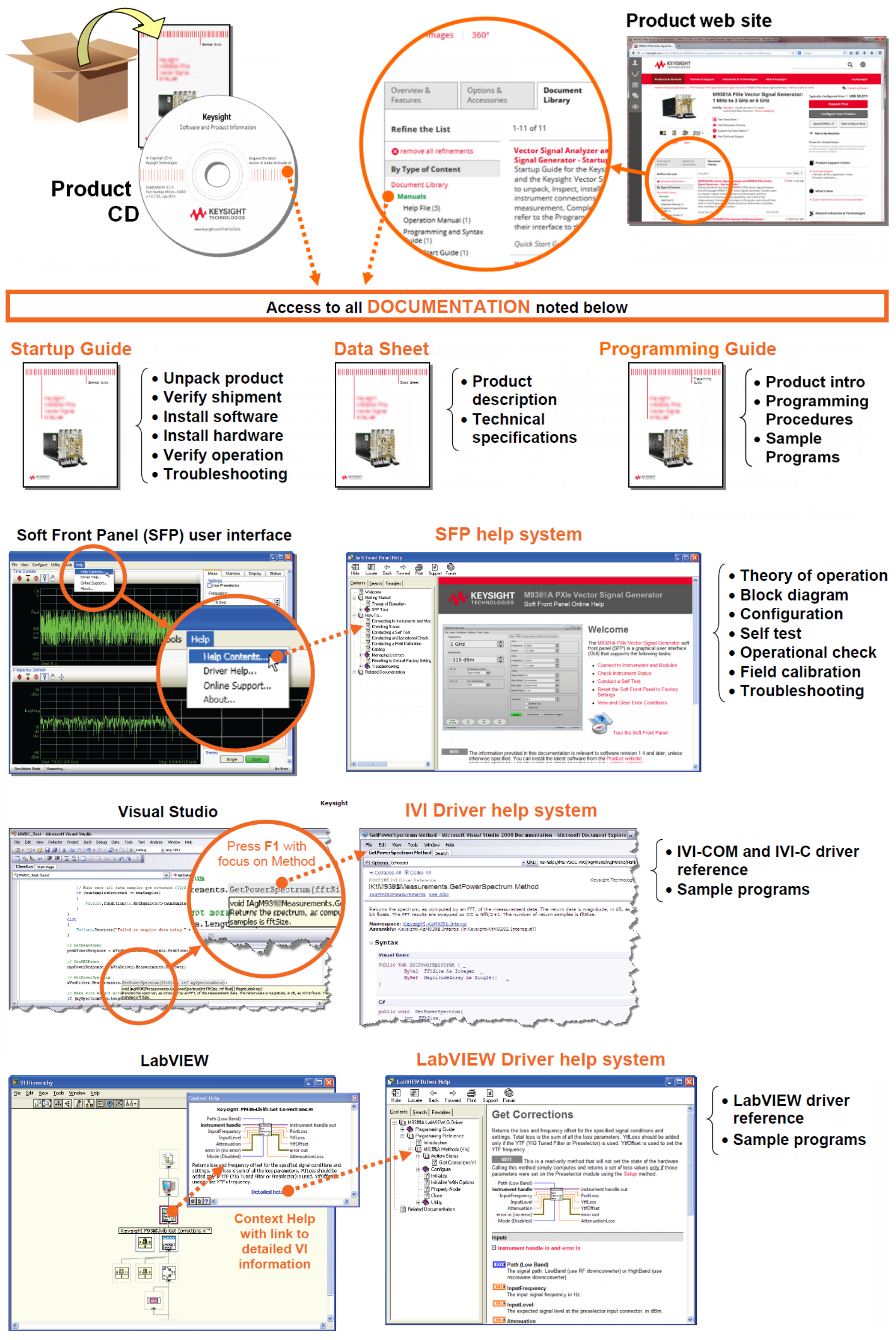


Overall Process Flow

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).

Documentation Map



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, defined in the *Keysight M9391A PXIe VSA and M9381A PXIe VSG Startup Guide, M9300-90090*, but repeated here must be completed before programmatically controlling the M9391A PXIe VSA and M9381A PXIe VSG hardware with their IVI drivers.

    b. Install Agilent/Keysight IO Libraries Suite (IOLS), Version 16.3.16603.3 or newer;this installation includes Agilent/Keysight Connection Expert.

    c. (Required for MIMO) Install Agilent/Keysight 89600 Vector Signal Analyzer Software, Version 16.2 or newer.

    d. Install the M9391A PXIe VSA driver software, Version 1.1.228.0 or newer.

    e. Install the M938xA PXIe VSG driver software, Version 1.3.105.0 or newer.

    f. Install the M9018A PXIe Chassis driver software, Version 1.3.443.1 or newer.

    Driver software includes all IVI-COM, IVI-C, and LabVIEW G Drivers along with Soft Front Panel (SFP) programs 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/89600 (Required for MIMO) > Select **Technical Support** > Select the **Drivers, Firmware & Software** tab > Download the **Instrument Driver** that corresponds to "89600 VSA software".

- http://www.keysight.com/find/m9391a > Select **Technical Support** > Select the **Drivers, Firmware & Software** tab > Download the **Instrument Driver**.
- http://www.keysight.com/find/m9381a > Select **Technical Support** > Select the **Drivers, Firmware & Software** tab > Download the **Instrument Driver**.
- http://www.keysight.com/find/m9018a > 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 hardware modules and make cable connections.
5. Verify operation of the modules (or the system that the modules create).

NOTE **Before programming or making measurements, conduct a Self-Test on each M9391A PXIe VSA and each M9381A PXIe VSG to make sure there are no problems with the modules, cabling, or backplane trigger mapping.**

NOTE **Running Self-Test will fail if the modules that form an M9381A PXIe VSG or an M9391A PXIe VSA spans across slot 6 or slot 12 of the M9018A PXIe Chassis; if they do span across slot 6 or slot 12, the backplane triggers and bus segments must be routed properly. For details, see** Step 6 - Route Backplane Triggers and Bus Segments on the M9018A PXIe Chassis (page 70)**.**

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

APIs for the M9391A PXIe VSA and M938xA PXIe VSG

The following IVI driver terminology may be used when describing the Application Programming Interfaces (APIs) for the M9391A PXIe VSA and M938xA PXIe VSG.

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 M9391A PXIe VSA and the M9381A PXIe VSG 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

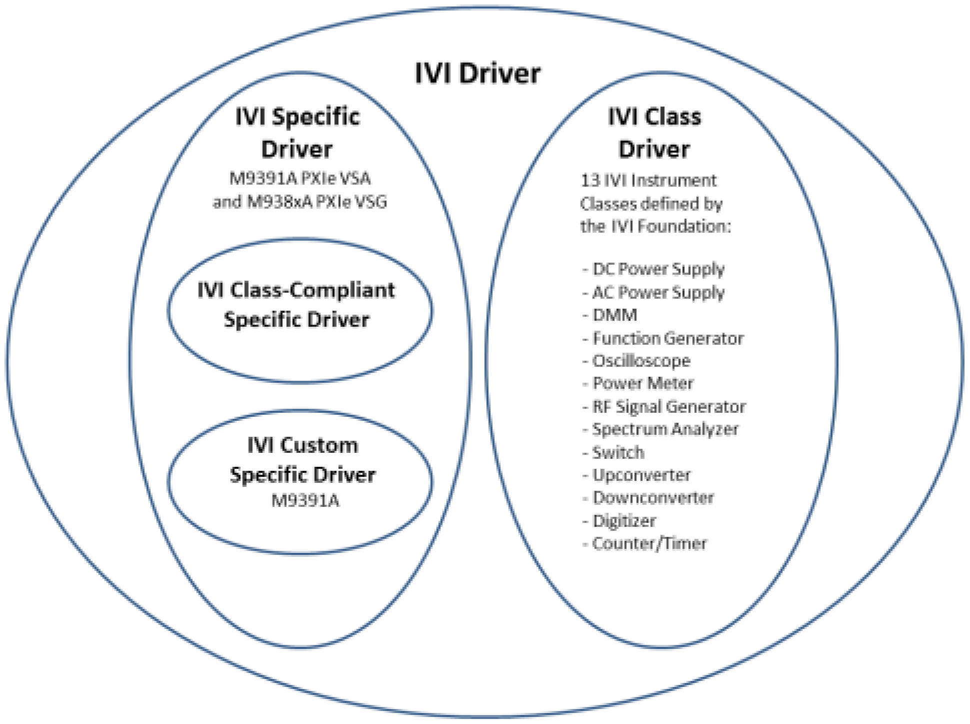
IVI Compliant or IVI Class Compliant

The M9391A PXIe VSA and M9381A PXIe VSG are 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 M9391A PXIe VSA or M938xA PXIe VSG

- 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

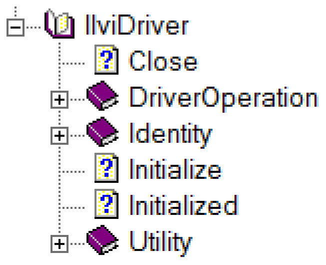
This release is not binary compatible with prior releases of the IVI-C driver. Programs using the C/C++ IVI-C driver must be recompiled for this version of the driver. Similarly, programs compiled with this version of the driver will not be compatible with older versions of the IVI-C driver. This incompatibility is due to renumbering of attribute constants defined in the AgM9391.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 M9391A PXIe VSA and M9381A PXIe VSG do not belong to one of the 13 IVI Classes, there is noClass-Compliant Hierarchy in their IVI Driver.
  - Instrument-Specific Hierarchy
    - The M9391A PXIe VSA's instrument-specific hierarchy has IAgM9391 at the root (where AgM9391 is the driver name).
      - IAgM9391 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 M938xA PXIe VSG's instrument-specific hierarchy has IAgM938x at the root (where AgM938x is the driver name).
      - IAgM938x 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 M9391A PXIe VSA or M938xA PXIe VSG must still include these IVI inherent capabilities.



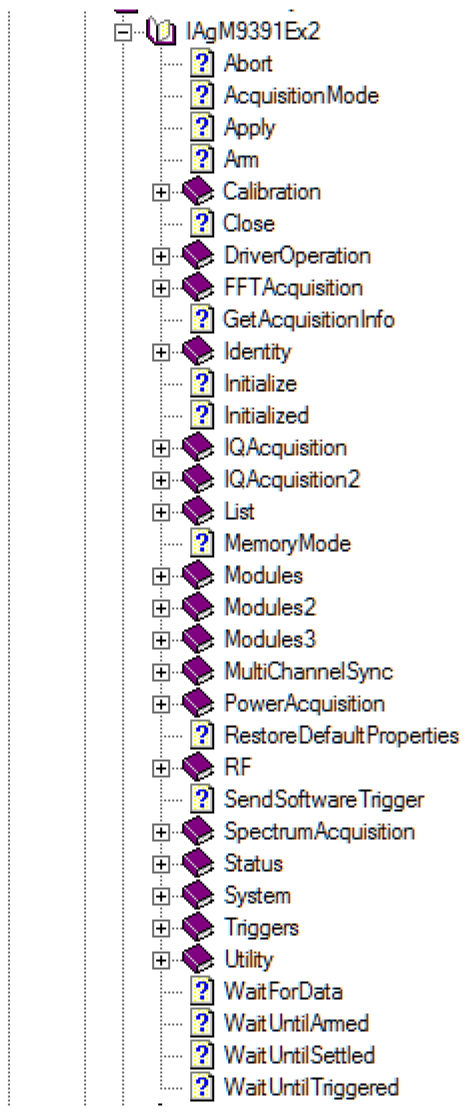
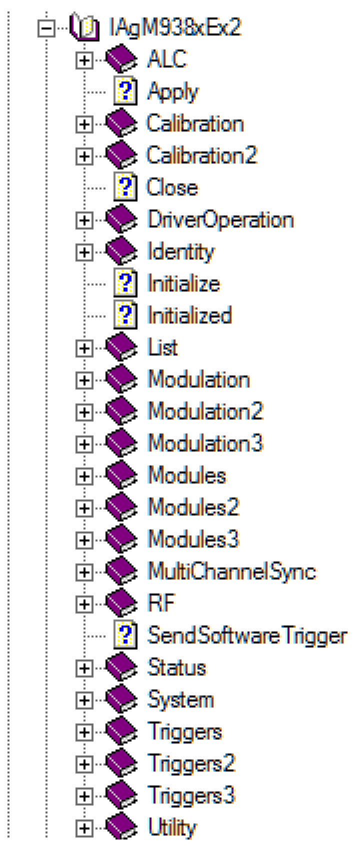
| | IiviDriver

Close
DriverOperation
Identity
Initialize
Initialized
Utility |

Instrument-Specific Hierarchies for the M9391A and M938xA

The following table lists the instrument-specific hierarchy interfaces for M9391A PXIe VSA and M938xA PXIe VSG.

Keysight M9391A PXIe VSA Instrument-Specific Hierarchy	Keysight M938xA PXIe VSG Instrument-Specific Hierarchy
AgM9391 is the driver name	AgM938x is the driver name
IAgM9391Ex is the root interface	IAgM938xEx is the root interface

Keysight M9391A PXIe VSA Instrument-Specific Hierarchy	Keysight M938xA PXIe VSG Instrument-Specific Hierarchy
	

```
IAgM9391Ex2
    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
```

```
IAgM938xEx2
  + ALC
    Apply
  + Calibration
  + Calibration2
    Close
  + DriverOperation
  + Identity
    Initialize
    Initialized
  + List
  + Modulation
  + Modulation2
  + Modulation3
  + Modules
  + Modules2
  + Modules3
  + MultiChannelSync
  + RF
    SendSoftwareTrigger
  + Status
  + System
  + Triggers
  + Triggers2
  + Triggers3
  + Utility
```

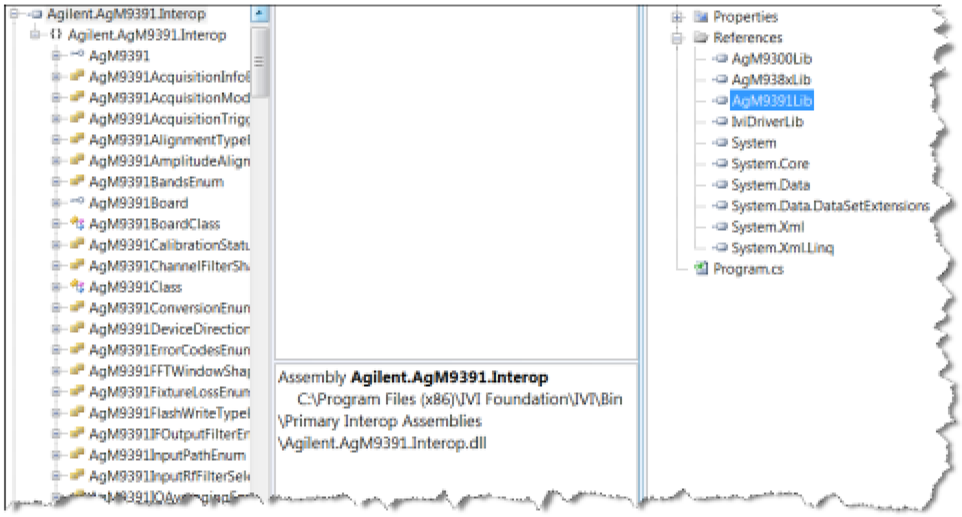
NOTE

All new code being created should use the IAgM9391Ex and IAgM938xEx extended interfaces in place of the IAgM9391 and IAgM938x interfaces. New functionalities have been added to the IAgM9391Ex and IAgM938xEx extended interfaces. These new functionalities were not available in the original IAgM9391 and IAgM938x interfaces, and have been left unchanged to support previously written code; this helps support backward code compatibility.

When Using Visual Studio

– To view the interfaces available in the M9381A PXIe VSG, right-click AgM938xLib library file, in the References folder, from the Solution Explorer window and select View in Object Browser.

- To view interfaces available in the M9391A PXIe VSA, right-click AgM9391Lib library file, in the References folder, from the Solution Explorer window and select View in Object Browser.



Assembly **Agilent.AgM9391.Interop**
C:\Program Files (x86)\IVI Foundation\IVI\Bin
\Primary Interop Assemblies
\Agilent.AgM9391.Interop.dll

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 "VsaVsgProperties" 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.**

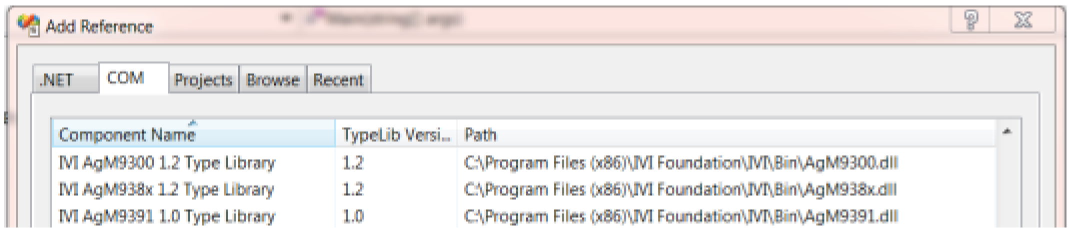
3. Select **Project** and click **Add Reference**. The Add Reference dialog appears. For this step, Solution Explorer must be visible (**View > Solution Explorer**) and the "Program.cs" editor window must be visible; select the **Program.cs** tab to bring it to the front view.

Step 2 - Add References

In order to access the M9391A PXIe VSA and M9381A PXIe VSG 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.

3. 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".)



Component Name	TypeLib Versi...	Path
IVI AgM9300 1.2 Type Library	1.2	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM9300.dll
IVI AgM938x 1.2 Type Library	1.2	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM938x.dll
IVI AgM9391 1.0 Type Library	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM9391.dll

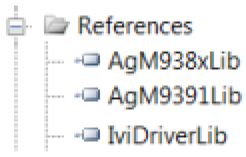
NOTE **If you have not installed the IVI driver for the M9391A PXIe VSA and M9381A PXIe VSG products (as listed in the previous section titled "Before Programming, Install Hardware, Software, and Software Licenses"), their IVI drivers will not appear in this list.**

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".

4. Scroll to IVI section and, using **Shift-Ctrl**, select the following type libraries then select **OK**.
IVI AgM938x 1.2 Type Library
IVI AgM9391 1.0 Type Library

NOTE **When any of the references for the AgM9391A or AgM938x 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).**

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



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

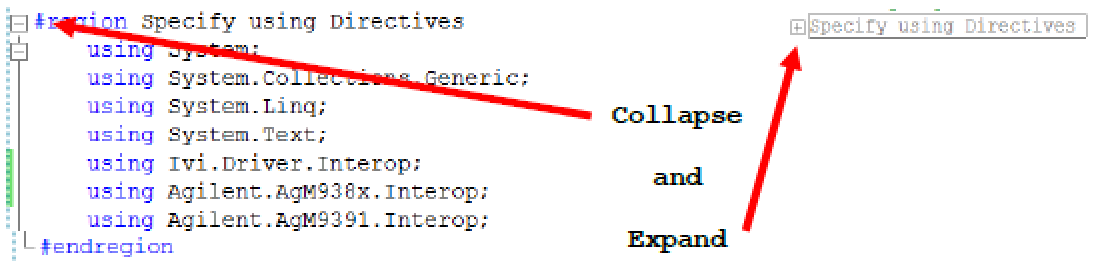
All data types (interfaces and enums) are contained within namespaces. (A namespace is a hierarchical naming scheme for grouping types into logical categories of related functionality. Design tools, such as Visual Studio, can use namespaces which makes it easier to browse and reference types in your 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 Agilent.AgM938x.Interop;
using Agilent.AgM9391.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 System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Ivi.Driver.Interop;
    using Agilent.AgM938x.Interop;
    using Agilent.AgM9391.Interop;
#endregion
```

```
Specify using Directives
```

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 M9391A PXIe VSA and M9381A PXIe VSG are both considered NoClass modules(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, the M9391A PXIe VSA and M938xA PXIe VSG IVI-COM drivers use direct instantiation. Because direct instantiation is used, their IVI-COM drivers may not be interchangeable with other VSA and VSG modules.

To Create Driver Instances

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

```
IAgM9391 VsaDriver = new AgM9391(); IAgM9381 VsgDriver = new AgM9381();
```

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.

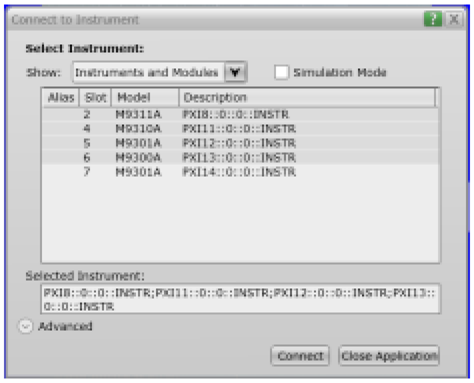
The `Initialize()` method has a number of options that can be defined. In this example, we prepare the `Initialize()` method by defining only a few of the parameters, then we call the `Initialize()` method with these parameters:

Resource Names

- If you are using Simulate Mode, you can set the Resource Name address string to:
  ```
  string VsaResourceName = "%";
  string VsgResourceName = "%";
  ```
- 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), or you can use the Keysight product's Soft Front Panel (SFP) to get the physical Resource Name string.

  Using the M938xA Soft Front Panel, you might get the following Resource

Name address string.



Select Instrument:

Show: Instruments and Modules

Alias	Slot	Model	Description
	2	M9311A	PXI8::0::0::INSTR
	4	M9310A	PXI11::0::0::INSTR
	5	M9301A	PXI12::0::0::INSTR
	6	M9300A	PXI13::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR

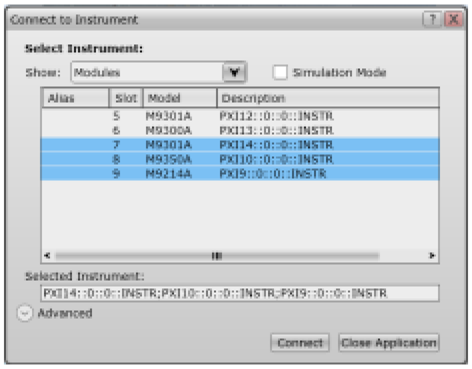
Selected Instrument:

PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::INSTR

ModuleName	M9311A PXIe Modulator	M9310A PXIe Source Output	M9301A PXIe Synthesizer	M9300A PXIe Reference
Slot Number	2	4	5	6
VISA Address	PXI8::0::0::INSTR;	PXI11::0::0::INSTR;	PXI12::0::0::INSTR;	PXI13::0::0::INSTR;

```
string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::I
NSTR";
```

Using the M9391A Soft Front Panel, you might get the following Resource Name address string.



Select Instrument:

Show: Modules

Alias	Slot	Model	Description
	5	M9301A	PXI12::0::0::INSTR
	6	M9300A	PXI13::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	8	M9350A	PXI10::0::0::INSTR
	9	M9214A	PXI9::0::0::INSTR

Selected Instrument:

PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR

ModuleName	M9301A PXIe Synthesizer	M9350A PXIe Downconverter	M9214A PXIe IF Digitizer
Slot Number	7	8	9

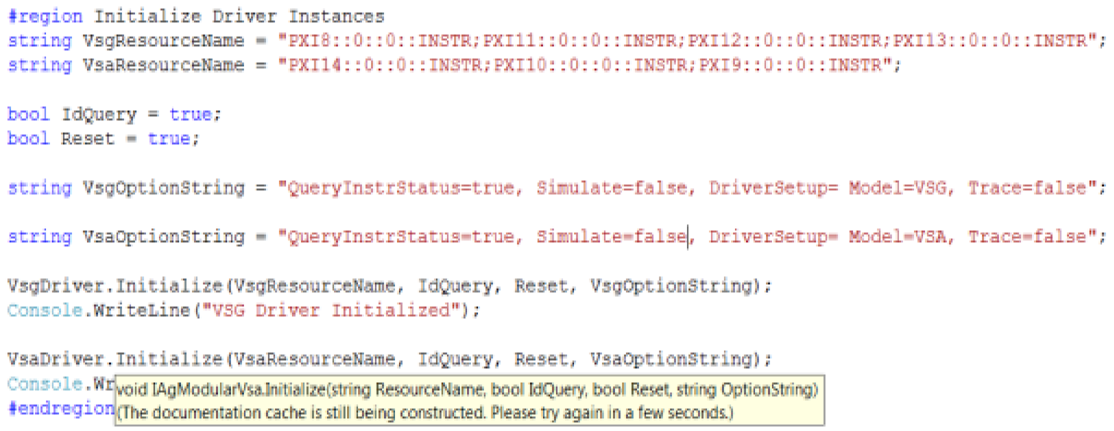
ModuleName	M9301A PXIe Synthesizer	M9350A PXIe Downconverter	M9214A PXIe IF Digitizer
VISA Address	PXI14::0::0::INSTR;	PXI10::0::0::INSTR;	PXI9::0::0::INSTR;

```
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::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.

```
// The M9300A PXIe Reference should be included as one of the modules in
// either the M9381A PXIe VSG configuration of modules or theor the M9391A //
PXIe VSA configuration of modules).
// If the M9300A PXIe Reference is only included in one configuration,
// that configuration should be initialized first.
// See "Understanding M9300A Frequency Reference Sharing".

string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::INSTR";
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;

bool IdQuery = true;
bool Reset = true;

string VsgOptionString = "QueryInstrStatus=true, Simulate=false, DriverSetup=
Model=VSG, Trace=false";
string VsaOptionString = "QueryInstrStatus=true, Simulate=false, DriverSetup=
Model=VSA, Trace=false";

// Initialize the drivers
VsgDriver.Initialize(VsgResourceName, IdQuery, Reset, VsgOptionString);
Console.WriteLine("VSG Driver Initialized");

VsaDriver.Initialize(VsaResourceName, IdQuery, Reset, VsaOptionString);
Console.WriteLine("VSA Driver Initialized");
```



```
#region Initialize Driver Instances
string VsgResourceName = "PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::INSTR";
string VsaResourceName = "PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR";

bool IdQuery = true;
bool Reset = true;

string VsgOptionString = "QueryInstrStatus=true, Simulate=false, DriverSetup= Model=VSG, Trace=false";

string VsaOptionString = "QueryInstrStatus=true, Simulate=false, DriverSetup= Model=VSA, Trace=false";

VsgDriver.Initialize(VsgResourceName, IdQuery, Reset, VsgOptionString);
Console.WriteLine("VSG Driver Initialized");

VsaDriver.Initialize(VsaResourceName, IdQuery, Reset, VsaOptionString);
Console.Wr
#endregion
```

void IAgModularVsa.Initialize(string ResourceName, bool IdQuery, bool Reset, string OptionString)
(The documentation cache is still being constructed. Please try again in a few seconds.)

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;PXI14::0::0::INSTR;PXI15::0::0::INSTR;PXI16::0::0::INSTR";	VsgResourceName or VsaResourceName – 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

Property Type and Example Value	Description of Property
	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, Simulate=true,	OptionString - Setup the following initialization options: – 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

Property Type and Example Value	Description of Property
	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= Trace=false";	– 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.) – Model=VSG or Model=VSA (Instrument model to use during

Property Type and Example Value	Description of Property
	simulation.) – Trace=false (If false, an output trace log of all driver calls is not saved in an XML file.)

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

AgM938x IVI Driver Reference
Start > All Programs > Keysight Instrument Drivers > IVI-COM-C Drivers > AgM938x Source > AgM938x IVI Driver Help

AgM9391 IVI Driver Reference
Start > All Programs > Keysight Instrument Drivers > IVI-COM-C Drivers > AgM9391A VSA > AgM9391 IVI Driver Help

M9300A Reference Sharing

The M9300A PXIe Reference can be shared by up to five configurations of modules that can be made up of the M9391A PXIe VSA or the M9381A PXIe VSG or both. The M9300A PXIe Reference must be included as one of the modules in at least one of these configurations. The configuration of modules that is initialized first must include the M9300A PXIe Reference 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.

Example: M9300A PXIe Reference with M9381A PXIe VSG

The **M9381A PXIe VSG should be initialized first** before initializing the VSA if:

- M9381A PXIe VSG configuration of modules includes:
  - M9311A PXIe Modulator
  - M9310A PXIe Source Output
  - M9301A PXIe Synthesizer
  - **M9300A PXIe Reference** // Note that the M9300A PXIe Reference is part of the M9381A PXIe VSG configuration of modules.

```
string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0
::0::INSTR";
```

- M9391A PXIe VSA configuration of modules includes:
    - M9301A PXIe Synthesizer
    - M9350A PXIe Downconverter
    - M9214A PXIe IF Digitizer

```
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR";
```

Example: M9300A PXIe Reference with M9391A PXIe VSA

The **M9391A PXIe VSA should be initialized first** before initializing the M9381A PXIe VSG if:

- M9381A PXIe VSG configuration of modules includes:
    - M9311A PXIe Modulator
    - M9310A PXIe Source Output
    - M9301A PXIe Synthesizer

```
string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR";
```

- M9391A PXIe VSA configuration of modules includes:
    - M9300A PXIe Reference* // Note that the M9300A PXIe Reference is part of the M9391A PXIe VSA configuration of modules.
    - M9301A PXIe Synthesizer
    - M9350A PXIe Downconverter
    - M9214A PXIe IF Digitizer

```
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI13::0
::0::INSTR;
```

Example: M9300A PXIe Reference Shared With Both Modules

The **M9391A PXIe VSA or the M9381A PXIe VSG can be initialized first** since the M9300A PXIe Reference is included in both configurations of modules:

- M9381A PXIe VSG configuration of modules includes:
    - M9311A PXIe Modulator
    - M9310A PXIe Source Output
    - M9301A PXIe Synthesizer
    - M9300A PXIe Reference* // Note that the M9300A PXIe Reference is part of the M9381A PXIe VSG configuration of modules.

```
string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR";PXI13::
0::0::INSTR;
```

- M9391A PXIe VSA configuration of modules includes:
  - M9300A PXIe Reference* // Note that the M9300A PXIe Reference is part of the M9391A PXIe VSA configuration of modules.
  - M9301A PXIe Synthesizer
  - M9350A PXIe Downconverter
  - M9214A PXIe IF Digitizer

```
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI13::0
::0::INSTR;
```

Step 6 - Write the Program Steps

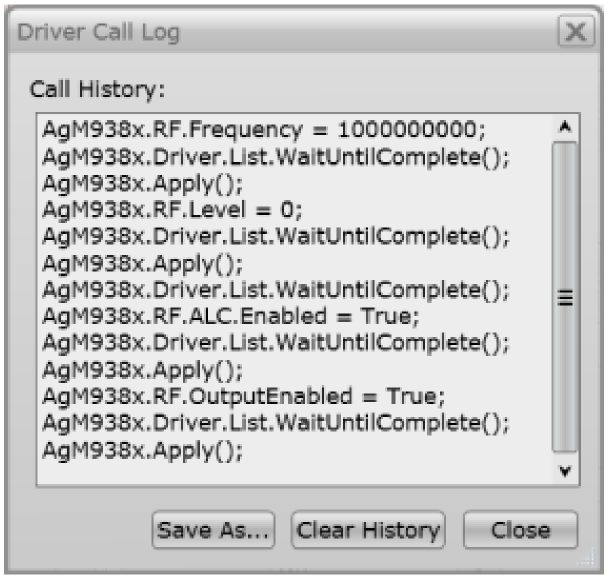
At this point, you can add program steps that use the driver instances to perform tasks.

Using the Soft Front Panel to Write Program Commands

In this example, open the Soft Front Panel for the M938xA PXIe VSG and perform the following steps:

1. Set the output frequency to 1 GHz.
2. Set the output level to 0 dBm.
3. Enable the ALC.
4. Enable the RF Output.

The illustration below shows the Driver Call Log created by the steps above.



```
AgM938x.RF.Frequency = 1000000000;
AgM938x.Driver.List.WaitUntilComplete();
AgM938x.Apply();
AgM938x.RF.Level = 0;
AgM938x.Driver.List.WaitUntilComplete();
AgM938x.Apply();
AgM938x.Driver.List.WaitUntilComplete();
AgM938x.RF.ALC.Enabled = True;
AgM938x.Driver.List.WaitUntilComplete();
AgM938x.Apply();
AgM938x.RF.OutputEnabled = True;
AgM938x.Driver.List.WaitUntilComplete();
AgM938x.Apply();
```

Below is the corresponding code in C#:

  - AgM938x is the driver name used by the SFP.
  - VsgDriver is the instance of the driver that is used in this example. This instance would have been created in, "Step 4 – Create Instances of the M9381A PXIe VSG and M9391A PXIe VSA".

```
IAgM938x VsgDriver = new AgM938x();
```

```
// Set the output frequency to 1 GHz
VsgDriver.RF.Frequency = 1000000000;
// Set the output level to 0 dBm
VsgDriver.RF.Level = 0;
// Enables the ALC
VsgDriver.ALC.Enabled = true;
// Enables the RF Output
VsgDriver.RF.OutputEnabled = true;
// Waits until the list is finished or the specified time passes
bool retval = VsgDriver.List.WaitUntilComplete();

//...or you could use the following:

// Waits 100 ms until output is settled before producing signal
bool retval = VsgDriver.RF.WaitUntilSettled(100);
```

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

  if(VsgDriver != null && VsgDriver.Initialized)
   {
       // Close the VSG driver
```

```
            VsgDriver.Close();
            Console.WriteLine("VSG Driver Closed");
        }
    }
```

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

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

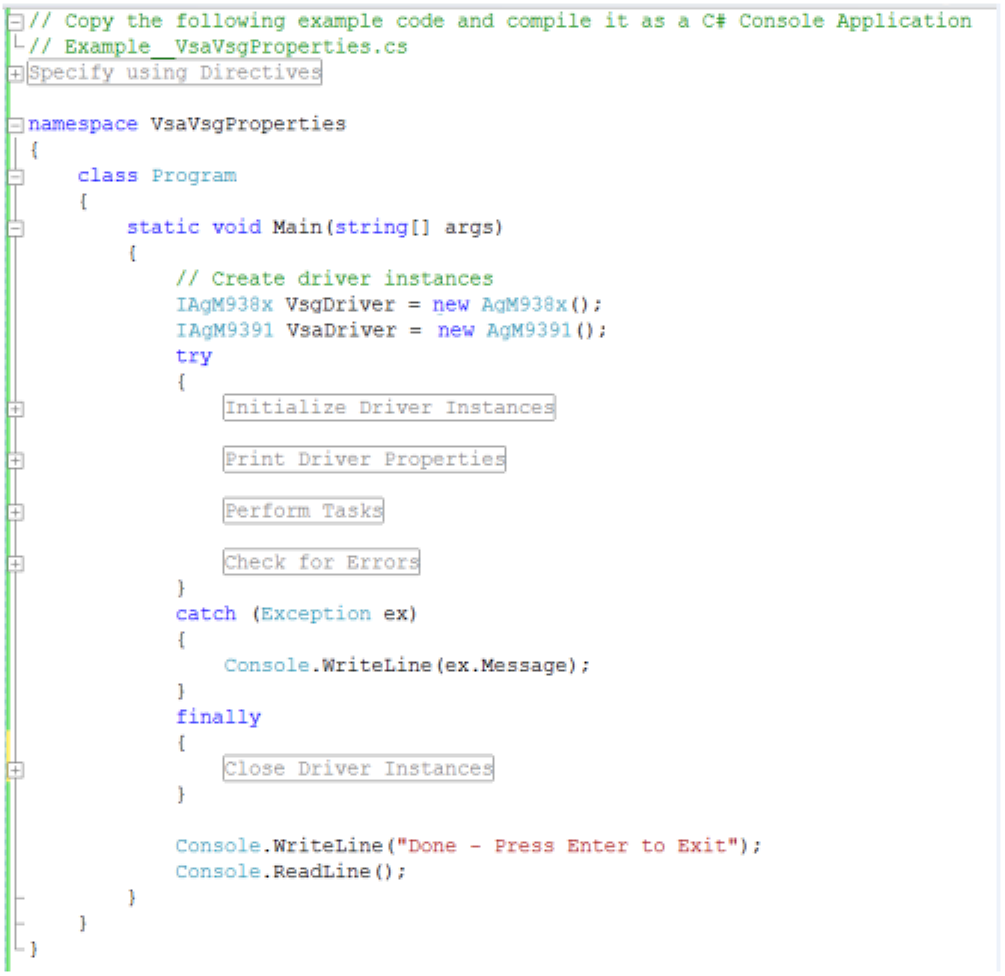
1. Open the solution file **SolutionNameThatYouUsed.sln** in Visual Studio 2010.
2. Set the appropriate platform target for your project.
   - In many cases, the default platform target (Any CPU) is appropriate.
   - However, if you are using a 64-bit PC (such as Windows 7) to build a .NET application that uses a 32-bit IVI-COM driver, you may need to specify your project's platform target as **x86.**
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 programs may be found by selecting: **C:\Program Files (x86) \Agilent\M9391\Help\Examples**

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.

Example programs may be found in **C:\Program Files (x86) \Agilent\M9391\Help\Examples**



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

namespace VsaVsgProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            IAgM938x VsgDriver = new AgM938x();
            IAgM9391 VsaDriver =  new AgM9391();
            try
            {
                Initialize Driver Instances

                Print Driver Properties

                Perform Tasks

                Check for Errors
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                Close Driver Instances
            }

            Console.WriteLine("Done - 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__VsaVsgProperties.cs
#region Specify using Directives
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ivi.Driver.Interop;
using Agilent.AgM938x.Interop;
using Agilent.AgM9391.Interop;
#endregion

namespace VsaVsgProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            IAgM938x VsgDriver = new AgM938x();
            IAgM9391 VsaDriver = new AgM9391();
      try
      {
        #region Initialize Driver Instances
        string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::INSTR";
        string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR";

        bool IdQuery = true;
        bool Reset   = true;

        string VsgOptionString = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=VSG, Trace=false";

        string VsaOptionString = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=VSA, Trace=false";

        VsgDriver.Initialize(VsgResourceName, IdQuery, Reset,
VsgOptionString);
        Console.WriteLine("VSG Driver Initialized");

        VsaDriver.Initialize(VsaResourceName, IdQuery, Reset,
VsaOptionString);
        Console.WriteLine("VSA Driver Initialized\n\n");
        #endregion

        #region Print Driver Properties
        // Print IviDriverIdentity properties for the PXIe VSG
        Console.WriteLine("Identifier:  {0}", VsgDriver.Identity.Identifier);
        Console.WriteLine("Revision:    {0}", VsgDriver.Identity.Revision);
        Console.WriteLine("Vendor:      {0}", VsgDriver.Identity.Vendor);
        Console.WriteLine("Description: {0}",
VsgDriver.Identity.Description);
        Console.WriteLine("Model:       {0}",
VsgDriver.Identity.InstrumentModel);
        Console.WriteLine("FirmwareRev: {0}",
VsgDriver.Identity.InstrumentFirmwareRevision);
        Console.WriteLine("Simulate:    {0}\n",
VsgDriver.DriverOperation.Simulate);

        // Print IviDriverIdentity properties for the PXIe VSA
        Console.WriteLine("Identifier:  {0}", VsaDriver.Identity.Identifier);
        Console.WriteLine("Revision:    {0}", VsaDriver.Identity.Revision);
        Console.WriteLine("Vendor:      {0}", VsaDriver.Identity.Vendor);
        Console.WriteLine("Description: {0}",
VsaDriver.Identity.Description);
        Console.WriteLine("Model:       {0}",
VsaDriver.Identity.InstrumentModel);
        Console.WriteLine("FirmwareRev: {0}",
VsaDriver.Identity.InstrumentFirmwareRevision);
        Console.WriteLine("Simulate:    {0}\n",
VsaDriver.DriverOperation.Simulate);
        #endregion

        #region Perform Tasks
        // TO DO: Exercise driver methods and properties.
        // Put your code here to perform tasks with PXIe VSG and PXIe VSA.
        #endregion

        #region Check for Errors
        // Check VSG instrument for errors
        int VsgErrorNum = -1;
        string VsgErrorMsg = null;
        while (VsgErrorNum != 0)
        {
            VsgDriver.Utility.ErrorQuery(ref VsgErrorNum, ref VsgErrorMsg);
            Console.WriteLine("VSG ErrorQuery: {0}, {1}\n", VsgErrorNum,
VsgErrorMsg);
        }

        // Check VSA instrument for errors
        int VsaErrorNum = -1;
        string VsaErrorMsg = null;
        while (VsaErrorNum != 0)
        {
            VsaDriver.Utility.ErrorQuery(ref VsaErrorNum, ref VsaErrorMsg);
            Console.WriteLine("VSA ErrorQuery: {0}, {1}\n", VsaErrorNum,
VsaErrorMsg);
        }
        #endregion
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (VsgDriver != null && VsgDriver.Initialized)
        {
          // Close the driver
          VsgDriver.Close();
          Console.WriteLine("VSG Driver Closed");
        }
```

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

            Console.WriteLine("Done - 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

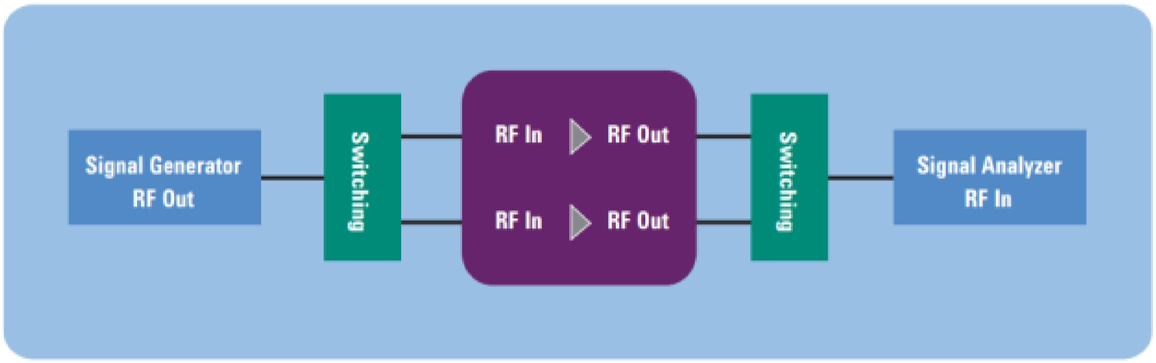
The following are the test challenges faced by Power Amplifier Testing:

- 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 M9381A PXIe VSG and M9391A PXIe VSA 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 M9381A PXIe VSG 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 M9381A PXIe VSG to get the correct DUT output power can be a major contributor to the test time and the overall cost of test.

The M9381A PXIe VSG 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 M9381A PXIe VSG 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 M9381A PXIe VSG and the input of the DUT, which requires higher output levels from the M9381A PXIe VSG. Since the tests are performed with a modulated signal, the M9381A PXIe VSG must also have adequate modulation performance at the higher power levels.

Performing a Channel Power Measurement, Using Immediate Trigger

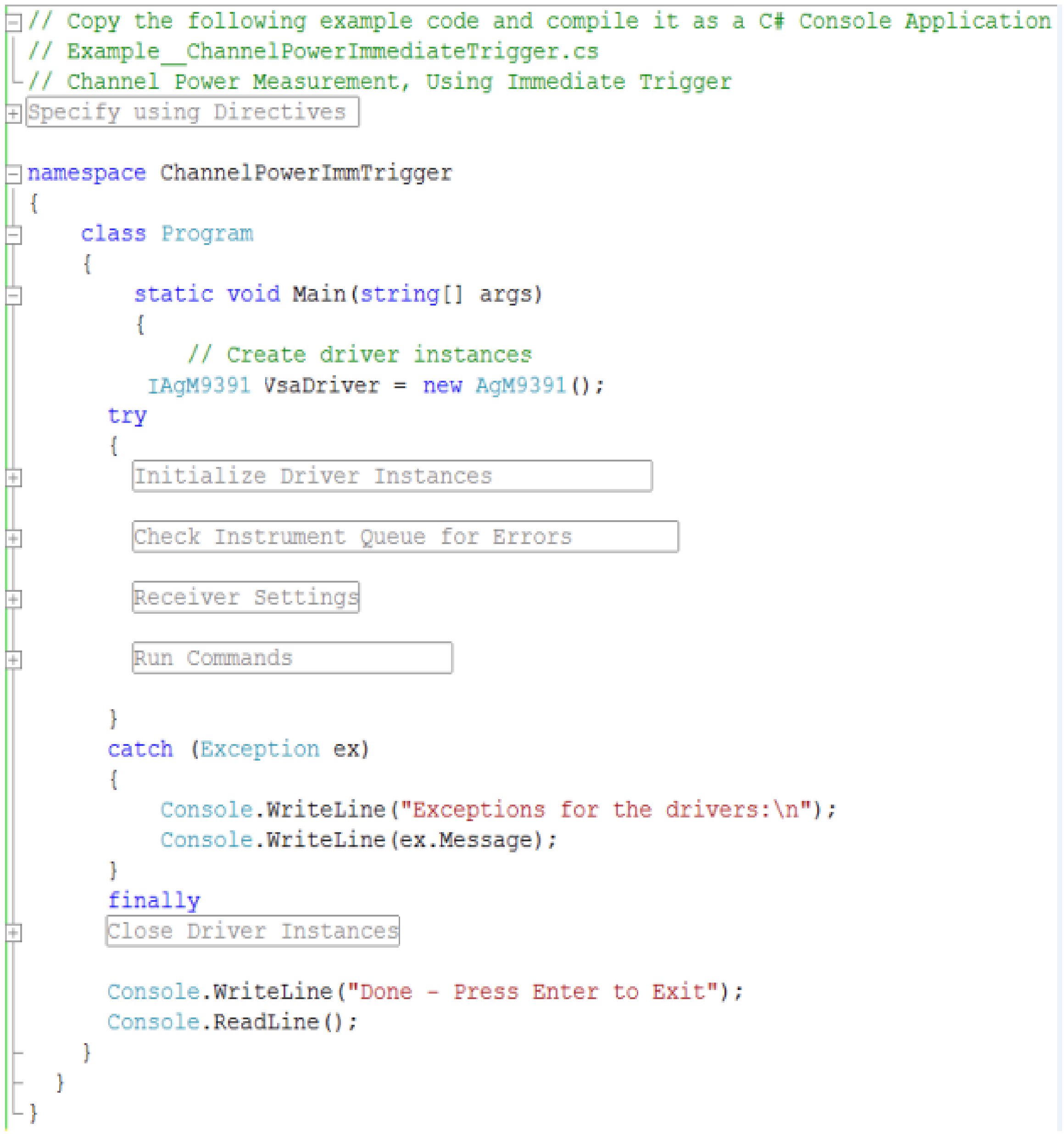
Standard	Sample Rate	Channel Filter Type	Channel Filter Parameter	Channel Filter Bandwidth	Channel Offsets
WCDMA	5 MHz	RRC	0.22	3.84 MHz	5, 10 MHz
LTE 10 MHz FDD	11.25 MHz	Rectangular	N/A	9 MHz	10, 20 MHz
LTE 10 MHz TDD	11.25 MHz	Rectangular	N/A	9 MHz	10, 20 MHz
1xEV-DO	2 MHz	RRC	0.22	1.23 MHz	1.25, 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 RF and Power Acquisition commands to the M9391A PXIe VSA driver and Apply changes to hardware,
2. Check the instrument queue for errors.
3. Perform a Channel Power Measurement,
4. Report errors if any occur, and close the drivers.

Example programs may be found at **C:\Program Files (x86)\Agilent\M9391\Help\Examples**



```
// Copy the following example code and compile it as a C# Console Application
// Example__ChannelPowerImmediateTrigger.cs
// Channel Power Measurement, Using Immediate Trigger
Specify using Directives

namespace ChannelPowerImmTrigger
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
         IAgM9391 VsaDriver = new AgM9391();
        try
        {
            Initialize Driver Instances

            Check Instrument Queue for Errors

            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 VSA, Check for Errors

- Send RF Settings to VSA Driver:
  - Frequency
  - Level
  - Peak to Average Ratio
  - Conversion Mode
  - IF Bandwidth
  - Set Acquisition Mode to "Power"

- Send Power Acquisition Setting to VSA Driver:
  - Sample Rate
  - Duration
  - Channel Filter
- Apply Method to Send Changes to Hardware
  - Wait for Hardware to Settle
- Send Arm Method to VSA
- Send Read Power Method to VSA

Close Driver for VSA

Example Program 2 - Channel Power Measurement Using Immediate Trigger

```
// Copy the following example code and compile it as a C# Console Application
// Example__ChannelPowerImmediateTrigger.cs
#region Specify using Directives
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Ivi.Driver.Interop;
    using Agilent.AgM9391.Interop;
#endregion

namespace ChannelPowerImmTrigger
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            VsaDriver = new AgM9391();
      try
      {
        #region Initialize Driver Instances
            string VSAResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI13::0::0::INSTR";

            bool IdQuery = true;
            bool Reset   = true;

            string VSAOptionString = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=M9391A, Trace=false";

            VsaDriver.Initialize(VSAResourceName, IdQuery, Reset,
VSAOptionString);
```

```
            Console.WriteLine("VSA Driver Initialized\n");
        #endregion

        #region Check Instrument Queue for Errors
            // Check VSA instrument for errors
            int VsaErrorNum = -1;
            string VsaErrorMsg = null;
            while (VsaErrorNum != 0)
            {
                VsaDriver.Utility.ErrorQuery(ref VsaErrorNum, ref 
VsaErrorMsg);
                Console.WriteLine("VSA ErrorQuery: {0}, {1}\n", VsaErrorNum, 
VsaErrorMsg);
            }
        #endregion

        #region Receiver Settings
            // Receiver Settings
            double Frequency = 2000000000.0;
            double Level = 5;
            double RmsValue = 5;
            double ChannelTime = 0.0001;
            double MeasureBW = 5000000.0;
            AgM9391ChannelFilterShapeEnum FilterType = 
AgM9391ChannelFilterShapeEnum.AgM9391ChannelFilterShapeRootRaisedCosine;
            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
            VsaDriver.RF.Frequency = Frequency;
            VsaDriver.RF.Power = Level;
            VsaDriver.RF.Conversion = 
AgM9391ConversionEnum.AgM9391VsaConversionAuto;
            VsaDriver.RF.PeakToAverage = RmsValue;
            VsaDriver.RF.IFBandwidth = 40000000.0; // Use IF filter wide 
enough for all adjacent channels
            // Configure the Acquisition
            VsaDriver.AcquisitionMode = 
AgM9391AcquisitionModeEnum.AgM9391AcquisitionModePower;
            VsaDriver.PowerAcquisition.Bandwidth = MeasureBW;  // 5 MHz
            VsaDriver.PowerAcquisition.Duration = ChannelTime; // 100 us
            VsaDriver.PowerAcquisition.ChannelFilter.Configure(FilterType, 
FilterAlpha, FilterBw);
            // Send Changes to hardware
            VsaDriver.Apply();
            VsaDriver.WaitUntilSettled(100);
            string response = "y";
            while (string.Compare(response, "y") == 0) {
                Console.WriteLine("Press Enter to Run Test");
                Console.ReadLine();

                VsaDriver.Arm();
                VsaDriver.PowerAcquisition.ReadPower(0,
AgM9391PowerUnitsEnum.AgM9391PowerUnitsdBm, 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 (VsaDriver != null && VsaDriver.Initialized)
        {
            // Close the driver
            VsaDriver.Close();
            Console.WriteLine("VSA 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 Acquisition interface of the M9391A PXIe VSA.

The M9391A PXIe VSA and the M9381A PXIe VSG offers two modes for adjusting frequency and amplitude:

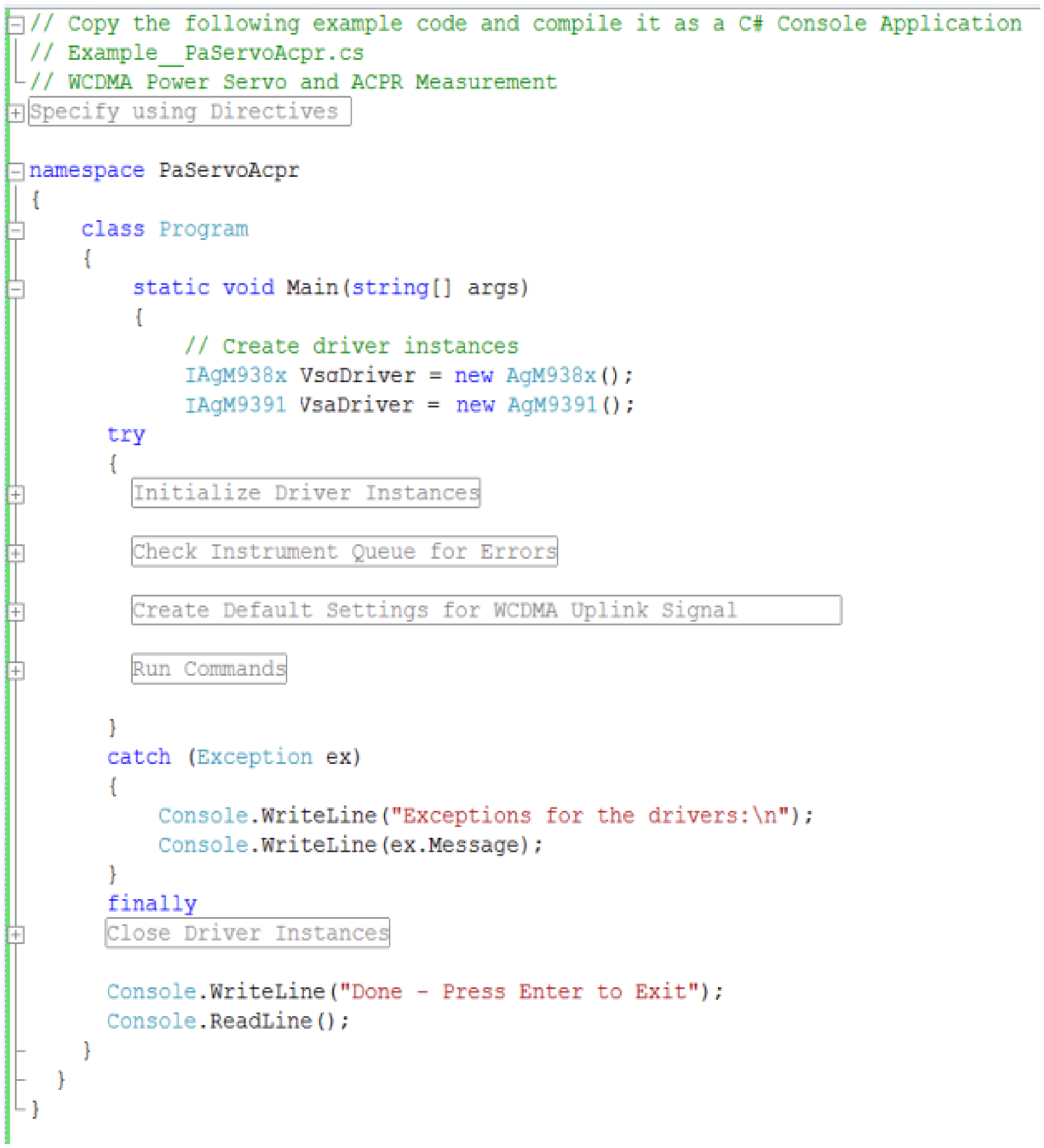
- RF Tuning – allows the M9381A PXIe VSG to be set across the complete operating frequency and amplitude range.
- Baseband Tuning – allows the frequency and amplitude to be adjusted within the IF bandwidth (160 MHz) and over a range of the output level.

Example Program 3 - Code Structure

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

1. Send RF and Modulation commands to the M9381A PXIe VSG driver and Apply changes to hardware,
2. Send RF and Power Acquisition commands to the M9391A PXIe VSA driver and Apply changes to hardware,
3. Run a Servo Loop until it is at the required output power from DUT,
4. Perform an ACPR Measurement for each Adjacent Channel to be measured,
5. Check drivers for errors and report the errors, if any, and close the drivers.

Example programs are available at **C:\Program Files (x86)\Agilent\M9391\Help\Examples**



```
// Copy the following example code and compile it as a C# Console Application
// Example__PaServoAcpr.cs
// WCDMA Power Servo and ACPR Measurement
Specify using Directives

namespace PaServoAcpr
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            IAgM938x VsgDriver = new AgM938x();
            IAgM9391 VsaDriver = new AgM9391();
        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 VSG and VSA and check for errors

- Send RF Settings to VSG Driver:
  - Frequency
  - RF Level to Maximum Needed
  - RF Enable On
  - ALC Enable Off (for baseband power changes)

- Send Modulation Commands to VSG Driver:
  - Load WCDMA Signal Studio File
  - Enable Modulation
  - Play ARB File
  - Set ARB Scale to 0.5
  - Set Baseband Power Offset to -10 dB
- Apply Method to Send Changes to Hardware
  - Wait for Hardware to Settle
- Send RF Settings to VSA Driver:
  - Frequency
  - Level
  - Peak to Average Ratio
  - Conversion Mode
  - IF Bandwidth
  - Set Acquisition Mode to "Power"
- Send Power Acquisition Setting to VSA Driver:
  - Sample Rate
  - Duration
  - Channel Filter
- Apply Method to Send Changes to Hardware
  - Wait for Hardware to Settle

Servo Loop:

- Set Baseband Power Offset on VSG to expected value
- Send Apply Method to VSG
- Send Arm Method to VSA
- Send ReadPower Method to VSA
- Repeat Until at Required Output Power from DUT
- Last Reading is Channel Power Measurement

ACPR Measurement:

- Set Acquisition Duration Property on VSA to Value for Adjacent Channel Measurements
- Set Frequency Offset Property on VSA to Channel Offset Frequency
- Send Apply Method to VSA
- Send Arm Method to VSA
- Send ReadPower Method to VSA
- Repeat for each Adjacent Channel to be Measured

Example Program 3 – WCDMA Power Servo and ACPR Measurement

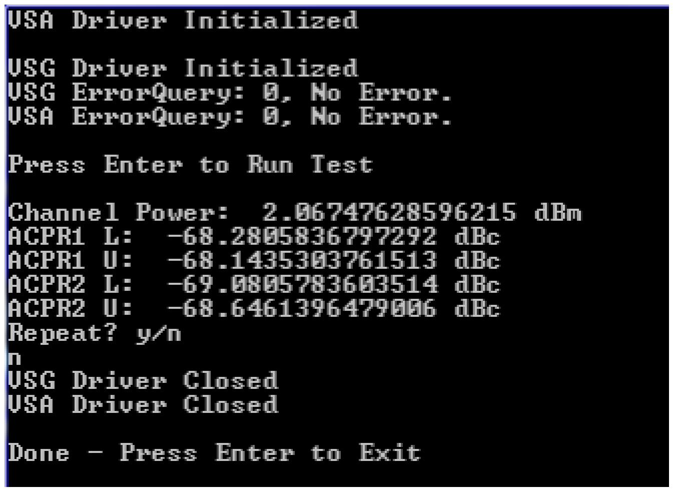
```
// Copy the following example code and compile it as a C# Console Application
// Example__PaServoAcpr.cs
// WCDMA Power Servo and ACPR Measurement
#region Specify using Directives
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Ivi.Driver.Interop;
    using Agilent.AgM938x.Interop;
    using Agilent.AgM9391.Interop;
#endregion

namespace PaServoAcpr
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create driver instances
            IAgM938x VsgDriver = new AgM938x();
            IAgM9391 VsaDriver = new AgM9391();
      try
      {
        #region Initialize Driver Instances
            string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI13::0::0::INSTR";
            string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR";

            bool IdQuery = true;
            bool Reset   = true;

            string VsgOptionString = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=VSG, Trace=false";

            string VsaOptionString = "QueryInstrStatus=true, Simulate=false,
DriverSetup= Model=VSA, Trace=false";

            VsaDriver.Initialize(VsaResourceName, IdQuery, Reset,
VsaOptionString);
            Console.WriteLine("VSA Driver Initialized\n");

            VsgDriver.Initialize(VsgResourceName, IdQuery, Reset,
VsgOptionString);
```

```
                Console.WriteLine("VSG Driver Initialized");

            #endregion

            #region Check Instrument Queue for Errors
                // Check VSG instrument for errors
                int VsgErrorNum = -1;
                string VsgErrorMsg = null;
                while (VsgErrorNum != 0)
                {
                    VsgDriver.Utility.ErrorQuery(ref VsgErrorNum, ref 
VsgErrorMsg);
                    Console.WriteLine("VSG ErrorQuery: {0}, {1}", VsgErrorNum, 
VsgErrorMsg);
                }

                // Check VSA instrument for errors
                int VsaErrorNum = -1;
                string VsaErrorMsg = null;
                while (VsaErrorNum != 0)
                {
                    VsaDriver.Utility.ErrorQuery(ref VsaErrorNum, ref 
VsaErrorMsg);
                    Console.WriteLine("VSA ErrorQuery: {0}, {1}\n", VsaErrorNum, 
VsaErrorMsg);
                }
            #endregion

            #region Create Default Settings for WCDMA Uplink Signal
                // Source Settings
                double Frequency = 1000000000.0;
                double Level = 3;
                // If a Signal Studio waveform file is used, it may require a 
software
                            // license.
                string ExamplesFolder = "C:Program Files (x86)AgilentM938xExample 
Waveforms";
                string WaveformFile = "WCDMA_UL_DPCHH_2DPDCH_1C.wfm";
                string FileName = ExamplesFolder + WaveformFile;
                string ArbRef = "Mod Waveform";

                // Receiver Settings
                double ChannelTime = 0.0001;
                double AdjacentTime = 0.0005;
                double IfBandwidth = 40000000.0;
                double PowerOffset = 0;
                double MeasureBW = 5000000.0;
                AgM9391ChannelFilterShapeEnum FilterType = 
AgM9391ChannelFilterShapeEnum.AgM9391ChannelFilterShapeRootRaisedCosine;
                double FilterAlpha = 0.22;
                double FilterBw = 3840000.0;
                double[] FreqOffset = new double[] {-5000000.0, 5000000.0, -
    10000000.0, 10000000.0};

                double MeasuredPower = 0;
                bool Overload = true;
                double MeasuredChannelPower;
                bool ChannelPowerOverload;
                double[] MeasuredACPR = new double[4];
                double SampleRate = 0;
                double RmsValue = 0;
                double ScaleFactor = 0;
            #endregion

            #region Run Commands
                // These commands are sent to the VSG Driver, "Apply" or
    "PlayArb"
                            // methods send to hardware
                VsgDriver.RF.Frequency = Frequency;
                VsgDriver.RF.Level = Level;
                VsgDriver.RF.OutputEnabled = true;
                VsgDriver.ALC.Enabled = false;
                VsgDriver.Modulation.IQ.UploadArbAgilentFile(ArbRef, FileName);
                VsgDriver.Modulation.Enabled = true;
                VsgDriver.Modulation.BasebandPower = -10;
                // Play the ARB, sending all changes to hardware
                VsgDriver.Modulation.PlayArb(ArbRef,
    AgM938xStartEventEnum.AgM938xStartEventImmediate);
                VsgDriver.Modulation.Scale = 0.5;
                VsgDriver.Apply();

                // Get the Sample Rate and RMS Value (Peak to Average Ratio) of
                            // the Current Waveform
                AgM938xMarkerEnum RfBlankMarker =
    AgM938xMarkerEnum.AgM938xMarkerNone;
                AgM938xMarkerEnum AlcHoldMarker =
    AgM938xMarkerEnum.AgM938xMarkerNone;
                VsgDriver.Modulation.IQ.ArbInformation(ArbRef, ref SampleRate,
    ref RmsValue, ref ScaleFactor, ref RfBlankMarker, ref AlcHoldMarker);

                // Setup the RF Path in the Receiver
                VsaDriver.RF.Frequency = Frequency;
                VsaDriver.RF.Power = Level + PowerOffset;
                VsaDriver.RF.Conversion =
    AgM9391ConversionEnum.AgM9391ConversionAuto;
                VsaDriver.RF.PeakToAverage = RmsValue;
                VsaDriver.RF.IFBandwidth = IfBandwidth;
                // Configure the Acquisition
                VsaDriver.AcquisitionMode =
    AgM9391AcquisitionModeEnum.AgM9391AcquisitionModePower;
```

```
            VsaDriver.PowerAcquisition.Bandwidth = MeasureBW;
            VsaDriver.PowerAcquisition.Duration = ChannelTime;
            VsaDriver.PowerAcquisition.ChannelFilter.Configure(FilterType,
FilterAlpha, FilterBw);
            // Send Changes to hardware
            VsaDriver.Apply();
            VsaDriver.WaitUntilSettled(100);

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

                // Run a group of baseband power commands to change the
source
                                // level and make a power measurement at each
step.
                // Simulates Servo loop timing, but does not use the measured
                                // power to adjust the next source level
                VsaDriver.PowerAcquisition.Duration = ChannelTime;
                VsaDriver.Apply();
                double[] LevelOffset = new double[] {-3, -2, -1, -0.5, -
0.75};
                for (int Index = 0;Index < LevelOffset.Length - 1;Index++) {
                    VsgDriver.Modulation.BasebandPower = LevelOffset[Index];
                    VsgDriver.Apply();
                    VsaDriver.Arm();
                    VsaDriver.PowerAcquisition.ReadPower(0,
AgM9391PowerUnitsEnum.AgM9391PowerUnitsdBm, ref MeasuredPower, ref Overload);
                }

                // Loop Through the channel offset frequencies for an
                                // ACPR measurement
                // Use the last value of the servo loop for the channel power
                MeasuredChannelPower = MeasuredPower;
                ChannelPowerOverload = Overload;
                VsaDriver.PowerAcquisition.Duration = AdjacentTime;
                for (int Index = 0;Index < FreqOffset.Length;Index++) {
                    VsaDriver.PowerAcquisition.OffsetFrequency = FreqOffset
[Index];
                    VsaDriver.Apply();
                    VsaDriver.Arm();
                    VsaDriver.PowerAcquisition.ReadPower(0,
AgM9391PowerUnitsEnum.AgM9391PowerUnitsdBm, ref MeasuredPower, ref Overload);
                    MeasuredACPR[Index] = MeasuredPower -
MeasuredChannelPower;
                }

                // Make sure the VSA frequency offset is back to 0 (on
repeat)
```

```
                VsaDriver.PowerAcquisition.OffsetFrequency = 0;
                VsaDriver.Apply();
                if (ChannelPowerOverload == true) {
                    Console.WriteLine("Channel Power Measurement Overload");
                }
                Console.WriteLine("Channel Power:  {0} dBm",
MeasuredChannelPower);
                Console.WriteLine("ACPR1 L:  {0} dBc", MeasuredACPR[0]);
                Console.WriteLine("ACPR1 U:  {0} dBc", MeasuredACPR[1]);
                Console.WriteLine("ACPR2 L:  {0} dBc", MeasuredACPR[2]);
                Console.WriteLine("ACPR2 U:  {0} dBc", MeasuredACPR[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 (VsgDriver != null && VsgDriver.Initialized)
        {
          // Close the driver
          VsgDriver.Close();
          Console.WriteLine("VSG Driver Closed");
        }

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

      }
      #endregion

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

Disclaimer



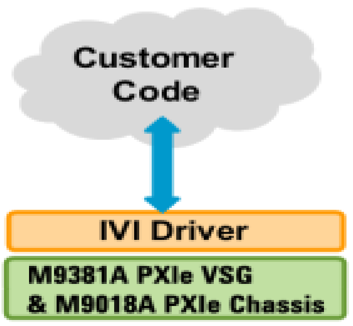
```
VSA Driver Initialized

VSG Driver Initialized
VSG ErrorQuery: 0, No Error.
VSA ErrorQuery: 0, No Error.

Press Enter to Run Test

Channel Power:  2.06747628596215 dBm
ACPR1 L:  -68.2805836797292 dBc
ACPR1 U:  -68.1435303761513 dBc
ACPR2 L:  -69.0805783603514 dBc
ACPR2 U:  -68.6461396479006 dBc
Repeat? y/n
n
VSG Driver Closed
VSA Driver Closed

Done - Press Enter to Exit
```

Working with 802.11ac MIMO RnD and DVT Tests

This section of the programming guide focuses on performing 802.11ac MIMO R&D/DVT Tests related to Receiver and Transmitter (Rx/Tx) Physical Layer performance characterization.

NOTE This programming guide may also be used when creating programs for 802.11a,b,g and n.

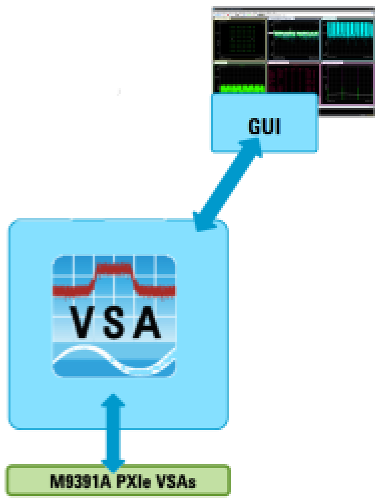
Receiver Tests, with IVI-COM controlling M9381A PXIe VSGs and M9018A PXIe Chassis



In this section, you will learn how to use Visual Studio 2010 to write source code for an IVI-COM Console Application in Visual C#. You will compile the source code using the .NET Framework Library and produce an Assembly.exe file that routes backplane triggers on the M9018A PXIe Chassis for the M9381A PXIe VSGs, M9300A PXIe References, and M9391A PXIe VSAs. It then sets the controls for the M9381A PXIe VSGs and starts them playing a waveform file.

Knowledge of Visual Studio 2010 with the .NET Framework, knowledge of the programming syntax for Visual C#, and knowledge of 802.11ac PHY Layer receiver tests is required.

Transmitter Tests, with 89600 VSA Software controlling M9391A PXIe Vector Signal Analyzers



Also, you will learn how to use Keysight 89600 VSA Software to control M9391A PXIe VSAs that perform transmitter tests for PHY Layer characterization.

Knowledge of 89600 VSA Software and knowledge of 802.11ac PHY Layer transmitter tests is required.

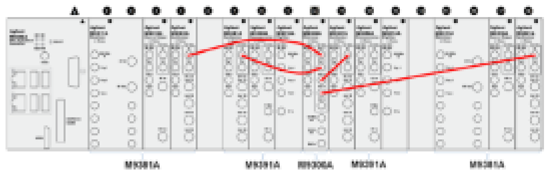
The following 802.11ac MIMO R&D/DVT Tests related to Rx/Tx PHY Layer characterization are covered:

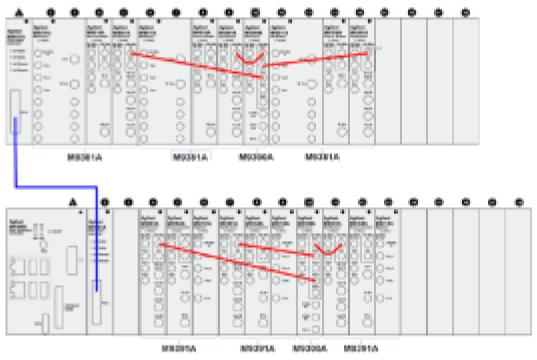
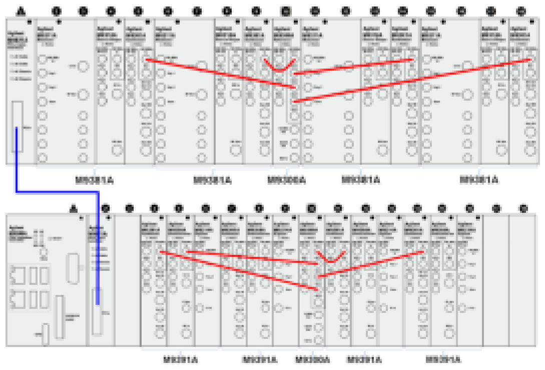
Example Program 4: How to Perform Transmitter Tests with 89600 VSA Software(Playing Waveforms on M9381A PXIe VSGs, Using External Trigger)

Preparing the Hardware and Software for 802.11ac MIMO RnD DVT Tests

1. **Select and Configure a Hardware Configuration for 802.11ac MIMO R&D/DVT Test**

   An overview of all hardware modules and their cable connections are shown here for convenience, but detailed steps on each of the following configurations, as well as configurations with external controllers, are defined in the *Keysight MIMO PXI Test Solution Startup Guide, Y1299-90001*– complete all hardware configuration steps before trying to programmatically control M9381A PXIe VSGs and M9018A PXIe Chassis hardware with an IVI driver or to control M9391A PXIe VSAs with 89600 VSA Software.

Configuration	Description
2x2 MIMO in One M9018A PXIe Chassis 	This configuration (occupying 16 slots) consists of two M9381A PXIe VSGs and two M9391A PXIe VSAs in a single M9018A PXIe Chassis – with or without an M9036A PXIe Embedded Controller (the configuration shown includes an M9036A PXIe Embedded Controller).A single M9300A PXIe Reference is shared across both M9381A PXIe VSGs and both M9391A PXIe VSAs.
3x3 MIMO in Two M9018A PXIe Chassis	The top of this configuration (occupying 14 slots) consists of three M9381A PXIe VSGs in a single M9018A PXIe Chassis – with an M9021A PCIe Cable Interface. A single M9300A PXIe Reference, in slot 10, is

Configuration	Description
	shared across all three M9381A PXIe VSGs. The bottom of this configuration (occupying 11 slots) consists of three M9391A PXIe VSAs in a single M9018A PXIe Chassis – with an M9021A PCIe Cable Interface and an M9036A PXIe Embedded Controller.A single M9300A PXIe Reference, in slot 10, is shared across all three M9391A PXIe VSAs.
4x4 MIMOin Two M9018A PXIe Chassis 	The top of this configuration (occupying 18 slots) consists of four M9381A PXIe VSGs in a singleM9018A PXIe Chassis – with an M9021A PCIe Cable Interface.A single M9300A PXIe Reference, in slot 10, is shared across all four M9381A PXIe VSGs. The bottom of this configuration (occupying 14 slots) consists of four M9391A PXIe VSAs in a single M9018A Chassis – with an M9021A PCIe Cable Interface and an M9036A PXIe Embedded Controller.A single M9300A PXIe Reference, in slot 10, is shared across all four M9391A PXIe VSAs.

2. Install / Verify Software for 802.11ac MIMO R&D/DVT Test
   Refer to the *Installing Hardware, Software, and Licenses* section for a list of software that is needed as well as the order that software must be installed.
3. Programmatically control M9381A PXIe VSGs with C# Console Application
4. Create driver instances, set resource names and various initialization values.

5. Add references, initialize drivers, and clear the error queues for an M9018A PXIe Chassis, an M9300A PXIe Reference, and two, three, or four M9381A PXIe VSG driver instances.
6. Route/Configure a number of backplane triggers on the M9018A PXIe Chassis.

NOTE **Running Self-Test will fail if the modules that form an M9381A PXIe VSG or an M9391A PXIe VSA spans across slot 6 or slot 12 of the M9018A PXIe Chassis; if they do span across slot 6 or slot 12, the backplane triggers and bus segments must be routed properly. For details, see *Step 6 – Route Backplane Triggers and Bus Segments on the M9018A PXIe Chassis*.**

7. On each M9381A PXIe VSG in the selected configuration:
   - Set up some initial values on each M9381A PXIe VSG.
   - Load and start playing a waveform file on each M9381A PXIe VSG.
   - The waveform file continues playing until the console application is closed.
8. **Create an N-Channel M9391A PXIe VSA Configuration in the 89600 VSA Software**.
   Include the M9300A PXIe Reference module in Channel 1, but not in subsequent channels.

NOTE **When creating Channel 1, 2, 3, and 4 configurations with 89600 VSA software, note the order in which you add M9391 channels to the hardware configuration. This is needed later to determine which channel is used to receive an external, software or magnitude trigger, and it will be needed for trigger allocation on the PXI backplane as well.**

9. **Use 89600 VSA Software to Control M9391A PXIe VSAs and Perform Transmitter Tests**

For each M9391A PXIe VSA, M9381A PXIe VSG, and M9018A PXIe Chassis used in the configuration, use their soft front panels to verify that they have the most up to date FPGA bits, and **after routing backplane triggers and bus segments**, run self-test on each M9391A PXIe VSA, M9381A PXIe VSG, M9300A PXIe Reference, and M9018A PXIe Chassis - one at a time.

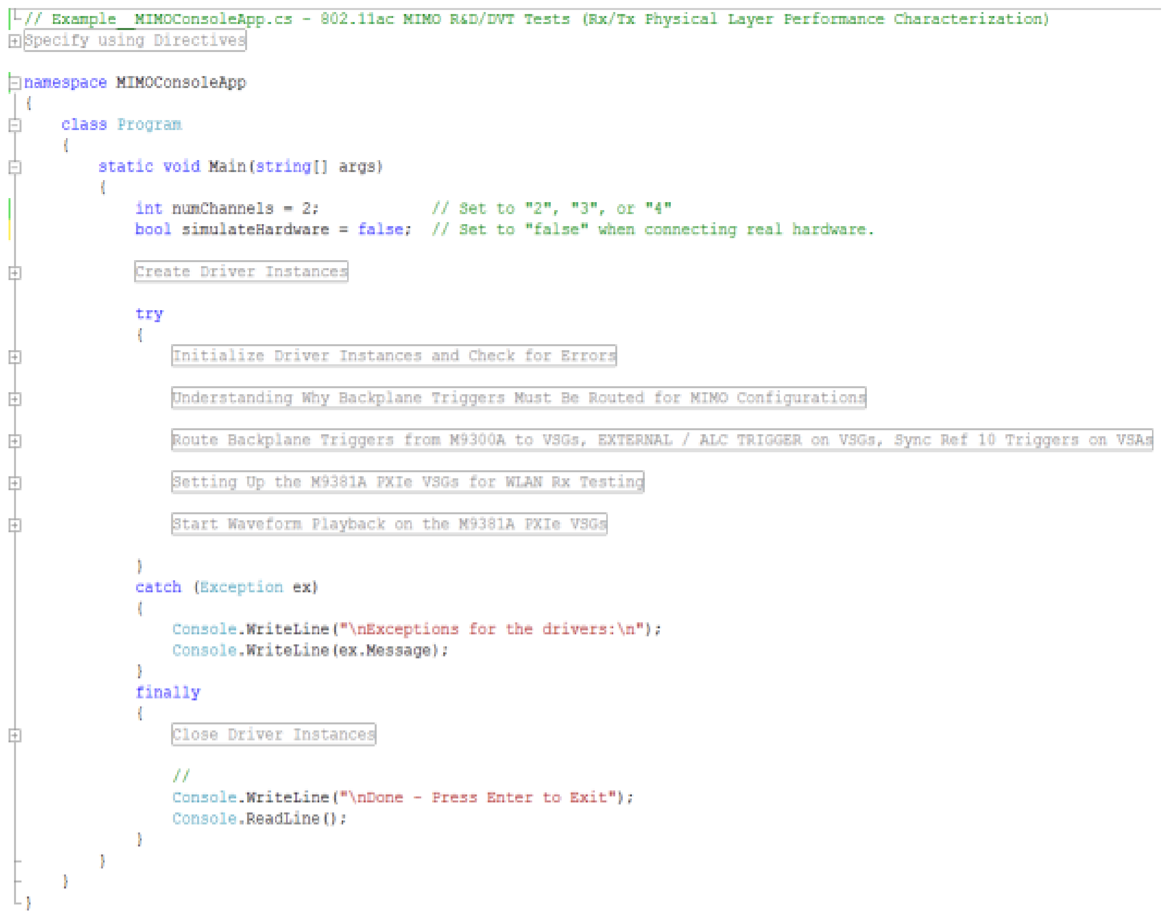
Example Program 4 - How to Perform Transmitter Tests with 89600 VSA Software

The following example code builds on the previously presented *Tutorial: Creating a Project with IVI-COM Using C#*. It demonstrates how to instantiate two, three, or four M9381A PXIe VSG driver instances, set resource names and various initialization values, initialize and clear the error queues for the M9018A PXIe Chassis, an M9300A PXIe Reference, and two, three, or four M9381A PXIe VSG driver instances.

Once initialized, a number of backplane triggers on the M9018A PXIe Chassis are configured, initial values are set on each M9381A PXIe VSG, and a waveform file is loaded and played on each M9381A PXIe VSG – this waveform can then be analyzed with 89600 VSA Software that is controlling two, three, or four M9391A PXIe VSAs.

The console application window that is running remains open while the waveform file is playing – if the console application window is closed by selecting the Enter key, the waveform file will stop playing and all of the M9381A PXIe VSG drivers will be closed.

This example program can be accessed from **C:\Program Files (x86)\Agilent\M9391\Help\Examples\Example__MIMOConsoleApp.cs** and **C:\Program Files (x86)\Agilent\M938x\Help\Examples\Example__MIMOConsoleApp.cs**.


```
// Example__MIMOConsoleApp.cs - 802.11ac MIMO R&D/DVT Tests (Rx/Tx Physical Layer Performance Characterization)
Specify using Directives

namespace MIMOConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int numChannels = 2;           // Set to "2", "3", or "4"
            bool simulateHardware = false;  // Set to "false" when connecting real hardware.

            Create Driver Instances

            try
            {
                Initialize Driver Instances and Check for Errors

                Understanding Why Backplane Triggers Must Be Routed for MIMO Configurations

                Route Backplane Triggers from M9300A to VSGs, EXTERNAL / ALC TRIGGER on VSGs, Sync Ref 10 Triggers on VSAs

                Setting Up the M9381A PXIe VSGs for WLAN Rx Testing

                Start Waveform Playback on the M9381A PXIe VSGs

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

                //
                Console.WriteLine("\nDone - Press Enter to Exit");
                Console.ReadLine();
            }
        }
    }
}
```

Example Program4 - Pseudo-Code

- Initialize Drivers for M9300A, M9018A, M9381A Channel 1 and Channel 2 // For a 2x2 MIMO Configuration
- Initialize Drivers for M9300A, M9018A, M9381A Channel 1, Channel 2, and Channel 3 // For a 3x3 MIMO Configuration

- Initialize Drivers for M9300A, M9018A, M9381A Channel 1, Channel 2, Channel 3, and Channel 4 // For a 4x4 MIMO Configuration
- Check each Driver for Errors
- Route/Configure the Backplane Triggers on the M9018A PXIe Chassis (2x2 MIMO)
    - Route Backplane **External Trigger from M9300A**to each M9311A (VSG)
        - Connect Bus 2 to 1 and 3 so that the M9300A PXIe Reference can generate a 'Synchronization Playback Trigger' on PXI TRIG 0 ...this trigger will be received by VSG 1 and VSG 2// (M9300A trigger from slot 10 to 2, 15, 7, 11)
    - Use **Default Backplane Triggers** for VSG1
        - Uses Bus Segment 1: 'EXTERNAL TRIGGER' M9301A to M9311A (slot 5 to 2)
        - VSG1 TRIG 7: Uses Bus Segment 1: 'ALC TRIGGER' M9311A to M9310A (slot 2 to 4)
    - Use **Default Backplane Triggers** for VSG2
        - VSG2 TRIG 6: Uses Bus Segment 3: 'EXTERNAL TRIGGER' M9301A to M9311A" (slot 18 to 15)
        - VSG2 TRIG 7: Uses Bus Segment 3: 'ALC TRIGGER' M9311A to M9310A" (slot 15 to 17)
    - Route **MASTER/SLAVE Backplane Triggers** for VSAs in 2x2 MIMO Chassis 1:
        - MASTER/SLAVE Backplane Triggers are used to coordinate the capture alignments on the VSA
        - M9391A PXIe VSA1 to VSA2 TRIG 1: Connect Bus 2 to 3: MASTER to SLAVE
        - M9391A PXIe VSA2 to VSA1 TRIG 2: Connect Bus 3 to 2: SLAVE to MASTER

NOTE **See the following code examples for additional trigger changes needed for a 3x3 or 4x4 MIMO configuration.**

Set Controls for all M9381A PXIe VSGs

- Configure PXI TRIG 0 to listen for 'Playback Synchronization Pulse'
- Disable ALC
- Enable Pulse Blanking
- Set PLL Mode to Best Wide Offset
- Set RF Frequency
- Set Amplitude
- Enable Modulation
- Enable RF Out

Start Waveform Playback on all M9381A PXIe VSGs

– Play a Waveform on all M9381A PXIe VSGs - Using External Trigger Mode (This trigger is from M9300A on PXI TRIG 0.)

Analyze the Waveform being produced from M9381A PXIe VSGs with 89600 VSA Software controlling M9391A PXIe VSAs.

Step 1 : Create a Console Application

Perform the following steps:

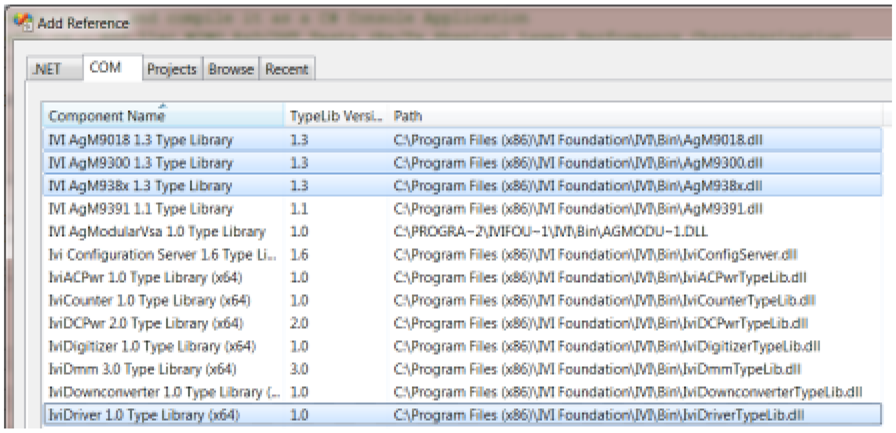
1. Launch Visual Studio and create a new Console Application in Visual C# by selecting: **File > New > Project** and select a Visual C# Console Application.
   Enter "MIMOConsoleApp" as the **Name** of the project and click **OK**.

2. 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 M9018A PXIe Chassis, M9300A PXIe Reference, and M9381A PXIe VSG 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.
3. 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".)



Component Name	TypeLib Versi...	Path
IVI AgM9018 1.3 Type Library	1.3	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM9018.dll
IVI AgM9300 1.3 Type Library	1.3	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM9300.dll
IVI AgM938x 1.3 Type Library	1.3	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM938x.dll
IVI AgM9391 1.1 Type Library	1.1	C:\Program Files (x86)\IVI Foundation\IVI\Bin\AgM9391.dll
IVI AgModularVsa 1.0 Type Library	1.0	C:\PROGRA~2\IVIFOU~1\IVI\Bin\AGMODU~1.DLL
Ivi Configuration Server 1.6 Type Li...	1.6	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviConfigServer.dll
IviACPwr 1.0 Type Library (x64)	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviACPwrTypeLib.dll
IviCounter 1.0 Type Library (x64)	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviCounterTypeLib.dll
IviDCPwr 2.0 Type Library (x64)	2.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviDCPwrTypeLib.dll
IviDigitizer 1.0 Type Library (x64)	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviDigitizerTypeLib.dll
IviDmm 3.0 Type Library (x64)	3.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviDmmTypeLib.dll
IviDownconverter 1.0 Type Library (...	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviDownconverterTypeLib.dll
IviDriver 1.0 Type Library (x64)	1.0	C:\Program Files (x86)\IVI Foundation\IVI\Bin\IviDriverTypeLib.dll

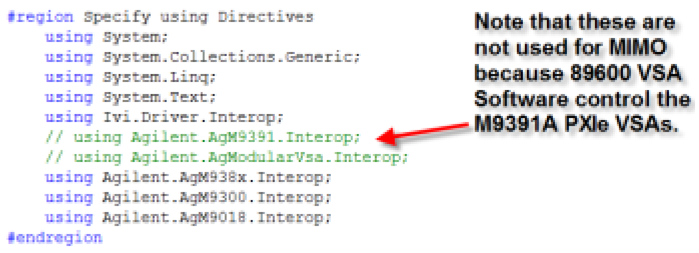
4. Scroll to the IVI section and, using Shift-Ctrl, select the following type libraries then select **OK**.

```
IVI AgM9018 1.3 Type Library
IVI AgM9300 1.3 Type Library
IVI AgM938x 1.3 Type Library
```

NOTE

When any of the references for AgM9018, AgM9300, or AgM938x are added, the IVIDriver 1.0 Type Library is also automatically added. This reference houses the interface definitions for IVI inherent capabilities which are located in the file IviDriverTypeLib.dll (dynamically linked library).

Step 3 : Add Using Statements

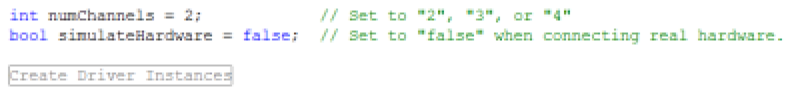


```
#region Specify using Directives
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Ivi.Driver.Interop;
    // using Agilent.AgM9391.Interop;
    // using Agilent.AgModularVsa.Interop;
    using Agilent.AgM938x.Interop;
    using Agilent.AgM9300.Interop;
    using Agilent.AgM9018.Interop;
#endregion
```

Note that these are not used for MIMO because 89600 VSA Software control the M9391A PXIe VSAs.

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.

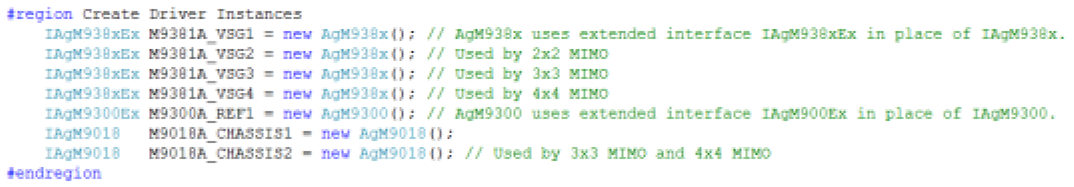
Step 4 - Create Driver Instances



```
int numChannels = 2;            // Set to "2", "3", or "4"
bool simulateHardware = false;  // Set to "false" when connecting real hardware.

Create Driver Instances
```

Change the value of numChannels to either: 2, 3, or 4 depending on the number of channels you would like to control with the console application.

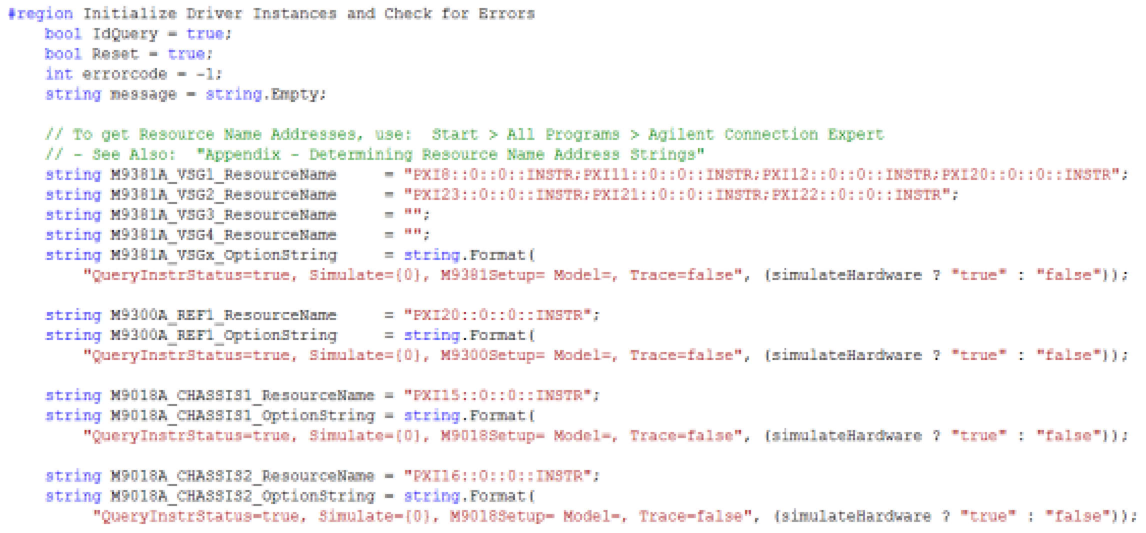
To create driver instances



```
#region Create Driver Instances
    IAgM938xEx M9381A_VSG1 = new AgM938x(); // AgM938x uses extended interface IAgM938xEx in place of IAgM938x.
    IAgM938xEx M9381A_VSG2 = new AgM938x(); // Used by 2x2 MIMO
    IAgM938xEx M9381A_VSG3 = new AgM938x(); // Used by 3x3 MIMO
    IAgM938xEx M9381A_VSG4 = new AgM938x(); // Used by 4x4 MIMO
    IAgM9300Ex M9300A_REF1 = new AgM9300(); // AgM9300 uses extended interface IAgM900Ex in place of IAgM9300.
    IAgM9018   M9018A_CHASSIS1 = new AgM9018();
    IAgM9018   M9018A_CHASSIS2 = new AgM9018(); // Used by 3x3 MIMO and 4x4 MIMO
#endregion
```

When creating driver instances for the M9381A PXI VSG, note that the IAgM938xEx interface is needed for MIMO in place of the IAgM938x interface. The IAgM938xEx interface is an "extended" interface and includes additional commands that are needed for MIMO that were not previously available in IAgM938x.

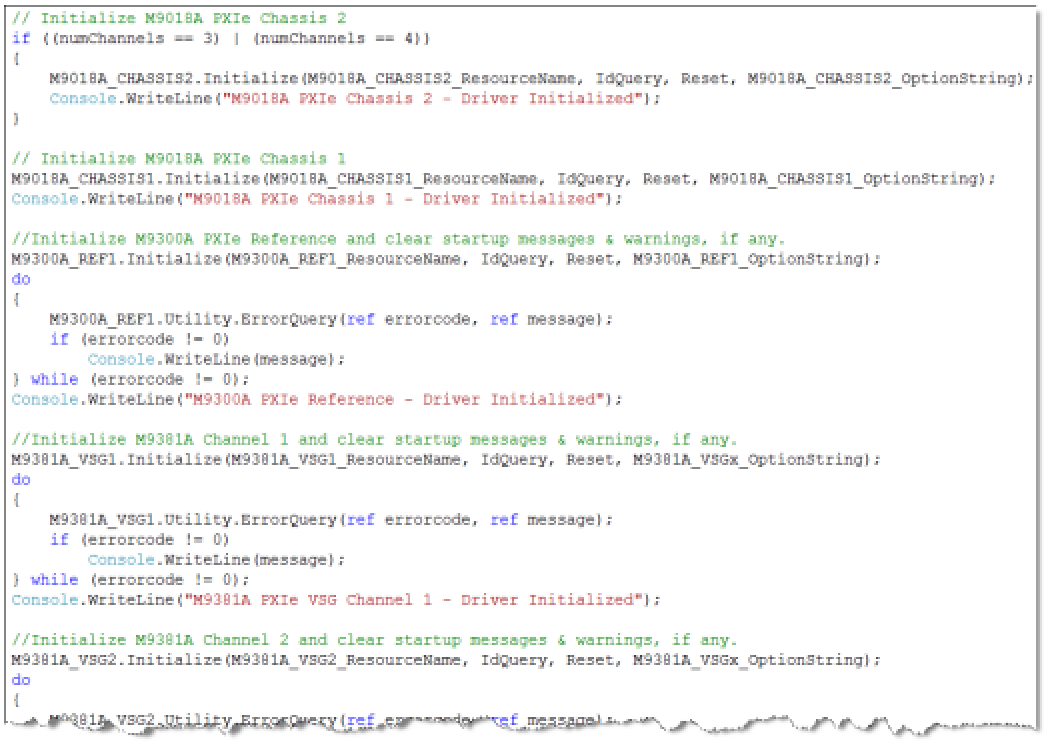
Step 5 - Initialize Driver Instances and Check for Errors



```
#region Initialize Driver Instances and Check for Errors
    bool IdQuery = true;
    bool Reset = true;
    int errorcode = -1;
    string message = string.Empty;

    // To get Resource Name Addresses, use:  Start > All Programs > Agilent Connection Expert
    // - See Also:  "Appendix - Determining Resource Name Address Strings"
    string M9381A_VSG1_ResourceName     = "PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI20::0::0::INSTR";
    string M9381A_VSG2_ResourceName     = "PXI23::0::0::INSTR;PXI21::0::0::INSTR;PXI22::0::0::INSTR";
    string M9381A_VSG3_ResourceName     = "";
    string M9381A_VSG4_ResourceName     = "";
    string M9381A_VSGx_OptionString     = string.Format(
        "QueryInstrStatus=true, Simulate={0}, M9381Setup= Model=, Trace=false", (simulateHardware ? "true" : "false"));

    string M9300A_REF1_ResourceName     = "PXI20::0::0::INSTR";
    string M9300A_REF1_OptionString     = string.Format(
        "QueryInstrStatus=true, Simulate={0}, M9300Setup= Model=, Trace=false", (simulateHardware ? "true" : "false"));

    string M9018A_CHASSIS1_ResourceName = "PXI15::0::0::INSTR";
    string M9018A_CHASSIS1_OptionString = string.Format(
        "QueryInstrStatus=true, Simulate={0}, M9018Setup= Model=, Trace=false", (simulateHardware ? "true" : "false"));

    string M9018A_CHASSIS2_ResourceName = "PXI16::0::0::INSTR";
    string M9018A_CHASSIS2_OptionString = string.Format(
         "QueryInstrStatus=true, Simulate={0}, M9018Setup= Model=, Trace=false", (simulateHardware ? "true" : "false"));
```

To Establish a Communication Link, get the Resource Name Addresses

Use either **Agilent/Keysight Connection Expert** or see "Appendix – Determining Resource Name Address Strings".



```
// Initialize M9018A PXIe Chassis 2
if ((numChannels == 3) | (numChannels == 4))
{
    M9018A_CHASSIS2.Initialize(M9018A_CHASSIS2_ResourceName, IdQuery, Reset, M9018A_CHASSIS2_OptionString);
    Console.WriteLine("M9018A PXIe Chassis 2 - Driver Initialized");
}

// Initialize M9018A PXIe Chassis 1
M9018A_CHASSIS1.Initialize(M9018A_CHASSIS1_ResourceName, IdQuery, Reset, M9018A_CHASSIS1_OptionString);
Console.WriteLine("M9018A PXIe Chassis 1 - Driver Initialized");

//Initialize M9300A PXIe Reference and clear startup messages & warnings, if any.
M9300A_REF1.Initialize(M9300A_REF1_ResourceName, IdQuery, Reset, M9300A_REF1_OptionString);
do
{
    M9300A_REF1.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
        Console.WriteLine(message);
} while (errorcode != 0);
Console.WriteLine("M9300A PXIe Reference - Driver Initialized");

//Initialize M9381A Channel 1 and clear startup messages & warnings, if any.
M9381A_VSG1.Initialize(M9381A_VSG1_ResourceName, IdQuery, Reset, M9381A_VSGx_OptionString);
do
{
    M9381A_VSG1.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
        Console.WriteLine(message);
} while (errorcode != 0);
Console.WriteLine("M9381A PXIe VSG Channel 1 - Driver Initialized");

//Initialize M9381A Channel 2 and clear startup messages & warnings, if any.
M9381A_VSG2.Initialize(M9381A_VSG2_ResourceName, IdQuery, Reset, M9381A_VSGx_OptionString);
do
{
    M9381A_VSG2.Utility.ErrorQuery(ref errorcode, ref message);
```

The above is a partial view of code that shows how to initialize each of the drivers needed. The complete example program can be accessed from **C:\Program Files (x86)**

\Agilent\M9391\Help\Examples\Example__MIMOConsoleApp.cs and C:\Program Files (x86)\Agilent\M938x\Help\Examples\Example__MIMOConsoleApp.cs.

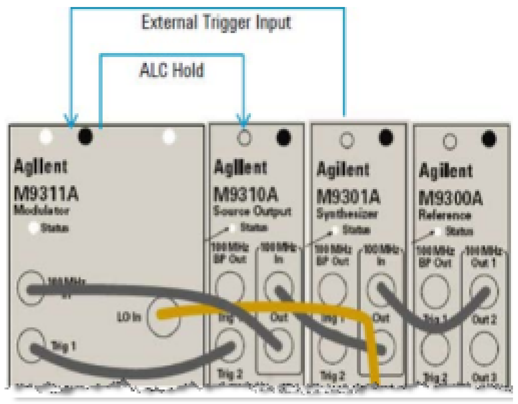
Step 6 - Route Backplane Triggers and Bus Segments on the M9018A PXIe Chassis

In a MIMO configuration, there are a number of backplane triggers that are required for each M9381A PXIe VSG and they must be routed properly across backplane trigger bus segments on the M9018A PXIe Chassis.

- The M9018A PXIe Chassis contains three PXI Trigger Bus Segments:
  - Bus Segment 1 is used by Slots 1 to 6
  - Bus Segment 2 is used by Slots 7 to 12
  - Bus Segment 3 is used by Slots 13 to 18
- By default, each Bus Segment is isolated from the other two Bus Segments.
- Also, each Bus Segment has its own set of PXI Trigger lines named:PXI TRIG 0, PXI TRIG 1, PXI TRIG 2, PXI TRIG 3, PXI TRIG 4, PXI TRIG 5, PXI TRIG 6, PXI TRIG 7.

Routing an External Trigger Input and ALC Hold on each M9381A PXIe VSG

There are three requirements for each M9381A PXIe VSG in the system:



1. There must be a PXI trigger from M9301A to M9311A for front panel External Trigger Input (Default: PXI TRIG 6).
2. There must be a PXI trigger from M9311A to M9310A for ALC Hold (Default: PXI TRIG 7).
3. None of these triggers can conflict with each other. For example, if there are two or more M9381A PXIe VSGs (or portions of a M9381A PXIe VSG) occupying a single Bus Segment, they cannot all use the default trigger lines.

To satisfy the requirements for each M9381A PXIe VSG in the system, listed above:

1. Route triggers between bus segments on the M9018A PXIe Chassis
2. Redefine the triggers used by the M9381A PXIe VSGs.

NOTE **If a hardware configuration with M9381A PXIe VSG modules spans across an M9018A PXIe Chassis bus segment (slot 6 or slot 12), it is necessary to route/configure the backplane triggers on the M9018A**

PXIe Chassis for External Trigger Input (Default: PXI TRIG 6) and ALC Hold (Default: PXI TRIG 7) – this routing/configuring must be performed prior to verifying that each M9381A PXIe VSG can connect and run self-test.

Routing a Synchronization Playback Trigger from the M9300A PXIe Reference to each M9381A PXIe VSG

Because the M9018A PXIe Chassis has three isolated Bus Segments, a PXI TRIG line must be selected to deliver a 'Synchronization Playback Trigger' that will be sent to each M9381A PXIe VSG in the system – in order for each M9381A PXIe VSG to receive this trigger, the Bus Segments must be connected when coming from the M9300A PXIe Reference.

Since the M9300A PXIe Reference is located in slot 10, which is in Bus Segment 2, the Bus Segment 1 and Bus Segment 3 must be connected on the PXI TRIG line that will be used; in our examples, PXI TRIG 0 has been selected to be this PXI TRIG line.

So, the following command allows the 'Synchronization Playback Trigger' to be sent from the M9300A PXIe Reference to each M9381A PXIe VSG in the system.

```
 M9018A_CHASSIS1.TriggerBus.Connect(0,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1And3);
```

Routing MASTER_SLAVE Backplane Triggers for each M9391A PXIe VSA

The MASTER/SLAVE Backplane Triggers are used to coordinate the capture alignments on the M9391A PXIe VSAs. In each system, one of the M9391A PXIe VSAs must be designated as the MASTER and each of the other M9391A PXIe VSAs in a system must be designated as a SLAVE to this MASTER.

Because the M9018A PXIe Chassis has three isolated Bus Segments, a PXI TRIG line must be selected to deliver MASTER/SLAVE Backplane Triggers that will be sent from the MASTER to the last SLAVE in each system – and each SLAVE must also be able to send back a signal to the MASTER.

Each of these MASTER/SLAVE Backplane Triggers must be on a separate PXI TRIG line and must be able to propagate from one Bus Segment to the other.

2x2 MIMO Configuration

In our example, PXI TRIG 1 is used to send the MASTER/SLAVE Backplane Triggers from the MASTER to the SLAVE and the Bus Segments must be connected coming from Bus Segment 2 to Bus Segment 3.
In order for the MASTER/SLAVE Backplane Triggers to be sent between the MASTER and each SLAVE:

```
M9018A_CHASSIS1.TriggerBus.Connect(1,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To3);
Console.WriteLine("\tVSA1 to VSA2 TRIG 1: Connect Bus 2 to 3: MASTER to
SLAVE");


M9018A_CHASSIS1.TriggerBus.Connect(2,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus3To2);
Console.WriteLine"\tVSA2 to VSA1 TRIG 2: Connect Bus 3 to 2: SLAVE to
MASTER");
```

Examples with a 3x3 and 4x4 MIMO configuration are shown in the following code examples. The complete example program can be accessed from C:\Program Files (x86)\Agilent\M9391\Help\Examples\Example__MIMOConsoleApp.cs and C:\Program Files (x86)\Agilent\M938x\Help\Examples\Example__MIMOConsoleApp.cs.

The following code examples show the backplane trigger routings for:

- 2x2 MIMO
- 3x3 MIMO
- 4x4 MIMO

2x2 MIMO

```
//(numChannels == 2)
{
// Two M9381A PXIe VSGs and Two M9391A PXIe VSAs in one M9018A PXIe Chassis
// /--------:-----------------:-----------------:-----------------
// |        :                 :                 :                 :
// |        :<-Bus Segment 1->:<-Bus Segment 2->:<-Bus Segment 3->:
// |        :                 :                 :                 :
// |        : 1| 2--3| 4| 5| 6: 7| 8| 9|10|11|12:13|14|15-16|17|18:
// |        :  %<---VSG1-->%  %<-VSA1->%R %<-VSA2->%  %<---VSG2-->%
// |        :                 :         e       :                 :
// |        :                 :         f       :                 :
// |        :                 :                 :                 :
// \--------------------------------------------------------------/
//
Console.WriteLine("\nRoute Backplane Triggers for VSGs in 2x2 MIMO Chassis
1:");
// Route Backplane External Trigger from M9300A to each M9311A (VSG) for
// Playback synchronization (slot 10 to 2, 15, 7, 11)
M9018A_CHASSIS1.TriggerBus.Connect(0,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1And3);
Console.WriteLine("\tVSGs TRIG 0: Connect Bus 2 to 1 and 3: \n\tso that the
M9300A PXIe Reference can generate a \n\t\'Synchronization Playback Trigger\'
on PXI TRIG 0 \n\t...this trigger will be received by VSG 1 and VSG 2.\n");

// Use Default Backplane Triggers for VSG1
Console.WriteLine("\tVSG1 TRIG 6: Uses Bus Segment 1: 'EXTERNAL TRIGGER'
M9301A to M9311A"); // (slot 5 to 2)
Console.WriteLine("\tVSG1 TRIG 7: Uses Bus Segment 1: 'ALC HOLD TRIGGER'
M9311A to M9310A"); // (slot 2 to 4)

// Use Default Backplane Triggers for VSG2
Console.WriteLine("\tVSG2 TRIG 6: Uses Bus Segment 3: 'EXTERNAL TRIGGER'
M9301A to M9311A"); // (slot 18 to 15)
Console.WriteLine("\tVSG2 TRIG 7: Uses Bus Segment 3: 'ALC HOLD TRIGGER'
M9311A to M9310A"); // (slot 15 to 17)

Console.WriteLine("\nRoute MASTER/SLAVE Backplane Triggers for VSAs in 2x2
MIMO Chassis 1:");
// MASTER/SLAVE Backplane Triggers are used to coordinate the
// capture alignments on the VSAs
M9018A_CHASSIS1.TriggerBus.Connect(1,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To3);
Console.WriteLine("\tVSA1 to VSA2 TRIG 1: Connect Bus 2 to 3: MASTER to
SLAVE");

M9018A_CHASSIS1.TriggerBus.Connect(2,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus3To2);
Console.WriteLine("\tVSA2 to VSA1 TRIG 2: Connect Bus 3 to 2: SLAVE to
MASTER");
}
```

3x3 MIMO

```
// (numChannels == 3)
{
// Three M9381A PXIe VSGs in Chassis 1   (Chassis 2 holds Three M9391A PXIe
VSAs)
// /--------:-----------------:-----------------:-----------------
// |        :                 :                 :                 :
// |        :<-Bus Segment 1->:<-Bus Segment 2->:<-Bus Segment 3->:
// |        :                 :                 :                 :
// |        : 1| 2--3| 4| 5| 6--7| 8| 9|10|11-12:13|14|15|16|17|18:
// |        :  %<---VSG1-->%<---VSG2-->%R %<---VSG3-->%<--Blank-->%
// |        :                 :         e       :                 :
// |        :                 :         f       :                 :
// \--------------------------------------------------------------/
//
```

```
// M9381A PXIe VSG1 in slots 2-5, VSG2 in 6-9, and VSG3 in 11-14
// M9300A PXIe Reference in slot 10
// To assure the ALC Hold and front panel External Trigger routings within
each VSG
// don't conflict with adjacent VSGs, assign a trigger to use for the user
generated
// backplane trigger, from the M9300A PXIe Reference to all M9311As (VSGs),
// that does not conflict with any other trigger and then route it
appropriately on
// the chassis backplane.
//
// VSG1 and VSG3 use default triggers, PXI TRIG 6 for front panel 'EXTERNAL
TRIGGER'
// and PXI TRIG 7 for 'ALC HOLD'.
// VSG2 uses PXI TRIG 4 for front panel 'EXTERNAL TRIGGER' and PXI TRIG 5 for
// 'ALC HOLD'.
// Use PXI TRIG 0 for the Backplane External Trigger for playback
synchronization.
//
Console.WriteLine("\nRoute Backplane Triggers for VSGs in 3x3 MIMO Chassis
1:");
// Route Backplane External Trigger from M9300A to each M9311A (VSG) for
// Playback synchronization (slot 10 to 2, 6, 11)
M9018A_CHASSIS1.TriggerBus.Connect(0,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1And3);
Console.WriteLine("\tVSGs TRIG 0: Connect Bus 2 to 1 and 3: \n\tso that the
M9300A PXIe Reference can generate a \n\t\'Synchronization Playback Trigger\'
on PXI TRIG 0 \n\t...this trigger will be received by VSG 1, VSG 2, and VSG
3.\n");
// Use Default Backplane Triggers for VSG1
Console.WriteLine("\tVSG1 TRIG 6: Uses Bus Segment 1: 'EXTERNAL TRIGGER'
M9301A to M9311A"); // (slot 5 to 2)
Console.WriteLine("\tVSG1 TRIG 7: Uses Bus Segment 1: 'ALC TRIGGER' M9311A to
M9310A");      // (slot 2 to 4)

// Redefine Backplane Triggers for VSG2
M9018A_CHASSIS1.TriggerBus.Connect(4, AgM9018TrigBusEnum.AgM9018TrigBus2To1);
// 'EXTERNAL TRIGGER' from M9301A to M9311A (slot 9 to 6)

M9381A_VSG2.System.PXIResources.AddHint("M9301A", "M9311A", "EXTERNAL
TRIGGER",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_4);
Console.WriteLine("\tVSG2 TRIG 4: Connect Bus 2 to 1: 'EXTERNAL TRIGGER'
M9301A to M9311A");

M9018A_CHASSIS1.TriggerBus.Connect(5, AgM9018TrigBusEnum.AgM9018TrigBus1To2);
// 'ALC HOLD' from M9311A to M9310A (slot 6 to 8)
```

```
M9381A_VSG2.System.PXIResources.AddHint("M9311A", "M9310A", "ALC HOLD",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_5);
Console.WriteLine("\tVSG2 TRIG 5: Connect Bus 1 to 2: 'ALC HOLD' M9311A to
M9310A");

// Use Default Backplane Triggers for VSG3, but have to connect Bus Segment 2
and 3
M9018A_CHASSIS1.TriggerBus.Connect(6, AgM9018TrigBusEnum.AgM9018TrigBus3To2);
// 'EXTERNAL TRIGGER' from M9301A to M9311A (slot 14 to 11)

Console.WriteLine("\tVSG3 TRIG 6: Connect Bus 3 to 2: 'EXTERNAL TRIGGER'
M9301A to M9311A");
M9018A_CHASSIS1.TriggerBus.Connect(7, AgM9018TrigBusEnum.AgM9018TrigBus2To3);
// 'ALC HOLD' from M9311A to M9310A (slot 11 to 13)
Console.WriteLine("\tVSG3 TRIG 7: Connect Bus 2 to 3: 'ALC HOLD' M9311A to
M9310A");

// Since VSA Chassis 2 is required for a 3x3 MIMO, set up its triggers as
well.
if ( M9018A_CHASSIS2 != null && M9018A_CHASSIS2.Initialized )
{
Console.WriteLine("\nRoute MASTER/SLAVE Backplane Triggers for VSAs in 3x3
MIMO Chassis 2:");
// MASTER/SLAVE Backplane Triggers are used to coordinate the capture
alignments on the VSAs
M9018A_CHASSIS2.TriggerBus.Connect(1,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus1To2To3);
Console.WriteLine("\tVSA1 to VSA3 TRIG 1: Connect Bus 1 to 2 to 3: MASTER to
SLAVE 2");
M9018A_CHASSIS2.TriggerBus.Connect(2,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1);
Console.WriteLine("\tVSA2 to VSA1 TRIG 2: Connect Bus 2 to 1: SLAVE 1 to
MASTER");
M9018A_CHASSIS2.TriggerBus.Connect(3,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus3To2To1);
Console.WriteLine("\tVSA3 to VSA1 TRIG 3: Connect Bus 3 to 2 to 1: SLAVE 2 to
MASTER");
}
}
```

4x4 MIMO

```
// (numChannels == 4)
{
// Four M9381A PXIe VSGs in Chassis 1   (Chassis 2 holds Four M9391A PXIe
VSAs)
// /--------:-----------------:-----------------:-----------------
// |        :                 :                 :                 :
// |        :<-Bus Segment 1->:<-Bus Segment 2->:<-Bus Segment 3->:
// |        :                 :                 :                 :
// |        : 1| 2--3| 4| 5| 6--7| 8| 9|10|11-12:13|14|15-16|17|18:
// |        :  %<---VSG1-->%<---VSG2-->%R %<---VSG3-->%<---VSG4-->%
// |        :                 :         e       :                 :
// | Four M9381A PXIe VSGs    :         f       :                 :
// \-------------------------------------------------------------/
//
// M9381A PXIe VSG1 in slots 2-5, VSG2 in 6-9, VSG3 in 11-14, and VSG4 in 15-
18
// M9300A PXIe Reference in slot 10
// To assure the ALC Hold and front panel External Trigger routings within
each
// VSG do not conflict with adjacent VSGs, assign a trigger to use for the
user
// generated backplane trigger, from the M9300A PXIe Reference to all M9311As
// (VSGs), that does not conflict with any other trigger and then route it
// appropriately on the chassis backplane.
//
// VSG1 and VSG3 use default triggers, PXI TRIG 6 for front panel 'EXTERNAL
// TRIGGER' and PXI TRIG 7 for 'ALC HOLD'.
// VSG2 and VSG4 uses PXI TRIG 4 for front panel 'EXTERNAL TRIGGER' and PXI
// TRIG 5 for 'ALC HOLD'.
// Use PXI TRIG 0 for the Backplane External Trigger for playback
synchronization.
//

Console.WriteLine("\nRoute Backplane Triggers for VSGs in 4x4 MIMO Chassis
1:");
// Route Backplane External Trigger from M9300A to each M9311A (VSG) for
Playback
// synchronization (slot 10 to 2, 6, 11, 15)
M9018A_CHASSIS1.TriggerBus.Connect(0,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1And3);
Console.WriteLine("\tVSGs TRIG 0: Connect Bus 2 to 1 and 3: \n\tso that the
M9300A PXIe Reference can generate a \n\t\'Synchronization Playback Trigger\'
on PXI TRIG 0 \n\t...this trigger will be received by VSG 1, VSG 2, VSG 3,
and VSG 4.\n");

// Use Default Backplane Triggers for VSG1
Console.WriteLine("\tVSG1 TRIG 6: Uses Bus Segment 1: 'EXTERNAL TRIGGER'
M9301A to M9311A");           // (slot 5 to 2)
Console.WriteLine("\tVSG1 TRIG 7: Uses Bus Segment 1: 'ALC TRIGGER' M9311A to
M9310A");           // (slot 2 to 4)

// Redefine Backplane Triggers for VSG2
M9018A_CHASSIS1.TriggerBus.Connect(4, AgM9018TrigBusEnum.AgM9018TrigBus2To1);
```

```
// 'EXTERNAL TRIGGER' from M9301A to M9311A (slot 9 to 6)
    M9381A_VSG2.System.PXIResources.AddHint("M9301A", "M9311A", "EXTERNAL
TRIGGER",

Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_4);
Console.WriteLine("\tVSG2 TRIG 4: Connect Bus 2 to 1: 'EXTERNAL TRIGGER'
M9301A to M9311A");

M9018A_CHASSIS1.TriggerBus.Connect(5, AgM9018TrigBusEnum.AgM9018TrigBus1To2);
// 'ALC HOLD' from M9311A to M9310A (slot 6 to 8)
M9381A_VSG2.System.PXIResources.AddHint("M9311A", "M9310A", "ALC HOLD",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_5);
Console.WriteLine("\tVSG2 TRIG 5: Connect Bus 1 to 2: 'ALC HOLD' M9311A to
M9310A");

// Use Default Backplane Triggers for VSG3, but have to connect Bus Segment 2
and 3
M9018A_CHASSIS1.TriggerBus.Connect(6, AgM9018TrigBusEnum.AgM9018TrigBus3To2);
// 'EXTERNAL TRIGGER' from M9301A to M9311A (slot 14 to 11)

M9381A_VSG3.System.PXIResources.AddHint("M9301A", "M9311A", "EXTERNAL
TRIGGER",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_6);
Console.WriteLine("\tVSG3 TRIG 6: Connect Bus 3 to 2: 'EXTERNAL TRIGGER'
M9301A to M9311A");

M9018A_CHASSIS1.TriggerBus.Connect(7, AgM9018TrigBusEnum.AgM9018TrigBus2To3);
// 'ALC HOLD' from M9311A to M9310A (slot 11 to 13)
M9381A_VSG3.System.PXIResources.AddHint("M9311A", "M9310A", "ALC HOLD",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_7);
Console.WriteLine("\tVSG3 TRIG 7: Connect Bus 2 to 3: 'ALC HOLD' M9311A to
M9310A");

// Redefine Backplane Triggers for VSG4
// 'EXTERNAL TRIGGER' from M9301A to M9311A (slot 9 to 6)

M9381A_VSG4.System.PXIResources.AddHint("M9301A", "M9311A", "EXTERNAL
TRIGGER",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_4);
Console.WriteLine("\tVSG4 TRIG 4: Uses Bus Segment 3: 'EXTERNAL TRIGGER'
M9301A to M9311A");

// 'ALC HOLD' from M9311A to M9310A (slot 6 to 8)
M9381A_VSG4.System.PXIResources.AddHint("M9311A", "M9310A", "ALC HOLD",
Agilent.AgM938x.Interop.AgM938xPXIResourceTypeEnum.AgM938xPXIResourceTypeTTL_
TRIGGER,
Agilent.AgM938x.Interop.AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_
TRIGGER_5);
Console.WriteLine("\tVSG4 TRIG 5: Uses Bus Segment 3: 'ALC HOLD' M9311A to
M9310A");

// Since VSA Chassis 2 is required for a 4x4 MIMO, set up its triggers as
well.
if ( M9018A_CHASSIS2 != null && M9018A_CHASSIS2.Initialized )
{
// M9391A PXIe VSAs in slots 2-4, 6-8, 12-14, 16-18
// Route PXI TRIG 1 from the MASTER M9214A to the last SLAVE 3 M9214A.
// Route PXI TRIG 2 from SLAVE 1 to MASTER
// Route PXI TRIG 3 from SLAVE 2 to MASTER
// Route PXI TRIG 4 from SLAVE 3 to MASTER
Console.WriteLine("\nRoute MASTER/SLAVE Backplane Triggers for VSAs in 4x4
MIMO Chassis 2:");
// MASTER/SLAVE Backplane Triggers are used to coordinate the capture
alignments
// on the VSAs
M9018A_CHASSIS2.TriggerBus.Connect(1,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus1To2To3);
Console.WriteLine("\tVSA1 to VSA4 TRIG 1: Connect Bus 1 to 2 to 3: MASTER to
SLAVE 3");
M9018A_CHASSIS2.TriggerBus.Connect(2,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus2To1);
Console.WriteLine("\tVSA2 to VSA1 TRIG 2: Connect Bus 2 to 1: SLAVE 1 to
MASTER");
M9018A_CHASSIS2.TriggerBus.Connect(3,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus3To2To1);
Console.WriteLine("\tVSA3 to VSA1 TRIG 3: Connect Bus 3 to 2 to 1: SLAVE 2 to
MASTER");
M9018A_CHASSIS2.TriggerBus.Connect(4,
Agilent.AgM9018.Interop.AgM9018TrigBusEnum.AgM9018TrigBus3To2To1);
Console.WriteLine("\tVSA4 to VSA1 TRIG 4: Connect Bus 3 to 2 to 1: SLAVE 3 to
MASTER");
}
}
```

Step 7 - Set Up the M9381A PXIe VSGs for WLAN Rx Testing

This section covers all of the settings needed to set up the M9381A PXIe VSGs – code snippets are shown for a 2x2 MIMO configuration, but the code needed for 2x2, 3x3, and 4x4 are available in an example file.

The complete example program can be accessed from **C:\Program Files (x86)\Agilent\M9391\Help\Examples\Example__MIMOConsoleApp.cs** and **C:\Program Files (x86)\Agilent\M938x\Help\Examples\Example__MIMOConsoleApp.cs**.

Disable ALC for WLAN waveforms to achieve best Residual EVM

```
M9381A_VSG1.ALC.Enabled = false;
... on VSG 1
M9381A_VSG2.ALC.Enabled = false;
... on VSG 2
```

Enable Pulse Blanking - Achieve Best Off Time Rejection

NOTE The following commands are dependent on whether or not the waveform being used contains pulse blanking markers. Notice the logic is shown for a waveform that contains pulse blanking markers.

```
if ( waveform being played contains pulse blanking markers )
{
Enable Pulse Blanking to achieve Best Off Time Rejection:
M9381A_VSG1.RF.OutputPulseMode =
AgM938xOutputPulseModeEnum.AgM938xOutputPulseModePulseOnWithTrigger;
... on VSG 1
M9381A_VSG2.RF.OutputPulseMode =
AgM938xOutputPulseModeEnum.AgM938xOutputPulseModePulseOnWithTrigger;
... on VSG 2
}
else // Waveform being played DOES NOT CONTAIN pulse blanking
// markers, and must do one of the following:
{
       ( Disable Pulse Blanking )     || ( Define Pulse Blanking Markers
Using Commands from the M938x API )
}
```

Set PLL MODE to Best Wide Offset

```
Set PLL MODE to Best Wide Offset:
M9381A_VSG1.Modules.Synthesizer.PLLMode =
AgM938xSynthesizerPLLModeEnum.AgM938xSynthesizerPLLModeBestWideOffset;
... on VSG 1
M9381A_VSG2.Modules.Synthesizer.PLLMode =
AgM938xSynthesizerPLLModeEnum.AgM938xSynthesizerPLLModeBestWideOffset;
... on VSG 2"
```

Set RF Frequency

```
double Frequency = 5.2e9; // Use 5.2 GHz for testing 802.11ac WLAN
Set RF Frequency:
M9381A_VSG1.RF.Frequency = Frequency;
... on VSG 1
M9381A_VSG2.RF.Frequency = Frequency;
... on VSG 2
```

Set Amplitude (Power_Level)

```
double Level = -2;
Set the amplitude (power/level) of the RF output signal in dBm:
M9381A_VSG1.RF.Level = Level;
... on VSG 1
M9381A_VSG2.RF.Level = Level;
... on VSG 2
```

Enable Modulation

```
Enable Modulation:
M9381A_VSG1.Modulation.Enabled = true;
... on VSG 1
M9381A_VSG2.Modulation.Enabled = true;
... on VSG 2
```

Enable RF Output

```
Enable RF Output:
M9381A_VSG1.RF.OutputEnabled = true;
... on VSG 1
M9381A_VSG2.RF.OutputEnabled = true;
... on VSG 2
```

Step 8 - Start Continuous Waveform Playback without Power Search or IQ DC Calibration

A waveform file can be played on the M9381A PXIe VSGs in Continuous Mode or Sequence Mode.It can also be played with or without performing a Power Search or an IQ DC Calibration.Running a Power Search can improve amplitude accuracy and running an IQ DC Calibration can improve carrier feed-through.

To learn more, refer to the next topic titled *(Optional) Step 8 – Start Continuous or Sequence Waveform Playback with Power Search and IQ DC Cal*

Overview of the Process to Start Continuous Waveform Playback without Power Search or IQ DC Cal

Perform the following steps:

1. Specify a waveform file to upload and play.
2. Upload the *Specified Waveform File.*
3. Set up M9300A PXIe Reference to generate a user-defined Trigger on PXI TRIG 0.
4. Configure all M9381A PXIe VSGs to listen for an External Trigger on PXI TRIG 0.
5. Arm all M9381A PXIe VSGs and prepare for playing the *Specified Waveform File* when an External Trigger is received on PXI TRIG 0.
6. Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to start waveform playback on all of the M9381A PXIe VSGs.

1. Specify a Waveform File to Upload and Play

NOTE **If a Signal Studio waveform file is used, it may require a software license.**

```
string ExamplesFolder = "C:Program Files (x86)AgilentM938xExample Waveforms";
string WaveformFile = "WLAN_11ac_256QAM_80MHz.wfm";
string ArbFileName = ExamplesFolder + WaveformFile;
string mWaveformHandle = "Mod Waveform";
```

2. Upload the Specified Waveform File

```
M9381A_VSG1.Modulation.IQ.UploadArbAgilentFile(mWaveformHandle, ArbFileName);
// ... VSG 1
M9381A_VSG2.Modulation.IQ.UploadArbAgilentFile(mWaveformHandle, ArbFileName);
// ... VSG 2
```

3. Set Up M9300A PXIe Reference to Generate a User-Defined Trigger on PXI TRIG 0

This user-defined trigger (Playback Synchronization Trigger) on PXI TRIG 0 is used to initiate waveform playback. The M9300A PXIe Reference has `ProgrammableOutputTrigger2.Destination`, which specifies the destination of the programmable output trigger. This had a default value `AgM9300TriggerFrontPanelTrigger2` (which was Trig 2 on the M9300A front panel), but its default value is being reassigned in this code so that the M9300A PXIe Reference backplane trigger is on PXI TRIG 0.This `ProgrammableOutputTrigger2.Destination` then comes from `AgM9300TriggerEnum.AgM9300TriggerPXITrigger0`. Later, when trying to play a waveform, the trigger is generated inside the M9300A and comes out on PXI TRIG 0; this triggers the VSGs to start playing the specified waveform.

```
// Define that the 'Playback Synchronization Trigger', from the M9300A PXIe
// Reference, is to come out on PXI TRIG 0 - this backplane trigger
// line will then be used to
// start waveform playback on all M9381A PXIe VSGs.
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.Destination =
AgM9300TriggerEnum.AgM9300TriggerPXITrigger0;
M9300A_REF1.Apply();
```

NOTE

Every time waveform playback is started, the trigger destination on the M9300A PXIe Reference must be configured. This is to avoid a potential conflict with other processes, such as the 89600 VSA Software or other software, which may be sharing the M9300A PXIe Reference and could reset this setting.

4. Configure all M9381A PXIe VSGs to Listen for an External Trigger on PXI TRIG 0

This External Trigger will be generated by the M9300A PXIe Reference and output on the chassis backplane PXI TRIG 0 line and is used to start Waveform Playback on each M9381A PXIe VSG.

```
// Configure the 'External Trigger' on the backplane PXI TRIG 0 line to
deliver a
// 'Synchronization Playback Trigger' from the M9300A PXIe Reference:
M9381A_VSG1.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort);   // ...to VSG 1
M9381A_VSG2.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort);   // ...to VSG 2
```

5. Arm All M9381A PXIe VSGs and Prepare for Playing the Specified Waveform file when an External Trigger is Received on PXI TRIG 0

```
// Play the uploaded waveform file:
M9381A_VSG1.Modulation.PlayArb(mWaveformHandle,
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger);
// ...using VSG 1
M9381A_VSG2.Modulation.PlayArb(mWaveformHandle,
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger);
// ...using VSG 2
```

6. Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to Start Waveform Playback on all of the M9381A PXIe VSGs

This sync pulse is a user-defined trigger (Playback Synchronization Trigger) on PXI TRIG 0 that is used to initiate waveform playback.

```
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.GenerateTrigger();
```

Once a Sync Pluse has been triggered from the M9300A PXIe Reference and a waveform is playing, use 89600 VSA Software to control all M9391A PXIe VSAs and perform Transmitter Tests.

(Optional) Step 8 - Start Continuous or Sequence waveform Playback with Power Search and IQ DC Cal

A waveform file can be played on the M9381A PXIe VSGs in Continuous Mode or Sequence Mode. Improvements of repeatability and accuracy to the output power can be achieved by including an optional Power Search and an optional IQ DC Calibration (that improves carrier feedthrough).

- Continuous Waveform Playback – the same waveform is played repeatedly until stopped.
- Sequence Waveform Playback – the same waveform repeated for a specified number of times and then the *Zeros Waveform File* is played until the sequence is stopped.
- Power Search – is a calibration routine which is used to set an accurate RF level with the ALC off. During a power search cycle, modulation is temporarily switched off, the ALC system is temporarily switched on (just long enough to determine and store the ALC modulator value). This value gives the correct RF level (gain), then the modulation is switched back on. The gain of the RF system

is held constant so that the RF level is accurate even though there is no closed-loop feedback from the ALC. Power Search is useful for running a quick calibration that can help determine a power offset – this power offset can refine the output power level to more closely match the desired output power. Power Search could be thought of as an "open loop" power correction, compared to the use of the ALC which would be a "closed-loop" power correction.

Overview of Starting Continuous Waveform Playback with Power Search and IQ DC Cal

1. Specify a Waveform File to upload and play.
2. Upload the *Specified Waveform File.*
3. Create a *Zeros Waveform File* with the same Sample Rate and RMS Value as the *Specified Waveform File*. When creating the *Zeros Waveform File*, recover the Sample Rate and RMS Value from the *Specified Waveform File* that is loaded (with the reference name that is stored in the string variable mWaveformHandle).
4. Perform Power Search to obtain power offsets to the *Specified Waveform File*, then play the *Specified Waveform File*with offsets in Continuous or Sequence Mode.
   a. Play the *Zeros Waveform File* in Immediate Trigger mode.
   b. Turn on RF Blanking - to prevent unwanted signals from coming out during Power Search.
   c. Perform Power Search; obtain offsets to the *Specified Waveform File.*
   d. (Optional) Cache the Power Search offset data.
   e. (Optional) Perform IQ DC Calibration to improve carrier feedthrough.
   f. Stop the *Zeros Waveform File* playback.
5. Apply the Power Search offsets to the *Specified Waveform File.*
6. Continuously play the *Specified Waveform File* with Power Search Offsets applied.
7. Set up an M9300A PXIe Reference to generate a user-defined trigger on PXI TRIG 0.
8. Configure all M9381A PXIe VSGs to listen for an External Trigger on PXI TRIG 0.
9. Arm all M9381A PXIe VSGs and Prepare for continuously playing the *Specified Waveform File* in External Trigger mode, that have had the Power Search offsets applied when an External Trigger is Received on PXI TRIG 0.
10. Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to start waveform playback on all of the M9381A PXIe VSGs.

1 Specify a Waveform File to Upload and Play

NOTE **If a Signal Studio waveform file is used, it may require a software license.**

```
string ExamplesFolder = "C:Program Files (x86)AgilentM938xExample Waveforms";
string WaveformFile = "WLAN_11ac_256QAM_80MHz.wfm";
string ArbFileName = ExamplesFolder + WaveformFile;
string mWaveformHandle = "Mod Waveform";
```

2 Upload the Specified Waveform File

```
M9381A_VSG1.Modulation.IQ.UploadArbAgilentFile(mWaveformHandle, ArbFileName);
// ... VSG 1
M9381A_VSG2.Modulation.IQ.UploadArbAgilentFile(mWaveformHandle, ArbFileName);
// ... VSG 2
```

3. Create a Zeros Waveform File with Same Sample Rate and RMS Value as the Specified Waveform File

This Zeros Waveform File is for use with Power Search and for ending a Sequence Waveform Playback.

When creating the Zeros Waveform File, recover the Sample Rate and RMS Value from the *Specified Waveform File* that is loaded (with the reference name that is stored in the string variable `mWaveformHandle`); in order to reuse the Power Search result, the *Specified Waveform File* and the Power Search result must have the same RMS Value.

The `ArbInformation` method is used to get the Sample Rate, RMS Value, and other parameters from the desired waveform:

```
// Create a zeros waveform file with zeros to end a sequence waveform
playback
// Create file w/ zeros at same Sample Rate and RMS Values as Specified
Waveform File
double[] ArbData = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
double sampleRateVSG1 = 0;
double sampleRateVSG2 = 0;
double rmsValueVSG1 = 0;
double rmsValueVSG2 = 0;
double scaleFactor = 0;
AgM938xMarkerEnum alcMarker = AgM938xMarkerEnum.AgM938xMarkerNone;
AgM938xMarkerEnum blankingMarker = AgM938xMarkerEnum.AgM938xMarkerNone;

// Remove the sequence if it already exists
M9381A_VSG1.Modulation.Sequence.Remove("Seq");
M9381A_VSG2.Modulation.Sequence.Remove("Seq");
```

```
// Remove the zeros waveform if it already exists
M9381A_VSG1.Modulation.IQ.RemoveArb("zeros");
M9381A_VSG2.Modulation.IQ.RemoveArb("zeros");

// Recover the Sample Rate and RMS Value from the waveform
```

4. Perform Power Search to Obtain Offsets to the Specified Waveform File

Performing a Power Search improves amplitude accuracy. The *Specified Waveform File* can then be played with offsets applied in continuous or sequence mode.

Play the Zeros Waveform File in Immediate Trigger Mode

```
M9381A_VSG1.Modulation.PlayArb("zeros",
AgM938xStartEventEnum.AgM938xStartEventImmediate);
M9381A_VSG2.Modulation.PlayArb("zeros",
AgM938xStartEventEnum.AgM938xStartEventImmediate);

// Allow some time for all waveforms to start playing
System.Threading.Thread.Sleep(100);
```

Turn on RF Blanking to prevent unwanted signals from coming out during Power Search.

```
M9381A_VSG1.Calibration.PowerSearch.BlankRFDuringSearch = true;
M9381A_VSG2.Calibration.PowerSearch.BlankRFDuringSearch = true;
```

Perform Power Search and Obtain Offsets to the Specified Waveform File

```
double powOffsetVSG1;
double scaleOffsetVSG1;
double powOffsetVSG2;
double scaleOffsetVSG2;
M9381A_VSG1.Calibration.PowerSearch.DoPowerSearch(ref powOffsetVSG1,
ref scaleOffsetVSG1);
M9381A_VSG2.Calibration.PowerSearch.DoPowerSearch(ref powOffsetVSG2,
ref scaleOffsetVSG2);
```

(Optional) Cache the Power Search Data

```
// Store these values for future use - they are not used in this routine.
double cachepowOffsetVSG1 = powOffsetVSG1;
double cachescaleOffsetVSG1 = scaleOffsetVSG1;
double cachepowOffsetVSG2 = powOffsetVSG2;
double cachescaleOffsetVSG2 = scaleOffsetVSG2;
```

(Optional) Perform IQ DC Calibration – Improves Carrier Feedthrough

```
M9381A_VSG1.Calibration.IQAlignment.AlignIQAtDC();
```

```
M9381A_VSG2.Calibration.IQAlignment.AlignIQAtDC();
```

Stop the Zeros Waveform File playback

```
M9381A_VSG1.Modulation.Stop();
M9381A_VSG2.Modulation.Stop();

// Allow some time for all waveforms to stop playing
System.Threading.Thread.Sleep(100);
```

5. Apply the Power Search Offsets to the Specified Waveform File

```
// Must apply the result we measured in power search previously when we
replay
M9381A_VSG1.Calibration.PowerSearch.UsePowerSearchResult(powOffsetVSG1,
scaleOffsetVSG1);
M9381A_VSG2.Calibration.PowerSearch.UsePowerSearchResult(powOffsetVSG2,
scaleOffsetVSG2);
```

6. Continuously Play the Specified Waveform File with Power Search Offsets Applied

Set Up M9300A PXIe Reference to Generate a User-Defined Trigger on PXI TRIG 0

```
// Define that the 'Playback Synchronization Trigger', from the
// M9300A PXIe Reference, is to come out on PXI TRIG 0 - this backplane
trigger
// line will then be used to start waveform playback on all M9381A PXIe VSGs.
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.Destination =
AgM9300TriggerEnum.AgM9300TriggerPXITrigger0; M9300A_REF1.Apply();
```

NOTE

Every time waveform playback is started, the trigger destination on the M9300A PXIe Reference must be configured. This is to avoid a potential conflict with other processes, such as the 89600 VSA Software or other software, which may be sharing the M9300A PXIe Reference and could reset this setting.

Configure all M9381A PXIe VSGs to Listen for an External Trigger on PXI TRIG 0

This External Trigger will be generated by the M9300A PXIe Reference and output on the chassis backplane PXI TRIG 0 line and is used to start Waveform Playback on each M9381A PXIe VSG.

```
// Configure the 'External Trigger' on the backplane PXI TRIG 0 line to
deliver // a 'Synchronization Playback Trigger' from the M9300A PXIe
Reference:
M9381A_VSG1.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort);
// ...to VSG 1
M9381A_VSG2.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort);
// ...to VSG 2
```

Arm All M9381A PXIe VSGs and Prepare for Continuosly Playing the Specified Waveform File when an External Trigger is Received on PXI TRIG 0

```
M9381A_VSG1.Modulation.PlayArb(mWaveformHandle,
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger);
// ...using VSG 1
M9381A_VSG2.Modulation.PlayArb(mWaveformHandle,
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger);
// ...using VSG 2
```

Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to Start Waveform Playback on all of the M9381A PXIe VSGs

```
// Generate a Sync Pulse on the M9300A PXIe Reference, on PXI TRIG 0,
// and use it to initiate waveform playback.
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.GenerateTrigger();
```

Once a Sync Pulse has been triggered from the M9300A PXIe Reference and a waveform is playing, use 89600 VSA Software to control all M9391A PXIe VSAs and perform Transmitter Tests.

Overview of Starting Sequence Waveform Playback with Power Search and IQ DC Cal

1. Specify a Waveform File to upload and play.
2. Upload the *Specified Waveform File.*
3. Create a *Zeros Waveform File* with the same Sample Rate and RMS Value as the *Specified Waveform File*. When creating the *Zeros Waveform File*, recover the Sample Rate and RMS Value from the *Specified Waveform File* that is loaded (with the reference name that is stored in the string variable `mWaveformHandle`).

4. Perform Power Search to obtain power offsets to the *Specified Waveform File,* then play the *Specified Waveform File* with offsets in Sequence Mode.
    a. Play the *Zeros Waveform File* in Immediate Trigger mode.
    b. Turn on RF Blanking to prevent unwanted signals from coming out during Power Search.
    c. Perform Power Search; obtain offsets to the *Specified Waveform File.*
    d. (Optional) Cache the Power Search offset data.
    e. (Optional) Perform IQ DC Calibration to improve carrier feedthrough.
    f. Stop the *Zeros Waveform File* playback.
5. Apply the Power Search offsets to the *Specified Waveform File.*

NOTE **The*Zeros Waveform File*creation was described in the above section.**

6. Create a *Sequence* that is made up of two steps. The first step in the sequence is the *Specified Waveform File* (that is played a specified number of times); the second step in the sequence is the *Zeros Waveform File* (that essentially turns off the RF output). This *Zeros Waveform File* is played until a Sequence.End command is issued.
Sequence = Specified Waveform File repeated X times + Zeros Waveform File repeated until end
    – Create a *Sequence.*
    – Set the RMS power in the *Sequence* definition.
    – Add the *Specified Waveform File* as Step 1 in the *Sequence.*
    – Set the number of times to repeat the *Specified Waveform File* in the *Sequence.*
    – Add the *Zeros Waveform File* as Step 2 in the *Sequence.*
    – Set the number of times to repeat the *Zeros Waveform File* in the *Sequence* to continue forever until a "Stop" command is issued.

    a. Set up an M9300A PXIe Reference to Generate a User-Defined Trigger on PXI TRIG 0.
    b. Configure all M9381A PXIe VSGs to Listen for an External Trigger on PXI TRIG 0.
    c. Arm all M9381A PXIe VSGs and Prepare for Playing a Sequence in External Trigger mode that have had the Power Search offsets applied when an External Trigger is Received on PXI TRIG 0.
    d. Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to start waveform playback on all of the M9381A PXIe VSGs.

6. Create a Sequence that is Made Up of Two Steps

NOTE The RMS value must be set during the*Sequence*definition or the power level will be in error.

```
// Create the sequence
M9381A_VSG1.Modulation.Sequence.Create("Seq");
M9381A_VSG2.Modulation.Sequence.Create("Seq");

// RMS power must be set in the sequence definition.
M9381A_VSG1.Modulation.RmsPower = rmsValueVSG1;
M9381A_VSG2.Modulation.RmsPower = rmsValueVSG2;

M9381A_VSG1.Modulation.Sequence.AddStep("Step1");
M9381A_VSG2.Modulation.Sequence.AddStep("Step1");

// Add a segment with the waveform we want to play
M9381A_VSG1.Modulation.Sequence.AddSegment(mWaveformHandle);
M9381A_VSG2.Modulation.Sequence.AddSegment(mWaveformHandle);

// Specify how many times to play this waveform
M9381A_VSG1.Modulation.Sequence.StepRepetitions = (int)
repetitionsControl.Value;
M9381A_VSG2.Modulation.Sequence.StepRepetitions = (int)
repetitionsControl.Value;

M9381A_VSG1.Modulation.Sequence.AddStep("Step2");
M9381A_VSG2.Modulation.Sequence.AddStep("Step2");

// After the Specified Waveform File, play a waveform that is all 0's
// to turn off RF output
M9381A_VSG1.Modulation.Sequence.AddSegment("zeros");
M9381A_VSG2.Modulation.Sequence.AddSegment("zeros");

M9381A_VSG1.Modulation.Sequence.StepRepetitions = 1;
M9381A_VSG2.Modulation.Sequence.StepRepetitions = 1;

// Play Step 2, with the zeros, forever, until the "Stop" command is issued
M9381A_VSG1.Modulation.Sequence.NextStep("Step2");
M9381A_VSG2.Modulation.Sequence.NextStep("Step2");

M9381A_VSG1.Modulation.Sequence.End();
M9381A_VSG2.Modulation.Sequence.End();

// Scale factor must be set in order to avoid modulation quality issues.
// Scale factor is not automatically set from the waveform settings
// for a sequence.
M9381A_VSG1.Modulation.Scale = scaleFactor;
M9381A_VSG2.Modulation.Scale = scaleFactor;
```

Set Up M9300A PXIe Reference to Generate a User-Defined Trigger on PXI TRIG 00

```
// Define that the 'Playback Synchronization Trigger', from the
// M9300A PXIe Reference, is to come out on PXI TRIG 0 - this backplane
trigger
// line will then be used to start waveform playback on all M9381A PXIe VSGs.
```

```
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.Destination =
AgM9300TriggerEnum.AgM9300TriggerPXITrigger0;
M9300A_REF1.Apply();
```

NOTE

Every time waveform playback is started, the trigger destination on the M9300A PXIe Reference must be configured. This is to avoid a potential conflict with other processes, such as the 89600 VSA Software or other software, which may be sharing the M9300A PXIe Reference and could reset this setting.

Configure all M9381A PXIe VSGs to Listen for an External Trigger on PXI TRIG 00

This External Trigger will be generated by the M9300A PXIe Reference and output on the chassis backplane PXI TRIG 0 line and is used to start Sequence Playback on each M9381A PXIe VSG.

```
// Configure the 'External Trigger' on the backplane PXI TRIG 0 line to
deliver
// a 'Synchronization Playback Trigger' from the M9300A PXIe Reference:
M9381A_VSG1.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort);   // ...to VSG 1

M9381A_VSG2.Triggers.ExternalTrigger.Configure(true, 0.01, 0.5,
AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
AgM938xTriggerEnum.AgM938xTriggerPXITrigger0,
AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, 10000,
AgM938xTimeoutModeEnum.AgM938xTimeoutModeTimeoutAbort); // ...to VSG 2
```

Arm All M9381A PXIe VSGs and Prepare for Playing the Sequence when an External Trigger is Received on PXI TRIG 0

```
M9381A_VSG1.Modulation.Sequence.Play("Seq",
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger); // ...using VSG 1
M9381A_VSG2.Modulation.Sequence.Play("Seq",
AgM938xStartEventEnum.AgM938xStartEventExternalTrigger); // ...using VSG 2
System.Threading.Thread.Sleep(100);
```

Generate a Sync Pulse from the M9300A PXIe Reference on PXI TRIG 0 to Start Sequence Playback on all of the M9381A PXIe VSGs

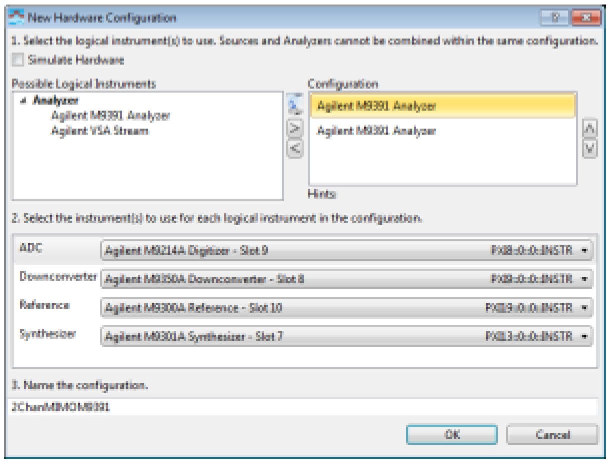
```
// Generate a Sync Pulse on the M9300A PXIe Reference, on PXI TRIG 0,
// and use it to initiate sequence playback.
M9300A_REF1.ReferenceBase2.ProgrammableOutputTrigger2.GenerateTrigger();
```

Once a Sync Pluse has been triggered from the M9300A PXIe Reference and a sequence is playing, use 89600 VSA Software to control all M9391A PXIe VSAs and perform Transmitter Tests.

Step 9 - Create an N-Channel Analyzer Hardware Configuration with 89600 VSA Software

Create a 2-Channel Analyzer of M9391A PXIe VSAs

Channel 1 Configuration



1. Select the logical instrument(s) to use. Sources and Analyzers cannot be combined within the same configuration.

Simulate Hardware

Possible Logical Instruments
- Analyzer
  - Agilent M9391 Analyzer
  - Agilent VSA Stream

Configuration
- Agilent M9391 Analyzer
- Agilent M9391 Analyzer

Hints:

2. Select the instrument(s) to use for each logical instrument in the configuration.

ADC	Agilent M9214A Digitizer - Slot 9	PXI8::0::0::INSTR
Downconverter	Agilent M9350A Downconverter - Slot 8	PXI9::0::0::INSTR
Reference	Agilent M9300A Reference - Slot 10	PXI19::0::0::INSTR
Synthesizer	Agilent M9301A Synthesizer - Slot 7	PXI13::0::0::INSTR

3. Name the configuration.

2ChanMIMOM9391

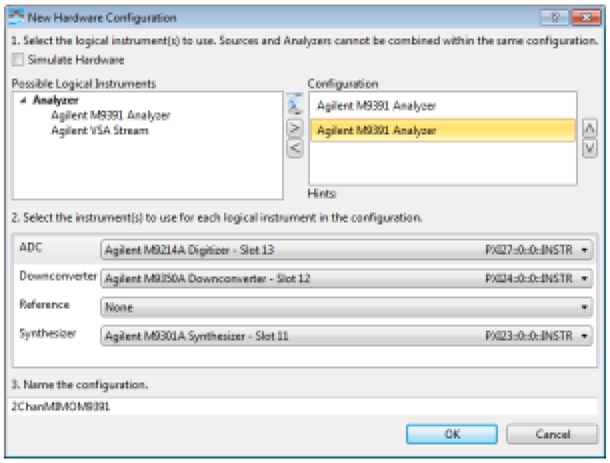
1. Select Start > All Programs > Agilent/Keysight > **Agilent/Keysight 89600 VSA 16.2** or newer.
2. Select Utilities > Hardware > Configurations. Select (green plus icon) **Add new configuration**.
3. Add the **Agilent/Keysight M9391 Analyzer** two times with the **> arrow** button – this will create a two channel analyzer.
4. Select the **first** "Agilent/Keysight M9391 Analyzer" under the Configuration group.
5. Select the Synthesizer, Reference, Downconverter, and ADC/Digitizer that form the **Channel 1** configuration; **include the M9300A Reference module in Channel 1, but not in Channel 2**.

   If you are using the "Recommended" configuration for a 2x2 MIMO Configuration in One M9018A PXIe Chassis, the Synthesizer, Reference, Downconverter, and ADC/Digitizer will look as shown in the above screen capture.

Note the order in which the Agilent/Keysight M9391 Analyzers (channels) are added to the hardware configuration. In this example, it is the first (top) configuration. This information will be needed later to determine which channel is used to receive an external, software or magnitude trigger, and it is also needed for trigger allocation on the PXI backplane.

Channel 2 Configuration

1. Select the **second** "Agilent/Keysight M9391 Analyzer" under the Configuration group.
2. Select the Synthesizer, Reference, Downconverter, and ADC/Digitizer that form the **Channel 2** configuration – **do not include the M9300A Reference in this Channel 2 configuration**.
3. **Connect** to the new hardware configuration.
4. Exit the 89600 VSA Software to ensure all settings are saved.



1. Select the logical instrument(s) to use. Sources and Analyzers cannot be combined within the same configuration.

Simulate Hardware: [ ]

Possible Logical Instruments:
- Analyzer
  - Agilent M9391 Analyzer
  - Agilent VSA Stream

Configuration:
- Agilent M9391 Analyzer
- Agilent M9391 Analyzer

Hints:

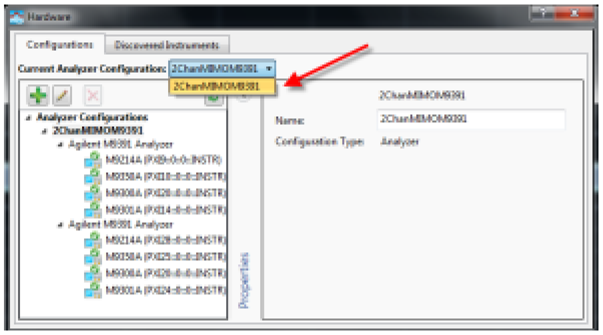
2. Select the instrument(s) to use for each logical instrument in the configuration.

ADC	Agilent M9214A Digitizer - Slot 13	PXI27::0::0::INSTR
Downconverter	Agilent M9350A Downconverter - Slot 12	PXI24::0::0::INSTR
Reference	None	
Synthesizer	Agilent M9301A Synthesizer - Slot 11	PXI23::0::0::INSTR

3. Name the configuration.

2ChanMIMOM9391

After completing the previous steps, the system should be properly configured and is ready to make measurements.



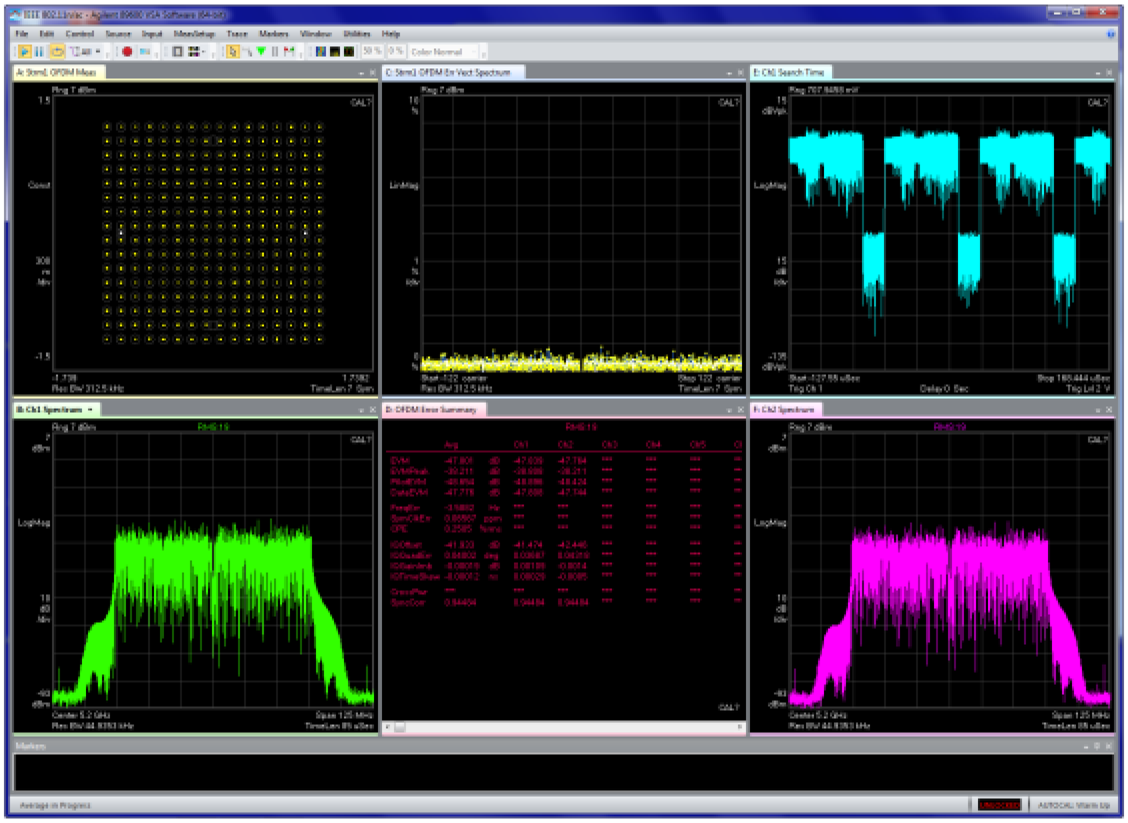
Current Analyzer Configuration: 2ChanMIMOM9391

- Analyzer Configurations
  - 2ChanMIMOM9391
    - Agilent M9391 Analyzer
      - M9214A (PXI9::0::0::INSTR)
      - M9350A (PXI10::0::0::INSTR)
      - M9308A (PXI28::0::0::INSTR)
      - M9301A (PXI14::0::0::INSTR)
    - Agilent M9391 Analyzer
      - M9214A (PXI28::0::0::INSTR)
      - M9350A (PXI25::0::0::INSTR)
      - M9308A (PXI28::0::0::INSTR)
      - M9301A (PXI24::0::0::INSTR)

2ChanMIMOM9391
Name: 2ChanMIMOM9391
Configuration Type: Analyzer

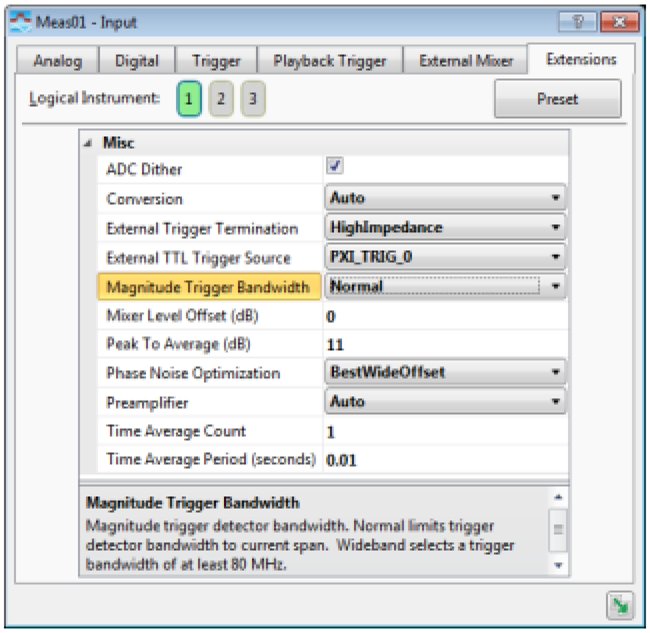
Step 10 – Start 89600 VSA Software to Analyze M9381A PXIe VSGs Waveform Output

1. Click **Start > All Programs > Agilent/Keysight 89600 Software 16.2 or newer > Agilent/Keysight 89600 VSA 16.2 (64-bit)**.
2. Click **File > Recall / Recall Setup** and select *80211ac_MCS8_1SS_80MHzBW.setx*. A display similar to the following should appear.



Step 11 – Optimize 89600 VSA Settings for WLAN Demodulation

The Input / Extensions menu on the 89600 VSA contains several settings that are needed for optimizing WLAN EVM performance.

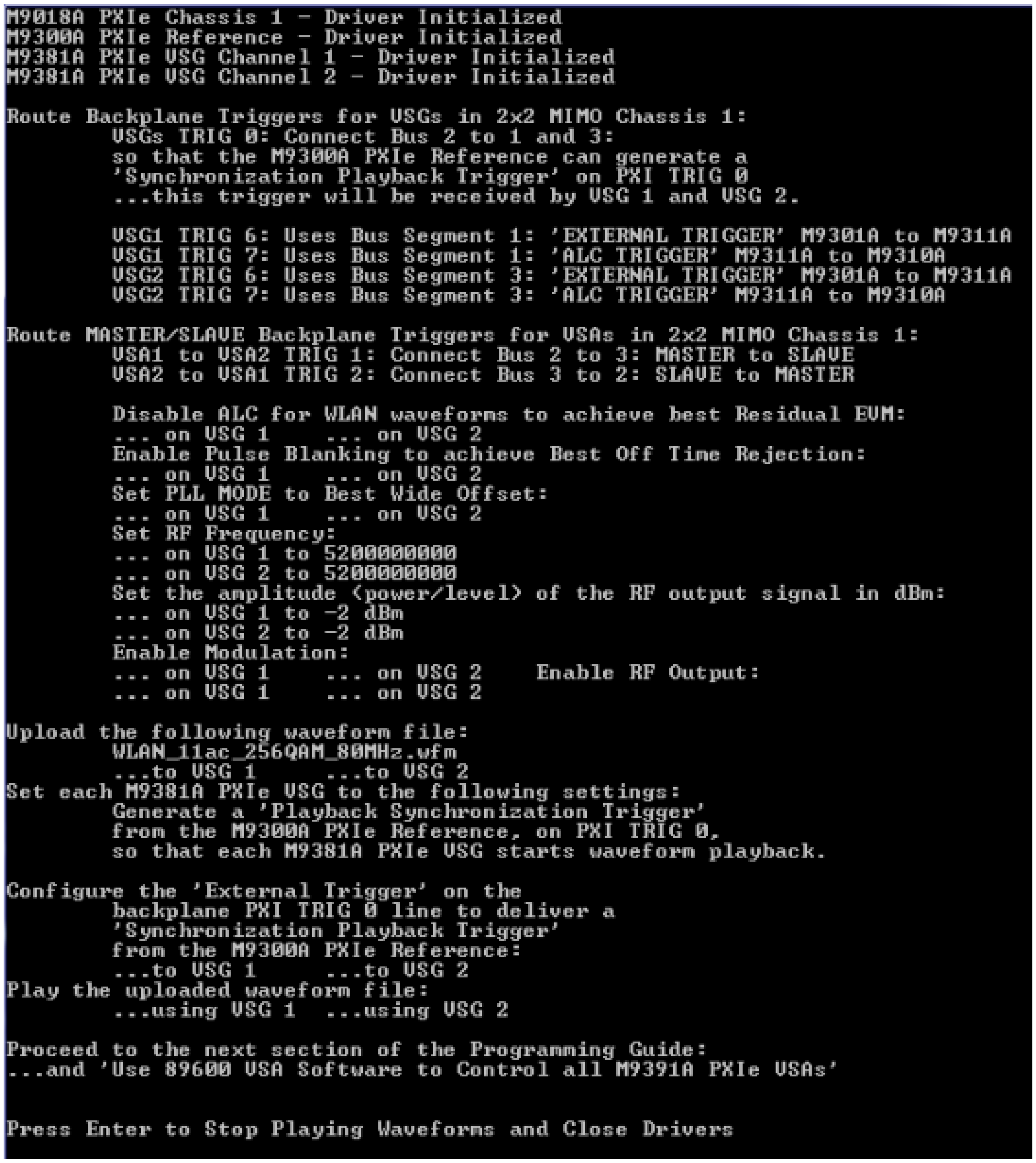


ADC Dither	✓
Conversion	Auto
External Trigger Termination	HighImpedance
External TTL Trigger Source	PXI_TRIG_0
Magnitude Trigger Bandwidth	Normal
Mixer Level Offset (dB)	0
Peak To Average (dB)	11
Phase Noise Optimization	BestWideOffset
Preamplifier	Auto
Time Average Count	1
Time Average Period (seconds)	0.01

Magnitude Trigger Bandwidth
Magnitude trigger detector bandwidth. Normal limits trigger detector bandwidth to current span. Wideband selects a trigger bandwidth of at least 80 MHz.

- Phase Noise Optimization should always be set to BestWideOffset for WLAN demodulation.
- Peak to Average (dB) should be set as close as possible to the actual PAR of the signal, to optimize mixer drive level and IF attenuation. This should be used in conjunction with the Range setting in Analog settings.
- Additionally, the Mixer Level Offset (dB) can be used to provide additional adjustment to mixer drive level.
- Conversion mode should always be set to Auto or SingleHighSide or SingleLowSide for best WLAN EVM performance.

NOTE The peak to average setting must be set for EACH channel, while the PLL mode is "shared" and will automatically be set on all channels when you change it for one.



```
M9018A PXIe Chassis 1 - Driver Initialized
M9300A PXIe Reference - Driver Initialized
M9381A PXIe VSG Channel 1 - Driver Initialized
M9381A PXIe VSG Channel 2 - Driver Initialized

Route Backplane Triggers for VSGs in 2x2 MIMO Chassis 1:
        VSGs TRIG 0: Connect Bus 2 to 1 and 3:
        so that the M9300A PXIe Reference can generate a
        'Synchronization Playback Trigger' on PXI TRIG 0
        ...this trigger will be received by VSG 1 and VSG 2.

        VSG1 TRIG 6: Uses Bus Segment 1: 'EXTERNAL TRIGGER' M9301A to M9311A
        VSG1 TRIG 7: Uses Bus Segment 1: 'ALC TRIGGER' M9311A to M9310A
        VSG2 TRIG 6: Uses Bus Segment 3: 'EXTERNAL TRIGGER' M9301A to M9311A
        VSG2 TRIG 7: Uses Bus Segment 3: 'ALC TRIGGER' M9311A to M9310A

Route MASTER/SLAVE Backplane Triggers for VSAs in 2x2 MIMO Chassis 1:
        VSA1 to VSA2 TRIG 1: Connect Bus 2 to 3: MASTER to SLAVE
        VSA2 to VSA1 TRIG 2: Connect Bus 3 to 2: SLAVE to MASTER

        Disable ALC for WLAN waveforms to achieve best Residual EVM:
        ... on VSG 1    ... on VSG 2
        Enable Pulse Blanking to achieve Best Off Time Rejection:
        ... on VSG 1    ... on VSG 2
        Set PLL MODE to Best Wide Offset:
        ... on VSG 1    ... on VSG 2
        Set RF Frequency:
        ... on VSG 1 to 5200000000
        ... on VSG 2 to 5200000000
        Set the amplitude (power/level) of the RF output signal in dBm:
        ... on VSG 1 to -2 dBm
        ... on VSG 2 to -2 dBm
        Enable Modulation:
        ... on VSG 1    ... on VSG 2    Enable RF Output:
        ... on VSG 1    ... on VSG 2

Upload the following waveform file:
        WLAN_11ac_256QAM_80MHz.wfm
        ...to VSG 1     ...to VSG 2
Set each M9381A PXIe VSG to the following settings:
        Generate a 'Playback Synchronization Trigger'
        from the M9300A PXIe Reference, on PXI TRIG 0,
        so that each M9381A PXIe VSG starts waveform playback.

Configure the 'External Trigger' on the
        backplane PXI TRIG 0 line to deliver a
        'Synchronization Playback Trigger'
        from the M9300A PXIe Reference:
        ...to VSG 1     ...to VSG 2
Play the uploaded waveform file:
        ...using VSG 1  ...using VSG 2

Proceed to the next section of the Programming Guide:
...and 'Use 89600 VSA Software to Control all M9391A PXIe VSAs'


Press Enter to Stop Playing Waveforms and Close Drivers
```

Once all measurements have been made with the 89600 VSA software, you can close all driver instances by pressing the Enter key on the Console Application.

Using Shared LO for Phase-Coherent Signal Generation and Signal Acquisition

This section of the Programming Guide focuses on creating programs for multichannel (MIMO) operations for M9381A PXIe VSGs and M9391A PXIe VSAs when using a shared local oscillator (LO) for each set of transmitters (M9381A) and receivers (M9391A).

Using a Shared LO

Sharing the LO (M9301A Synthesizer) between sets of M9381A or M9391A instruments enables you to build multi-channel phase-coherent measurement systems.

Multi-channel signal generation and signal acquisition has conventionally been performed using independent LOs for each set of M9381A and M9391A instruments. Such setups yield less than 1 degree phase jitter. This, however, does not serve the purpose of several applications in the wireless communication domain that require phase stability and coherence, for example, phased array and beam steering applications. Since using separate LOs in a multi-channel environment can substantially contribute to phase incoherence, more robust techniques must be adopted for these applications. Sharing the M9301A Synthesizer between sets of M9381A or M9391A instruments minimizes phase drifts and ensures constant phase relationships between the channels. Phase noise affects all the channels of such a test system in a consistent manner, thus, its effect is considerably minimized.

Implementation of Shared LO

You can use the four output ports of an M9301A Synthesizer to provide an LO signal for up to four M9381 VSGs or M9391 VSAs in a single chassis. For more than four instruments, you can use a single output port routed through a V2802A LO distribution unit to split the signal. For more information, refer to *Sharing the M9301A Synthesizer's LO* topic in the *Keysight M9391A and M9381A Startup Guide*.

Sharing the M9300A Frequency Reference between the drivers is different from sharing the M9301A Synthesizer Module. When the M9300A Frequency Reference is shared by multiple instances of M9381A or M9391A drivers, the latest instance of the driver, by default, takes full control of the M9300A Frequency Reference. In case of shared M9301A Synthesizer Module, a single instance of the M9381A or M9391A driver is designated as Master while the other drivers are specified as Slaves. Only the Master driver can tune the shared LO; the slaves have read-only access to the hardware registers to query the state of the hardware. Instruments designated as using a shared synthesizer in a slave role periodically read back the current state of the synthesizer module:

- Whenever any changes are applied to the hardware via a call to Apply()
- Once every second via a polling mechanism

The designation of Master and Slave drivers should occur at the time of initialization and cannot be changed during program execution. For more information, you may refer to the *Initialize* Method topic in the M938x/M9391A IVI driver documentation.

The Initialization Settings for Shared LO (page 98) section explains all the startup options that need to be set for sharing the M9301A synthesizer between M938x or M9391 drivers.

Prerequisites for Using a Shared LO

Before you use a shared LO for phase coherent multichannel operation, ensure that you configure the following required hardware options:

- M9311A-012 for M9381A VSG. Look for the option on the M9311A Modulator.
- M9350A-012 for M9391A VSA. Look for the option on the M9214A Digitizer.

LO Level Field Alignment

The LO Level Field Alignment is performed to offset the effects of losses arising from non-standard cabling and temperature variation. This alignment is mandatory when using an LO distribution unit, such as the V2801A LO Distribution Network. For both M9381A VSG and M9391A VSA, the LO Level Field Alignment must be performed on all channels and must be performed on the Master first.

Cabling of Instruments When Sharing Local Oscillator

For an overview of all the hardware modules and their cable connections and configurations to be performed for Multi-Channel operation using a shared LO, refer to the *Sharing the M9301A Synthesizer's LO* topic in the *Keysight M9391A and M9381A Startup Guide*. Complete all the hardware configuration steps before trying to programmatically control M9381A PXIe VSGs and M9391A VSAs with their respective IVI drivers.

Initialization Settings for Shared LO

Most of the shared LO functionality is configured during the driver initialization phase. You must configure the following options in the "`DriverSetup=`" section of the `OptionString`.

Option Name	Description	Default
ShareSynthesizerVisaSession	Set this option to true for all the M938x or M9391	False

	drivers that need to share a synthesizer module. An instrument that tries to connect to a synthesizer module that is already a part of a session without setting this option to true encounters an error and initialization fails.	
SynthesizerRole	By default, an instrument driver in the Shared LO model assumes the role of a Master. Declare one instrument as Master and all the other instruments as slaves by setting the option `SynthesizerRole=slave`. NOTE **1. After initialization, you cannot change the role of a driver within the session.** **2. If you set an instrument as slave without having the option M9311A-012 present for VSG or option M9350A-012 present for VSA, an error is generated when the `Apply()` method is called.**	Master
SynthesizerOutputPort	The M9301A synthesizer has four output ports. Each output port has been individually calibrated, and you can declare which output port the instrument is connected to for the highest accuracy. The output port is declared by setting the option `SynthesizerOutputPort=`*XX* where XX is the name of the port (1a, 1b, 2a, or 2b). You can change the value for this option within a session.	1a

For more information, refer to the *Initialize Method* topic in the M938x/M9391A IVI driver documentation.

Initialization Steps

Perform the following steps to configure the shared LO functionality in the "`DriverSetup=`" section of the `OptionString`:

1. Set the `ShareSynthesizerVisaSession` option to true or 1.
2. By default, `SynthesizerRole` is set to Master. So, for the Master driver, you need not configure any settings. For all the slave drivers, set `SynthesizerRole=slave`.
3. By default, `SynthesizerOutputPort` is set to port 1A. Any instrument connected to a different port must set `SynthesizerOutputPort=XX` where XX is the name of the port (1a, 1b, 2a, or 2b).

The following code snippet depicts a typical driver construction when using shared LO for a set of M9381A drivers.

```
string masterOptions = "DriverSetup=ShareSynthesizerVisaSession=true";
string slaveOptions = "DriverSetup=ShareSynthesizerVisaSession=true,
SynthesizerRole=slave,SynthesizerOutputPort=1b";


IAgM938xEx2 driver1 = new AgM938xClass();
IAgM938xEx2 driver2 = new AgM938xClass();


driver1.Initialize( resources, idquery, reset, masterOptions);
driver2.Initialize( resources, idquery, reset, slaveOptions);
```

LO Level Field Alignment

When using an LO Distribution unit, set the `ExtLoDistributionUnit` option to 1. This informs the driver to prepare itself for use with the unit by loading the LO Level field alignment data. For more information, see Appendix - Using LO Distribution Network in Multi-Channel Systems (page 135).

Example Programs

In this section, you will learn how to write IVI-COM Console Applications that make use of the multi-channel capability of M9381A and M9391A using shared LO.

For M9381A, the example programs may be found in:**C:\Program Files (x86)\IVI Foundation\IVI\Drivers\AgM938x\Examples.**For M9391A, example programs may be found in: **C:\Program Files (x86)\IVI Foundation\IVI\Drivers\AgM9391\Examples**

To understand and write these example programs, you require knowledge of the following:

- Visual Studio 2010 with the .NET framework
- Programming syntax for Visual C#
- Multi-channel capability of M9381A VSG and M9391A VSA using shared LO

To compile your program, use the .NET Framework library and produce an Assembly .exe file.

Example Program 5: Transmitter program to demonstrate how to perform a multi-channel synchronous modulated signal generation using M9381A IVI commands. Use of shared M9301A Synthesizer is also demonstrated.

Example Program 6: Receiver program to demonstrate how to perform a multi-channel IQ acquisition using M9391A IVI commands. Use of shared M9301A Synthesizer is also demonstrated.

These example programs demonstrate a two channel system. One channel is the Master, the other a Slave. The example programs can be extended to add more Slave channels.

Assumptions for Example Programs

The example programs make the following assumptions:

- There are two predefined IVI connection aliases for both M9381A and M9391A, "M9381A-Master" and "M9381A-Slave", and "M9391A-Master" and "M9391A-Slave", respectively.
- At least one of the channels includes the M9300 Reference module and it is located in the chassis System Timing Slot. For simplicity, the Master is assumed to include an M9300 Reference module, while the Slave is assumed to not include an M9300 Reference module. For most situations, this is adequate and recommended to avoid additional considerations of Reference sharing between instances, for example, 1 second latency for settings to propagate between shared Reference instances.

  Information for scenarios in which slave also shares the M9300A Reference has also been provided in the programming example.

For the use of a shared M9301A Synthesizer module for phase-coherent multi-channel systems, the following are the assumptions:

- Hardware option M9311A-012 is available for M9381A and M9350A-012 for M9391A.
- The Master channel uses Output Port 1A of the M9301A Synthesizer module, and the Slave channel uses Output Port 1B.

Example Program 5 - Pseudo-Code

This section presents the flow of the program. Code snippets are provided in the next section.

1. Create a Visual C# Console Application.
2. Add References.
    a. Add reference to IVI AgM938x 2.0 Type Library.
3. Add Using Statements.
4. Prepare for creation and initialization of drivers.

   a. Declare master and slave drivers as instances of IAgM938xEx2.
   b. Specify the mode of operation (simulation or non-simulation) and the resource descriptor.
   c. Set the initialization properties for the shared synthesizer.

5. Create and initialize driver instances.
   a. Create Master and Slave drivers.
   b. Initialize the Master driver.
   c. Clear startup messages and warnings, if any, on Master.
   d. Print Ivi Driver Identity and Shared Synthesizer properties for Master.
   e. Initialize the Slave driver.
   f. Clear startup messages and warnings, if any, on Slave.
   g. Print Ivi Driver Identity and Shared Synthesizer properties for Slave.
6. Enable an external reference, if planned.
7. Multichannel Sync Setup
   a. Set up the Reference module to drive the 10 MHz backplane clock.
   b. Initialize synchronization clocks for all source channels.
   c. Set up the master/slave roles. Exactly one system must be the SystemMaster.
   d. On the Master channel, configure the GroupSynchronizationSignal which is used to trigger slave channels.
   e. On each Slave channel set the SlaveSynchronizationSignal handshaking line to a unique PXI backplane trigger line.
   f. On each Slave channel, set the GroupSynchronizationSignal to the master's GroupSynchronizationSignal.
   g. On the Master channel, set GroupSynchronizationMask to the sum (or "or'ing") of all the values (2^SlaveSyncSignal)for the slaves it will trigger.
8. Set up and print the RF properties.
9. Play an ARB waveform file.
   a. Set up and load the encrypted ARB file.
      i. An encrypted ARB file is loaded into a catalog where it is referenced by a unique string key value.
      ii. Get the path of the executing assembly to prepend it to the waveform filename.
      iii. Load the ARB sequences into the modulator.
   b. To ensure power accuracy, turn off MIMO synchronization, play the ARB and use the DoPowerSearch method to do a power search.
   c. Play encrypted ARB file.
10. Disable the Modulation and RF Output.
11. Close all the drivers.

12. Close the Console Application.

Example programs may be found in: **C:\Program Files (x86)\IVI Foundation\IVI\Drivers\AgM938x\Examples**

Example Program 5 – Program Steps with Code Snippets

Step 1: Create a Visual C# Console Application

1. Launch Visual Studio and create a new Console Application in Visual C# by selecting: **File > New > Project**. Select a Visual C# Console Application. Enter "CS_MultiChannel" as the Name of the project and click **OK** .
2. 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 M9381A PXIe VSG driver interfaces, reference to its driver (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.
3. 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".)
4. Scroll to the IVI section and select the following type library; then select **OK**.
   IVI AgM938x 2.0 Type Library.

NOTE **When a reference for the AgM938x is added, the IVIDriver 2.0 Type Library is also automatically added. This reference houses the interface definitions for IVI inherent capabilities which are located in the file IviDriverTypeLib.dll (dynamically linked library).**

Step 3: Add Using Statements

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.

```
using System;
using System.IO;
using System.Reflection;
using Agilent.AgM938x.Interop;
```

Step 4: Prepare for Creation and Initialization of Drivers

1. If an external reference signal can be supplied to the M9300A "Ref In" port, then connect it and set this property to true. It will yield better results.

   ```
   private const Boolean USE_EXTERNAL_REFERENCE = false;
   ```

2. Declare master and slave drivers as instances of IAgM938xEx2.

   ```
   IAgM938xEx2 driverMaster = null;
   IAgM938xEx2 driverSlave = null;
   ```

   When creating driver instances for the M9381A PXI VSG, note that the IAgM938xEx2 interface is needed for multi-channel operation in place of the IAgM938x interface. The IAgM938xEx2 interface is an "extended" interface and includes additional commands that are needed for multi-channel capability that were not previously available in IAgM938x. This interface also exposes the Shared Synthesizer properties.

3. Specify the mode of operation (simulation or non-simulation) and the resource descriptor.
   The resource descriptor is the same value as the "Selected Instrument" field of the SFP's (Soft Front Panel) connection dialog. This can be a comma or semicolon separated list of module addresses such as PXI10::15::0::INSTR;PXI10::16::0::INSTR;PXI10::17::0::INSTR;PXI10::18::0::INSTR or the name of a saved "Instrument Connection" created by the SFP such as M9381A. Refer to the SFP Help topic *Connect to Instruments and Modules* for more details.
   You have two options to specify the resource descriptor and mode of operation:

   – Change the code to modify the default values, if required. Hard-code your resource descriptor and set the simulation mode to true or false.

     ```
     string resourceMaster = "M9381A-Master"; // Use the hardware
     associated with the connection named "M9381A-Master"
     string resourceSlave = "M9381A-Slave"; // Use the hardware
     associated with the connection named "M9381A-Slave"
     resourceSlave = "M9301A;M9310A;M9311A"; // Simulation mode:
     Initial Slave M9381A without M9300A

     bool simulated = true;
     ```

   – Pass the resource descriptor in the command line when you run the example program to use that hardware in non-simulated mode. The following program code takes care of the command-line arguments:

     ```
     if (args.Length > 1)
     ```

```
    {
        resourceMaster = args[0];
        resourceSlave = args[1];
        simulated = false;
    }
```

4. When using a shared synthesizer, set the following initialization properties:
   - ShareSynthesizerVisaSession flag to 1 or true
   - Each slave instance must set the SynthesizerRole flag to slave
   - Each instance should specify which output port of the synthesizer they are connected to by setting the SynthesizerOutputPort to the appropriate value. i.e. 1b, 2a, or 2b.

```
string masterDriverSetup = string.Empty;
string slaveDriverSetup = string.Empty;
masterDriverSetup = "ShareSynthesizerVisaSession=1,"; // Default for
SynthesizerRole=master.  Default for SynthesizerOutputPort=1a.
slaveDriverSetup =
"ShareSynthesizerVisaSession=1,SynthesizerRole=slave,SynthesizerOutputP
ort=1b,";

string masterOptions =
                string.Format("QueryInstrStatus=true, Simulate={0},
DriverSetup={1} Model=, Trace=false",
                              simulated ? "true" : "false",
masterDriverSetup);
string slaveOptions =
                string.Format("QueryInstrStatus=true, Simulate={0},
DriverSetup={1} Model=, Trace=false",
                              simulated ? "true" : "false",
slaveDriverSetup);
```

Step 5: Create and Initialize Driver Instances

1. Create Master and Slave drivers.

```
driverMaster = new AgM938xClass();
driverSlave = new AgM938xClass();

const bool idquery = true;
const bool reset = true;
```

2. Initialize the Master driver. See the *Initializing the IVI-COM Driver* topic in the M938x IVI documentation for additional information.

```
driverMaster.Initialize(resourceMaster, idquery, reset, masterOptions);
```

3. Clear startup messages and warnings, if any, on Master.

```
int errorcode = 0;
string message = string.Empty;
do
{
    driverMaster.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
    {
        Console.WriteLine(message);
    }
} while (errorcode != 0);

Console.WriteLine("Master Driver Initialized");
```

4. You may also want to print IVI Driver Identity and Shared Synthesizer properties.

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

Console.WriteLine("Shared Synthesizer Role: {0}",
driverMaster.Modules3.Synthesizer3.SharedRole);
Console.WriteLine("Synthesizer Output Port: {0}",
driverMaster.Modules3.Synthesizer3.OutputPort);
Console.WriteLine();
```

5. Similarly, initialize the Slave Driver, clear startup messages and warnings, and print its properties.

```
driverSlave.Initialize(resourceSlave, idquery, reset, slaveOptions);

errorcode = 0;
message = string.Empty;
do
{
    driverSlave.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
    {
        Console.WriteLine(message);
    }
} while (errorcode != 0);

Console.WriteLine("Slave Driver Initialized");


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

Console.WriteLine("Shared Synthesizer Role: {0}",
driverSlave.Modules3.Synthesizer3.SharedRole);
Console.WriteLine("Synthesizer Output Port: {0}",
driverSlave.Modules3.Synthesizer3.OutputPort);
Console.WriteLine();
```

NOTE When initializing the instruments, create the master first, followed by all the slaves.

Step 6: Enable an External Reference, if planned

- If the Master driver uses an external reference then enable it.

  ```
  driverMaster.Modules.Reference.ExternalReferenceEnabled = USE_EXTERNAL_
  REFERENCE;
  ```

- If the Slave uses an external reference then enable it.

  ```
  driverSlave.Modules.Reference.ExternalReferenceEnabled = USE_
  EXTERNAL_REFERENCE;
  ```

Step 7: Multichannel Sync Setup

You need to perform the following steps to configure the system for multi-channel operations.

1. The Reference module must be in the System Timing Slot. Set it up to drive the 10 MHz backplane clock. Usually, you can configure the Reference module to be in just one channel, for example, the Master.

   ```
   driverMaster.Modules2.Reference2.BackPlaneReferenceEnabled = true;
   ```

   If the Slave contains the Reference module, set it to drive the 10MHz backplane.

```
driverSlave.Modules2.Reference2.BackPlaneReferenceEnabled = true;
```

2. Initialize synchronization clocks for all source channels.

```
driverMaster.MultiChannelSync.InitializeSynchronizationClocks();
driverSlave.MultiChannelSync.InitializeSynchronizationClocks();
```

3. Set up the master/slave roles. Exactly one system must be the SystemMaster.

```
driverMaster.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleSystemMaste
r;
driverSlave.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleSlave;
```

4. On the Master channel, configure the GroupSynchronizationSignal which is used to trigger slave channels.

```
driverMaster.MultiChannelSync.GroupSynchronizationSignal =
AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_TRIGGER_1;
```

5. On each Slave channel set the SlaveSynchronizationSignal handshaking line to a unique PXI backplane trigger line.

```
driverSlave.MultiChannelSync.SlaveSynchronizationSignal =
AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_TRIGGER_2;
```

6. On each Slave channel, set the GroupSynchronizationSignal to the master's GroupSynchronizationSignal.

```
driverSlave.MultiChannelSync.GroupSynchronizationSignal =
AgM938xPXIResourcesEnum.AgM938xPXIResourcesTTL_TRIGGER_1;
```

7. On the Master channel, set GroupSynchronizationMask to the sum (or "or'ing") of all the values (2^SlaveSyncSignal)for the slaves it will trigger. The exponentiation (2^SlaveSyncSignal) can be accomplished via Pow (2,SlaveSyncSignal) or as below.

```
driverMaster.MultiChannelSync.GroupSynchronizationMask = 1 << (int)
driverSlave.MultiChannelSync.SlaveSynchronizationSignal;
```

After these steps, you can use the chassis interface and configure the chassis to properly route trigger lines between trigger segments. This step is critical for proper multi-channel operation when source channels span different bus segments.

Step 8: Set Up the RF Properties

To ensure that you have control over the drivers, print the RF properties of the drivers, change the values for these properties, and again display these properties.

```
const double frequency = 2.4e9; // 2.4 GHz
const double level = -3; // -3 dBm

Console.WriteLine("\t* Initial Master Frequency: {0:###0.#} MHz",
driverMaster.RF.Frequency / 1e6);
Console.WriteLine("\t* Initial Master Level: {0:0.#} dBm",
driverMaster.RF.Level);
Console.WriteLine("\t* Initial Slave Frequency: {0:###0.#} MHz",
driverSlave.RF.Frequency / 1e6);
Console.WriteLine("\t* Initial Slave Level: {0:0.#} dBm",
driverSlave.RF.Level);
Console.WriteLine();

driverMaster.RF.Frequency = frequency;
driverMaster.RF.Level = level;
driverMaster.RF.OutputEnabled = true;
driverMaster.Apply();

driverSlave.RF.Frequency = frequency;
driverSlave.RF.Level = level;
driverSlave.RF.OutputEnabled = true;
driverSlave.Apply();

Console.WriteLine("\t* New Master Frequency: {0:###0.#} MHz",
driverMaster.RF.Frequency / 1e6);
Console.WriteLine("\t* New Master Level: {0:0.#} dBm",
driverMaster.RF.Level);
Console.WriteLine("\t* New Slave Frequency: {0:###0.#} MHz",
driverSlave.RF.Frequency / 1e6);
Console.WriteLine("\t* New Slave Level: {0:0.#} dBm", driverSlave.RF.Level);
```

Step 9: Play an ARB Waveform File

This step involves creating the proper setup for loading an ARB file, performing a power search to ensure power accuracy, and playing the encrypted ARB file.

1. Set up and Load the Encrypted ARB file.1
   You need to perform the following steps to create the proper setup for loading an ARB file:
   a. An encrypted ARB file is loaded into a catalog where it is referenced by a unique string key value:

```
const string catalogReferenceName = "ExampleArb";

const string encryptedArbFileName = "12TONE.wfm";
Console.WriteLine("\t* Catalog Reference Name = {0}",
catalogReferenceName);
Console.WriteLine("\t* Encrypted ARB filename = {0}",
encryptedArbFileName);
Console.WriteLine("\t* Upload encrypted ARB file.");
```

b. Get the path of the executing assembly to prepend it to the waveform filename.

```
string location = Path.GetDirectoryName
(Assembly.GetExecutingAssembly().Location);
string arbFilePath = (location == null)
? encryptedArbFileName
: Path.Combine(location, encryptedArbFileName);
```

c. Load the ARB sequences into the modulator. Although it is known that this is the first ARB that is being played in the session, as a best practice, perform the following sequence of steps before you upload the file:

```
driverMaster.Modulation.Enabled = false;
driverMaster.Apply();
driverMaster.Modulation.Stop();
driverMaster.Modulation.IQ.RemoveArb(catalogReferenceName);
```

Now upload the ARB file.

```
driverMaster.Modulation.IQ.UploadArbAgilentFile
(catalogReferenceName, arbFilePath);
```

Perform similar steps for the slave driver.

```
driverSlave.Modulation.Enabled = false;
driverSlave.Apply();
driverSlave.Modulation.Stop();
driverSlave.Modulation.IQ.RemoveArb(catalogReferenceName);
driverSlave.Modulation.IQ.UploadArbAgilentFile
(catalogReferenceName, arbFilePath);
```

2. Power Search
To ensure power accuracy, turn off MIMO synchronization, play the ARB and use the DoPowerSearch method to do a power search.

```
Console.WriteLine();
Console.WriteLine("Power Search For Level Accuracy.");

double powerOffset = 0, scaleOffset = 0;

driverSlave.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleOff;
driverSlave.Modulation.Enabled = true;
driverSlave.Apply();
driverSlave.Modulation.PlayArb(catalogReferenceName,
AgM938xStartEventEnum.AgM938xStartEventImmediate);
driverSlave.Calibration.PowerSearch.DoPowerSearch(ref powerOffset, ref
scaleOffset);

Console.WriteLine("\t* Slave Power Offset: {0:0.###} dBm",
powerOffset);
```

```
Console.WriteLine("\t* Slave Scale Offset: {0:0.###}", scaleOffset);


powerOffset = 0; scaleOffset = 0;

driverMaster.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleOff;
driverMaster.Modulation.Enabled = true;
driverMaster.Apply();
driverMaster.Modulation.PlayArb(catalogReferenceName,
AgM938xStartEventEnum.AgM938xStartEventImmediate);
driverMaster.Calibration.PowerSearch.DoPowerSearch(ref powerOffset, ref
scaleOffset);

Console.WriteLine("\t* Master Power Offset: {0:0.###} dBm",
powerOffset);
Console.WriteLine("\t* Master Scale Offset: {0:0.###}", scaleOffset);
```

3. Play Encrypted ARB file.
   Re-enable MIMO synchronization and start a synchronous ARB playback. Always play the ARB on the slave channels first. This is analogous to the Arm() sequence for Multi-Channel PXIe VSAs.

```
Console.WriteLine();
Console.WriteLine("Play Synchronous ARB file.");

driverSlave.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleSlave;
driverSlave.Modulation.Enabled = true;
driverSlave.Apply();
driverSlave.Modulation.PlayArb(catalogReferenceName,
AgM938xStartEventEnum.AgM938xStartEventImmediate);

driverMaster.MultiChannelSync.SynchronizationRole =
AgM938xMultiChannelSyncRoleEnum.AgM938xMultiChannelSyncRoleSystemMaste
r;
driverMaster.Modulation.Enabled = true;
driverMaster.Apply();
driverMaster.Modulation.PlayArb(catalogReferenceName,
AgM938xStartEventEnum.AgM938xStartEventImmediate);
```

Step 10: Disable the Modulation and RF Output

```
driverMaster.Modulation.Enabled = false;
driverMaster.RF.OutputEnabled = false;
driverMaster.Apply();

driverSlave.Modulation.Enabled = false;
driverSlave.RF.OutputEnabled = false;
driverSlave.Apply();
```

Step 11: Close all the drivers

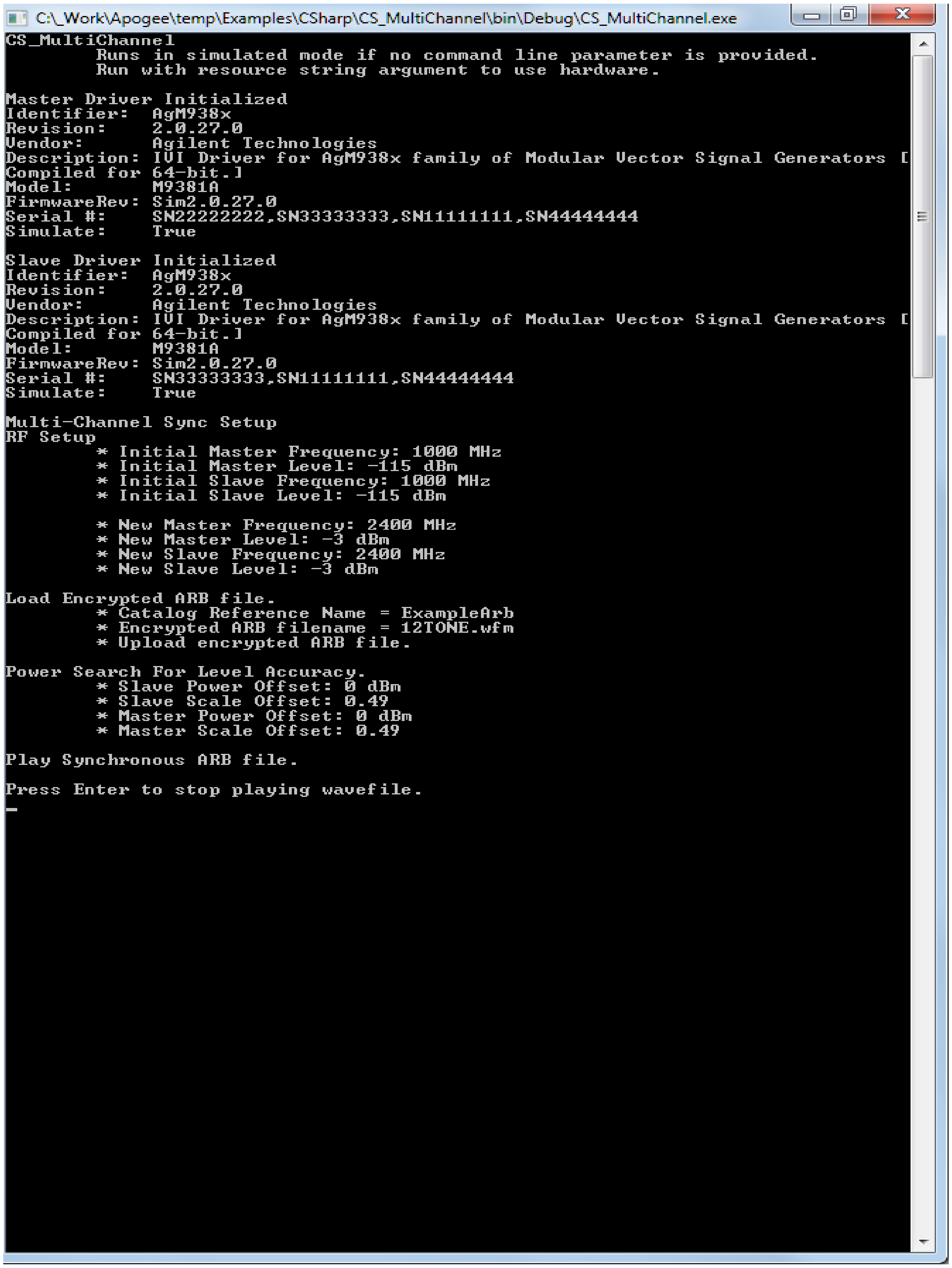
When closing the instruments it is recommended that you close all the slaves first and then close the master. Although no errors will be encountered if any other order is used, but this helps ensure that the "owner" of the synthesizer can cleanly shutdown the module. Note the use of the finally syntax, which implies that the preceding code was wrapped in a try/catch block.

```
finally
{
    if (driverSlave != null && driverSlave.Initialized)
    {
        driverSlave.Close();
        Console.WriteLine();
        Console.WriteLine("M9381A Slave driver closed.");
    }

    if (driverMaster != null && driverMaster.Initialized)
    {
        driverMaster.Close();
        Console.WriteLine();
        Console.WriteLine("M9381A Master driver closed.");
    }
}
```

Step 12: Close the Console Application

Prompt the user to press the Enter key to close the application.

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


C:_Work\Apogee\temp\Examples\CSharp\CS_MultiChannel\bin\Debug\CS_MultiChannel.exe

```
CS_MultiChannel
        Runs in simulated mode if no command line parameter is provided.
        Run with resource string argument to use hardware.

Master Driver Initialized
Identifier:   AgM938x
Revision:     2.0.27.0
Vendor:       Agilent Technologies
Description: IVI Driver for AgM938x family of Modular Vector Signal Generators [
Compiled for 64-bit.]
Model:        M9381A
FirmwareRev: Sim2.0.27.0
Serial #:     SN22222222,SN33333333,SN11111111,SN44444444
Simulate:     True

Slave Driver Initialized
Identifier:   AgM938x
Revision:     2.0.27.0
Vendor:       Agilent Technologies
Description: IVI Driver for AgM938x family of Modular Vector Signal Generators [
Compiled for 64-bit.]
Model:        M9381A
FirmwareRev: Sim2.0.27.0
Serial #:     SN33333333,SN11111111,SN44444444
Simulate:     True

Multi-Channel Sync Setup
RF Setup
        * Initial Master Frequency: 1000 MHz
        * Initial Master Level: -115 dBm
        * Initial Slave Frequency: 1000 MHz
        * Initial Slave Level: -115 dBm

        * New Master Frequency: 2400 MHz
        * New Master Level: -3 dBm
        * New Slave Frequency: 2400 MHz
        * New Slave Level: -3 dBm

Load Encrypted ARB file.
        * Catalog Reference Name = ExampleArb
        * Encrypted ARB filename = 12TONE.wfm
        * Upload encrypted ARB file.

Power Search For Level Accuracy.
        * Slave Power Offset: 0 dBm
        * Slave Scale Offset: 0.49
        * Master Power Offset: 0 dBm
        * Master Scale Offset: 0.49

Play Synchronous ARB file.

Press Enter to stop playing wavefile.
_
```

Example Program 6 – Pseudo-Code

This section presents the flow of the program. Code snippets are provided in the next section.

1. Create a Visual C# Console Application.
2. Add References. Add reference to IVI AgM9391 2.0 Type Library.
3. Add Using Statements.
4. Prepare for creation and initialization of drivers.
   a. Declare and set values for some configurable parameters to be used in the example.
   b. If an external reference signal can be supplied to the M9300A "Ref In" port, then connect it and set this property to true.
   c. Declare master and slave drivers as instances of AgM9391.
   d. Specify the mode of operation (simulation or non-simulation) and the resource descriptor.
   e. Set the initialization properties for the shared synthesizer.
5. Create and initialize driver instances.
   a. Create Master and Slave drivers.
   b. Initialize the Master driver.
   c. Clear startup messages and warnings, if any, on Master.
   d. Print Ivi Driver Identity and Shared Synthesizer properties for Master.
   e. Initialize the Slave driver.
   f. Clear startup messages and warnings, if any, on Slave.
   g. Print Ivi Driver Identity and Shared Synthesizer properties for Slave.
6. Enable an external reference, if planned.
7. Multichannel Sync Setup
   a. Set up the Reference module to drive the 10 MHz backplane clock.
   b. Initialize synchronization clocks for all source channels.
   c. Set up the master/slave roles. Exactly one system must be the SystemMaster.
   d. On the Master channel, configure the GroupSynchronizationSignal which is used to trigger slave channels.
   e. On each Slave channel set the SlaveSynchronizationSignal handshaking line to a unique PXI backplane trigger line.
   f. On each Slave channel, set the GroupSynchronizationSignal to the master's GroupSynchronizationSignal.
   g. On the Master channel, set GroupSynchronizationMask to the sum (or "or'ing") of all the values (2^SlaveSyncSignal)for the slaves it will trigger.
   h. Using the chassis interface, configure the chassis to properly route trigger lines between trigger segments.
8. Set the RF parameters.
9. Set up the receiver to make IQ measurements for a 40 MHz bursted signal.
10. Set up the triggering properties.

   a. Set up for a magnitude triggered mode acquisition.
   b. The triggering has to happen on the rising edge of the trigger and it happens when the signal crosses through the trigger level. Configure the master for the same.
   c. If unable to detect a burst, try a lower magnitude level or increase the output power level on the signal generator.
   d. Set up a bit of "off" time before the signal burst ramps up so that you can detect the beginning of the burst.
   e. Auto trigger on 10 second timeout if you never see a burst.

11. Apply the changes to the hardware.
12. Arm the receivers.
13. Set the timeout value on WaitForData.
14. Retrieve the IQ data.
15. Perform some actions on the interleaved IQ vector.
16. Close all the drivers.
17. Close the Console Application.

Example programs may be found in: **C:\Program Files (x86)\IVI Foundation\IVI\Drivers\AgM9391\Examples**

Example Program 6 - Program Steps with Code Snippets

Step 1: Create a Visual C# Console Application

1. Launch Visual Studio and create a new Console Application in Visual C# by selecting: **File > New > Project**. Select a Visual C# Console Application.
   Enter "CS_MultiChannel" as the Name of the project and click **OK** .
2. 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 M9391A PXIe VSA driver interfaces, reference to its driver (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.
3. 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".)
4. Scroll to the IVI section, select the following type library, and then select **OK**.
   IVI AgM9391 2.0 Type Library

NOTE

When a reference for the AgM9391 is added, the IVIDriver 2.0 Type Library is also automatically added. This reference houses the interface definitions for IVI inherent capabilities which are located in the file IviDriverTypeLib.dll (dynamically linked library).

Step 3: Add Using Statements

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.

```
using System;
using Agilent.AgM9391.Interop;
```

Step 4: Prepare for Creation and Initialization of Drivers

Perform the following steps.

1. Declare and set values for some configurable parameters to be used in the example.

   ```
   private const Double CENTER_FREQUENCY = 1000000000.0;
   private const Double EXPECTED_POWER = 2;
   private const Double IF_BANDWIDTH = 40000000.0;

   private const Double SAMPLE_RATE = 50000000.0;
   private const Double DURATION = 640e-6;

   private const Double MAGNITUDE_TRIGGER_LEVEL = -20.0;
   ```

2. If an external reference signal can be supplied to the M9300A "Ref In" port, then connect it and set this property to true. It will yield better results.

   ```
   private const Boolean USE_EXTERNAL_REFERENCE = false;
   ```

3. Declare the drivers.

   ```
   AgM9391 driverMaster = null;
   AgM9391 driverSlave = null;
   ```

4. Specify the mode of operation (simulation or non-simulation) and the resource descriptor.
   The resource descriptor is the same value as the "Selected Instrument" field of the SFP's (Soft Front Panel) connection dialog. This can be a comma or semicolon separated list of module addresses such as PXI10::15::0::INSTR;PXI10::16::0::INSTR;PXI10::17::0::INSTR;PXI10::18::0::INSTR or the name of a saved "Instrument Connection" created by the SFP such as M9391A. Refer to the SFP Help topic "Connect to Instruments and Modules" for more details.
   You have two options to specify the resource descriptor and mode of operation:

- Change the code to modify the default values, if required. Hard-code your resource descriptor and set the simulation mode to true or false.

  ```
  string resourceMaster = "M9391A-Master"; // Use the hardware
  associated with the connection named "M9391A-Master"
  string resourceSlave = "M9391A-Slave"; // Use the hardware
  associated with the connection named "M9391A-Slave"
  resourceSlave = "M9301A;M9214A;M9350A"; // Simulation mode:
  Initial Slave M9391A without M9300A

  bool simulated = true;
  ```

- Pass the resource descriptor in the command line when you run the example program to use that hardware in non-simulated mode.

  ```
  if (args.Length > 1)
  {
      resourceMaster = args[0];
      resourceSlave = args[1];
      simulated = false;
  }
  ```

5. When using a shared synthesizer, set the following initialization properties:
   - ShareSynthesizerVisaSession flag to 1 or true
   - Each slave instance must set the SynthesizerRole flag to slave
   - Each instance should specify which output port of the synthesizer they are connected to by setting the SynthesizerOutputPort to the appropriate value. i.e. 1b, 2a,, or 2b.

   ```
   string masterDriverSetup = string.Empty;
   string slaveDriverSetup = string.Empty;

   masterDriverSetup = "ShareSynthesizerVisaSession=1,"; // Default
   for SynthesizerRole=master. Default for SynthesizerOutputPort=1a.
   slaveDriverSetup =
   "ShareSynthesizerVisaSession=1,SynthesizerRole=slave,SynthesizerO
   utputPort=1b,";
   string masterOptions =
   string.Format("QueryInstrStatus=true, Simulate={0}, DriverSetup=
   {1} Model=, Trace=false",
   simulated ? "true" : "false", masterDriverSetup);
   string slaveOptions =
   string.Format("QueryInstrStatus=true, Simulate={0}, DriverSetup=
   {1} Model=, Trace=false",
   simulated ? "true" : "false", slaveDriverSetup);
   ```

Step 5: Create and Initialize Drivers

Perform the following steps.

1. Create Master and Slave Drivers.

```
driverMaster = new AgM9391Class();
driverSlave = new AgM9391Class();

const bool idquery = true;
const bool reset = true;
```

2. Initialize the Master driver. See driver help topic "Initializing the IVI-COM Driver" for additional information.

```
driverMaster.Initialize(resourceMaster, idquery, reset, masterOptions);
```

3. Clear startup messages & warnings if any, on Master.

```
int errorcode = 0;
string message = string.Empty;

do
{
    driverMaster.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
    {
        Console.WriteLine(message);
    }
} while (errorcode != 0);

Console.WriteLine("Master Driver Initialized");
```

4. You may also want to print IVI Driver Identity and Shared Synthesizer properties.

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

Console.WriteLine("Shared Synthesizer Role: {0}",
driverMaster.Modules3.Synthesizer2.SharedRole);
Console.WriteLine("Synthesizer Output Port: {0}",
driverMaster.Modules3.Synthesizer2.OutputPort);
Console.WriteLine();
```

5. Similarly, initialize the Slave Driver, clear startup messages and warnings, and

print its properties.

```
driverSlave.Initialize(resourceSlave, idquery, reset, slaveOptions);

errorcode = 0;
message = string.Empty;

do
{
    driverSlave.Utility.ErrorQuery(ref errorcode, ref message);
    if (errorcode != 0)
    {
        Console.WriteLine(message);
    }
} while (errorcode != 0);

Console.WriteLine("Slave Driver Initialized");

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

Console.WriteLine("Shared Synthesizer Role: {0}",
driverSlave.Modules3.Synthesizer2.SharedRole);
Console.WriteLine("Synthesizer Output Port: {0}",
driverSlave.Modules3.Synthesizer2.OutputPort);
Console.WriteLine();
```

NOTE When initializing the instruments it is recommended that the master be created first, followed by all the slaves.

Step 6: Enable an External Reference, if planned

– If the Master driver uses an external reference then enable it.

```
driverMaster.Modules.Reference.ExternalReferenceEnabled = USE_EXTERNAL_
REFERENCE;
```

– If the Slave driver uses an external reference then enable it.

```
driverSlave.Modules.Reference.ExternalReferenceEnabled = USE_EXTERNAL_
REFERENCE;
```

Step 7: Multichannel Sync Setup

Configure the system for multi-channel.

1. The Reference module must be in the System Timing Slot. Set it up to drive the 10 MHz backplane clock.

   ```
   driverMaster.Modules2.Reference2.BackPlaneReferenceEnabled = true;
   ```

   If the Slave contains the Reference module, you need to set it to drive the 10MHz backplane too.

   ```
   driverSlave.Modules2.Reference2.BackPlaneReferenceEnabled = true;
   ```

2. Initialize synchronization clocks for all receiver channels.

   ```
   driverMaster.MultiChannelSync.InitializeSynchronizationClocks();
   driverSlave.MultiChannelSync.InitializeSynchronizationClocks();
   ```

3. Set up the Master/Slave roles. Exactly one system must be the SystemMaster.

   ```
   driverMaster.MultiChannelSync.SynchronizationRole =
   AgM9391MultiChannelSyncRoleEnum.AgM9391MultiChannelSyncRoleSystemMaste
   r;
   driverSlave.MultiChannelSync.SynchronizationRole =
   AgM9391MultiChannelSyncRoleEnum.AgM9391MultiChannelSyncRoleSlave;
   ```

4. On the Master channel, configure the GroupSynchronizationSignal which is used to trigger slave channels.

   ```
   driverMaster.MultiChannelSync.GroupSynchronizationSignal =
   AgM9391PXIResourcesEnum.AgM9391PXIResourcesTTL_TRIGGER_1;
   ```

5. On each Slave channel set the SlaveSynchronizationSignal handshaking line to a unique PXI backplane trigger line.

   ```
   driverSlave.MultiChannelSync.SlaveSynchronizationSignal =
   AgM9391PXIResourcesEnum.AgM9391PXIResourcesTTL_TRIGGER_2;
   ```

6. On each Slave channel, set the GroupSynchronizationSignal to the master's GroupSynchronizationSignal.

   ```
   driverSlave.MultiChannelSync.GroupSynchronizationSignal =
   AgM9391PXIResourcesEnum.AgM9391PXIResourcesTTL_TRIGGER_1;
   ```

7. On the Master channel, set GroupSynchronizationMask to the sum (or "or'ing") of all the values (2^SlaveSyncSignal) for the slaves it will trigger. The exponentiation (2^SlaveSyncSignal) can be accomplished via Pow (2,SlaveSyncSignal) or as below.

   ```
   driverMaster.MultiChannelSync.GroupSynchronizationMask = 1 << (int)
   driverSlave.MultiChannelSync.SlaveSynchronizationSignal;
   ```

8. Using the chassis interface, configure the chassis to properly route trigger lines between trigger segments. This step is critical for proper multi-channel operation when receiver channels span different bus segments.

Step 8: Set the RF parameters

Set the measurement at the center frequency. Set a low power and we will search for an optimum power level where the receiver is not overloaded. Let the receiver pick the suggested conversion mode for the frequency. Use the largest IF bandwidth supported by the hardware.

```
driverMaster.RF.Frequency = CENTER_FREQUENCY;
driverMaster.RF.Power = EXPECTED_POWER;
driverMaster.RF.Conversion = AgM9391ConversionEnum.AgM9391ConversionAuto;
driverMaster.RF.IFBandwidth = IF_BANDWIDTH;

driverSlave.RF.Frequency = CENTER_FREQUENCY;
driverSlave.RF.Power = EXPECTED_POWER;
driverSlave.RF.Conversion = AgM9391ConversionEnum.AgM9391ConversionAuto;
driverSlave.RF.IFBandwidth = IF_BANDWIDTH;
```

Step 9: Set up the receiver to make IQ measurements for a 40 MHz bursted signal with which we will use a magnitude trigger

```
driverMaster.AcquisitionMode =
AgM9391AcquisitionModeEnum.AgM9391AcquisitionModeIQ;
driverMaster.IQAcquisition.SampleRate = SAMPLE_RATE;
driverMaster.IQAcquisition.SampleSize =
AgM9391SampleSizeEnum.AgM9391SampleSize64Bits;
driverMaster.IQAcquisition.Samples = (int)(DURATION * SAMPLE_RATE);

driverSlave.AcquisitionMode =
AgM9391AcquisitionModeEnum.AgM9391AcquisitionModeIQ;
driverSlave.IQAcquisition.SampleRate = SAMPLE_RATE;
driverSlave.IQAcquisition.SampleSize =
AgM9391SampleSizeEnum.AgM9391SampleSize64Bits;
driverSlave.IQAcquisition.Samples = (int)(DURATION * SAMPLE_RATE);
```

Step 10: Set up the triggering properties

Perform the following steps.

1. Since we anticipate a bursted signal we will set for a magnitude triggered mode acquisition.

   ```
   driverMaster.Triggers.AcquisitionTrigger.Mode =
   AgM9391AcquisitionTriggerModeEnum.AgM9391AcquisitionTriggerModeMagnitud
   e;
   ```

2. The Slave channel also needs to know what triggering mode is set.

   ```
   driverSlave.Triggers.AcquisitionTrigger.Mode =
   AgM9391AcquisitionTriggerModeEnum.AgM9391AcquisitionTriggerModeMagnitud
   e;
   ```

3. The triggering has to happen on the rising edge of the trigger and it happens when the signal crosses through the trigger level. If triggering is to be used, only the master is configured for the trigger. The master will hold off the slaves until trigger is received.

   ```
   IAgM9391TriggersAcquisitionTrigger acqTriggerMaster =
   driverMaster.Triggers.AcquisitionTrigger;

   acqTriggerMaster.MagnitudeTrigger.Slope =
   AgM9391TriggerSlopeEnum.AgM9391TriggerSlopePositive;
   ```

4. If unable to detect a burst, try a lower magnitude level or increase the output power level on the signal generator.

   ```
   acqTriggerMaster.MagnitudeTrigger.Level = MAGNITUDE_TRIGGER_LEVEL;
   ```

5. A bit of "off" time is required before the signal burst ramps up so we can detect the beginning of the burst.

   ```
   acqTriggerMaster.MagnitudeTrigger.PulseOffTime = 1e-6; // 1 usec should
   be enough.
   ```

6. You need to decide what to do if you never see a burst. As an example, you can auto trigger on 10 second timeout if no burst meets the criteria and just acquire what you can.

   ```
   acqTriggerMaster.TimeoutMode =
   AgM9391TriggerTimeoutModeEnum.AgM9391TriggerTimeoutModeAutoTriggerOnTim
   eout;
   acqTriggerMaster.Timeout = 10000; // ms
   ```

Step 11: Apply the changes to hardware

Apply changes to the Master first, then the Slave(s).

NOTE **The MultichannelSync.DelayAdjust property must be copied from the Master (after Master Apply()) to the Slave(s) (before Slave Apply()).**

```
driverMaster.Apply();
driverSlave.MultiChannelSync.DelayAdjust =
driverMaster.MultiChannelSync.DelayAdjust;
driverSlave.Apply();
```

Step 12: Arm the receivers

The Slave(s) must be Armed before the Master.

```
driverSlave.Arm();
driverMaster.Arm();

bool overloaded = false;
```

Step 13: Set the timeout value on WaitForData

When trigger time out mode is set to auto trigger on time out, the timeout value on WaitForData should be longer than the auto trigger timeout and long enough to allow data acquisition to complete. This example waits 10 seconds before auto triggering if no magnitude trigger has occurred and waits for 11 seconds for data acquisition to complete.

```
int waitForDataTimeout = (int)( acqTriggerMaster.Timeout * 1.1 );
if( !driverMaster.WaitForData( waitForDataTimeout ) ) // in milliseconds
{
    throw new ApplicationException( "WaitForData failed. No acquisition was
made." );
}
```

Step 14: Retrieve the IQ data

```
double[] interleavedIQBlockMaster = new double
[driverMaster.IQAcquisition.Samples * 2];
double[] interleavedIQBlockSlave = new double
[driverSlave.IQAcquisition.Samples * 2];

driverMaster.IQAcquisition.ReadIQData( 0,

AgM9391IQUnitsEnum.AgM9391IQUnitsSquareRootMilliWatts,
                                       0,
                                       driverMaster.IQAcquisition.Samples,
                                       ref interleavedIQBlockMaster,
                                       ref overloaded );

driverSlave.IQAcquisition.ReadIQData( 0,

AgM9391IQUnitsEnum.AgM9391IQUnitsSquareRootMilliWatts,
                                      0,
                                      driverSlave.IQAcquisition.Samples,
                                      ref interleavedIQBlockSlave,
                                      ref overloaded );
```

Step 15: Perform some action on the interleaved IQ vector

Here, you can enter your own program logic.

Step 16: Close all the drivers

When closing the instruments it is recommended that you close all the slaves first and then close the master. Although no errors will be encountered if any other order is used, but this helps ensure that the "owner" of the synthesizer can cleanly shutdown the module. Note the use of the finally syntax, which implies that preceding code was wrapped in a try/catch block.

```
finally
{
    if (driverMaster != null && driverMaster.Initialized)
    {
        driverMaster.Close();
        Console.WriteLine();
        Console.WriteLine("M9391A Master driver closed.");
    }
    if (driverSlave != null && driverSlave.Initialized)
    {
        driverSlave.Close();
        Console.WriteLine();
        Console.WriteLine("M9391A Slave driver closed.");
    }
}
```

Step 17: Close the Console Application

Prompt the user to press the Enter key to close the application.

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

Hints for Various Configuration Tasks

This section focuses on providing helpful pointers for programmatically performing some configuration tasks for M9391A VSG and M9381A VSG.

Restarting an Already Playing Waveform in M9381A VSG

You can configure an already playing arbitrary waveform on the M9381A VSG to restart in response to an external trigger.

Use Case

You might need to synchronize the meta information between a base station and an M9381A VSG emulating a mobile handset. For example, in an LTE system, infrequently changing information like cell identity and frequency plan is not transmitted in every 10 ms frame. However, the base station and mobile unit are required to be in sync on where they are in the frame sequence to agree upon things like the Master Information Block (MIB) and various types of System Information Blocks (SIB). The frame sequence is 1024 frames long and there is a counter called the System Frame Number (SFN) from 0 to 1023 frames. So, the base station issues a sync command on its SFN zero, and the mobile device transmits back a frame at its SFN zero. Once this agreement is reached, the user equipment can keep transmitting the same block of 1024 frames. Hence, the VSG must reset its System Frame Number when the base station issues an external trigger.

Description

This external trigger can be applied on the Trig 1 port of the Synthesizer (M9301A) module. Once the trigger is received, playback restarts in less than 14 usec of the trigger being received. During this time, a continuous waveform is played with the same RMS power as that of the original arbitrary waveform.

This feature is supported for both single channel and multi-channel synchronization operation for up to four channels contained in a single chassis. For multi-channel operation, there is an additional 0-100ns latency which varies on each restart due to the internal multi-channel synchronization mechanism.

NOTE

The initial waveform playback starts immediately; there is no provision for the initial waveform playback to start on receiving an external trigger.

The Arb Restart feature is not supported in the List Mode.

Program Steps

The following is the sequence of steps to be followed:

1. Configure the External Trigger using the External Trigger IVI commands. Refer to the IVI documentation for more information about the External Trigger IVI commands. The following is an example:

   ```
   driver.Triggers3.ExternalTrigger3.Configure(true, 1.6e-8, 0.5,
   AgM938xTriggerSlopeEnum.AgM938xTriggerSlopePositive,
   AgM938xTriggerEnum.AgM938xTriggerFrontPanelTrigger1,
   AgM938xTriggerTerminationEnum.AgM938xTriggerTermination50Ohm, timeout,
   AgM938xTimeoutModeEnum.AgM938xTimeoutModeWaitInfinite);
   ```

   Applying an external trigger in a multi-channel setup
   For a multi-channel setup, the external trigger must be routed to all the channels (master and slaves). The master channel receives the trigger from the front panel Trig1 on the Synthesizer module. The slave channel(s) can share the backplane trigger that routes the front panel Trig1 from the master channel Synthesizer to the master channel Digital Vector Modulator (M9311A). By default, the backplane trigger line Trig6 is used. Configure the chassis so Trig6 is routed to all the trigger segments, radiating out from the master Synthesizer. Set the master External Trigger source to FrontPanelTrigger1 and configure the slave modules to use PXITriggerTrig6 as the External Trigger Source. Ensure that Ext Trig is enabled on the slave.

2. Enable the ArbRestart feature by setting the `ExternalTrigger3.ArbRestart` property to true and calling the `Apply` method.

   ```
   driver.Triggers3.ExternalTrigger3.ArbRestart = true;
   driver.Apply();
   ```

   For a multi-channel setup, perform this step for all masters and slaves.

   NOTE **The Arb Restart is generally enabled before playing the Arb waveform, but can also be enabled or disabled after the arbitrary waveform has started playing.**

3. Play an arbitrary waveform using the `PlayArb()` function, specifying `StartEvent` as immediate.

   ```
   driver.Modulation.PlayArb(catalogReferenceName,
   AgM938xStartEventEnum.AgM938xStartEventImmediate);
   ```

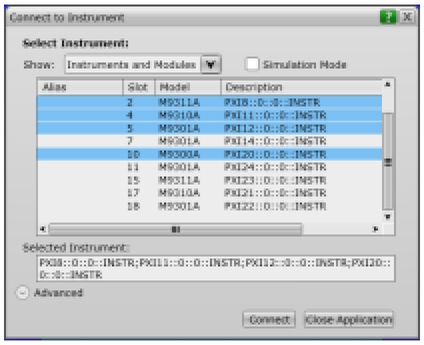
   where `catalogReferenceName` is the name used to reference the uploaded waveform.
   The playback starts immediately.
   For a multi-channel setup, perform this step on all the slaves first, followed by the master.

Appendix - Determining Resource Name Address Strings

The following information is for 2x2 MIMO in one M9018A PXIe Chassis.

Using the M9381A PXIe VSG #1 Soft Front Panel to get a Resource Name address string:

```
string VsgResourceName =
"PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI20::0::0::INSTR";
```



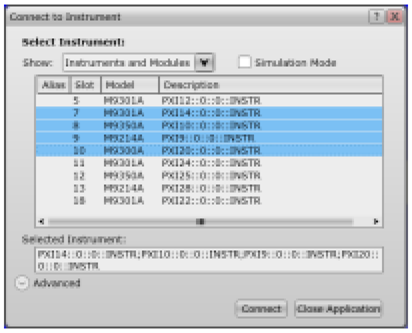
Alias	Slot	Model	Description
	2	M9311A	PXI8::0::0::INSTR
	4	M9310A	PXI11::0::0::INSTR
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	15	M9311A	PXI23::0::0::INSTR
	17	M9310A	PXI21::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument:
PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI20::0::0::INSTR

Slot	Model/Module Name	VISA Address
2	M9311A PXIe Modulator	PXI8::0::0::INSTR;
4	M9310A PXIe Source Output	PXI11::0::0::INSTR;
5	M9301A PXIe Synthesizer	PXI12::0::0::INSTR;
10	M9300A PXIe Reference	PXI20::0::0::INSTR;

Using the M9391A PXIe VSA #1 Soft Front Panel to get a Resource Name address string:

```
string VsaResourceName =
"PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI20::0::0::INSTR";
```



Alias	Slot	Model	Description
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	8	M9350A	PXI10::0::0::INSTR
	9	M9214A	PXI9::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	12	M9350A	PXI25::0::0::INSTR
	13	M9214A	PXI28::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

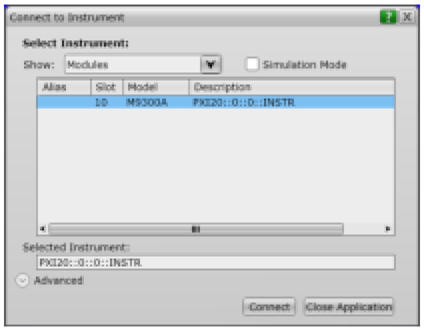
Selected Instrument:
PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI20::0::0::INSTR

Slot	Model/Module Name	VISA Address
7	M9301A PXIe Synthesizer	PXI14::0::0::INSTR;
8	M9350A PXIe Downconverter	PXI10::0::0::INSTR;
9	M9214A PXIe IF Digitizer	PXI9::0::0::INSTR;

10	M9300A PXIe Reference	PXI20::0::0::INSTR;

Using the M9300A PXIe Reference Soft Front Panel to get a Resource Name address string:

```
string ReferenceResourceName = "PXI20::0::0::INSTR";
```


Connect to Instrument

Select Instrument:

Show: Modules ☐ Simulation Mode

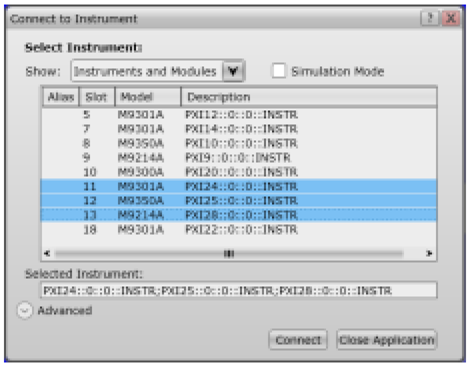
Alias	Slot	Model	Description
	10	M9300A	PXI20::0::0::INSTR

Selected Instrument: PXI20::0::0::INSTR

Slot	Model/Module Name	VISA Address
10	M9300A PXIe Reference	PXI20::0::0::INSTR;

Using the M9391A PXIe VSA #2 Soft Front Panel to get a Resource Name address string:

```
string VsaResourceName =
"PXI24::0::0::INSTR;PXI25::0::0::INSTR;PXI28::0::0::INSTR";
```


Connect to Instrument

Select Instrument:

Show: Instruments and Modules ☐ Simulation Mode

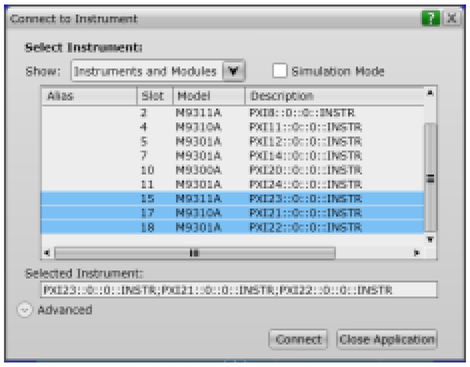
Alias	Slot	Model	Description
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	8	M9350A	PXI10::0::0::INSTR
	9	M9214A	PXI9::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	12	M9350A	PXI25::0::0::INSTR
	13	M9214A	PXI28::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument: PXI24::0::0::INSTR;PXI25::0::0::INSTR;PXI28::0::0::INSTR

Slot	Model/Module Name	VISA Address
11	M9301A PXIe Synthesizer	PXI24::0::0::INSTR;
12	M9350A PXIe Downconverter	PXI25::0::0::INSTR;
13	M9214A PXIe IF Digitizer	PXI28:0::0::INSTR;

Using the M9381A PXIe VSG #2 Soft Front Panel to get a Resource Name address string:

```
string VsgResourceName =
"PXI23::0::0::INSTR;PXI21::0::0::INSTR;PXI22::0::0::INSTR";
```



Connect to Instrument

Select Instrument:

Show: Instruments and Modules

Simulation Mode

Alias	Slot	Model	Description
	2	M9311A	PXI8::0::0::INSTR
	4	M9310A	PXI11::0::0::INSTR
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	15	M9311A	PXI23::0::0::INSTR
	17	M9310A	PXI21::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument:

PXI23::0::0::INSTR;PXI21::0::0::INSTR;PXI22::0::0::INSTR

	Slot	Model/Module Name	VISA Address
	15	M9311A PXIe Modulator	PXI23::0::0::INSTR;
	17	M9310A PXIe Source Output	PXI21::0::0::INSTR;
	18	M9301A PXIe Synthesizer	PXI22::0::0::INSTR;

Appendix - Verify Instruments Connect, Pass Self-Test, and are Updated

Before you attempt to programmatically control any hardware and make 802.11ac MIMO R&D/DVT Test measurements, connect to each of the instrument soft front panels, one at a time, perform self-test, and verify their FPGA firmware is fully updated. If any firmware updates are made, perform the self-test again.

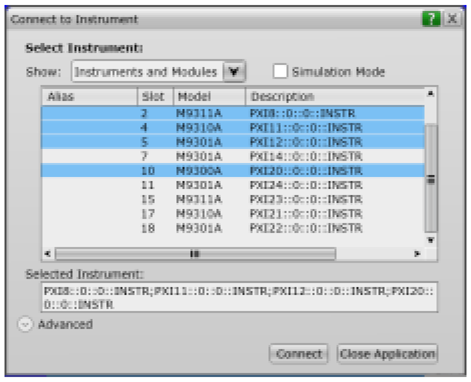
NOTE

Running Self-Test will fail if the modules that form an M9381A PXIe VSG or an M9391A PXIe VSA spans across slot 6 or slot 12 of the M9018A PXIe Chassis; if they do span across slot 6 or slot 12, the backplane triggers and bus segments must be routed properly. For details, see "Step 6 – Route Backplane Triggers and Bus Segments on the M9018A PXIe Chassis" on page 61.

In the following procedures, each instrument connection must be verified, each instrument must pass self-test, and each instrument's firmware version should be checked and updated if needed.

Verify that VSG 1 is Connected, Passes Self-Test, and Contains Up to Date Firmware

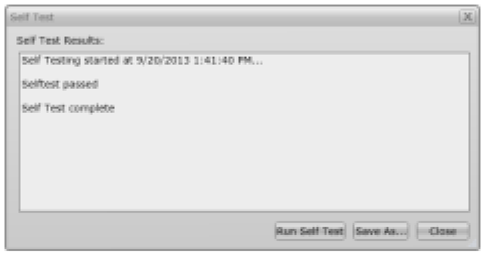
1. Select Start > All Programs > Keysight > M938x > **M9381 SFP** and run the soft front panel of the M9381A PXIe VSG - connect to VSG #1 and the M9300A PXIe Reference.



Alias	Slot	Model	Description
	2	M9311A	PXI8::0::0::INSTR
	4	M9310A	PXI11::0::0::INSTR
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	15	M9311A	PXI23::0::0::INSTR
	17	M9310A	PXI21::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument: PXI8::0::0::INSTR;PXI11::0::0::INSTR;PXI12::0::0::INSTR;PXI20::0::0::INSTR

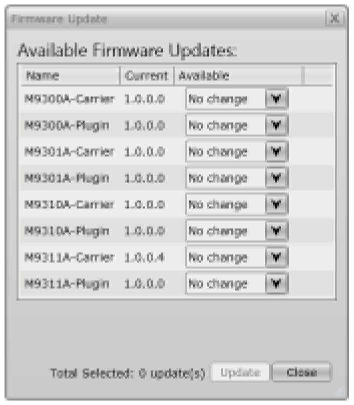
2. Run self-test.



Self Test Results:

```
Self Testing started at 9/20/2013 1:41:40 PM...
Selftest passed
Self Test complete
```

3. Check firmware and update if necessary.



Firmware Update

Available Firmware Updates:

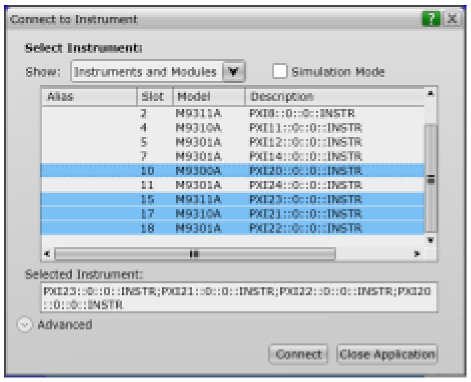
Name	Current	Available
M9300A-Carrier	1.0.0.0	No change
M9300A-Plugin	1.0.0.0	No change
M9301A-Carrier	1.0.0.0	No change
M9301A-Plugin	1.0.0.0	No change
M9310A-Carrier	1.0.0.0	No change
M9310A-Plugin	1.0.0.0	No change
M9311A-Carrier	1.0.0.4	No change
M9311A-Plugin	1.0.0.0	No change

Total Selected: 0 update(s)

4. Close the **Firmware Update** dialog box if no firmware updates are necessary.If firmware updates are required, install the updates, shut down the computer, cycle power on the M9018A PXIe Chassis, and repeat this procedure to verify connection, perform self-test, and verify that no further firmware updates are necessary.

Verify that VSG 2 is Connected, Passes Self-Test, and Contains Up to Date Firmware

1. Select Start > All Programs > Keysight > M938x > **M9381 SFP** and run the soft front panel of the M9381A PXIe VSG - connect to VSG #2 and the M9300A PXIe Reference.
2. Run self-test.
3. Check firmware and update if necessary.
4. Close the **Firmware Update** dialog box if no firmware updates are necessary.If firmware updates are required, install the updates, shut down the computer, cycle power on the M9018A PXIe Chassis, and repeat this procedure to verify connection, perform self-test, and verify that no further firmware updates are necessary.



Connect to Instrument

Select Instrument:

Show: Instruments and Modules

Simulation Mode: ☐

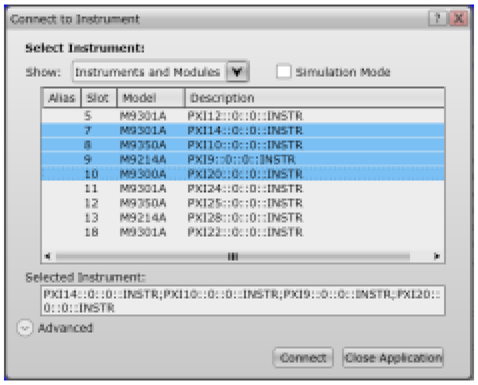
Alias	Slot	Model	Description
	2	M9311A	PXI8::0::0::INSTR
	4	M9310A	PXI11::0::0::INSTR
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	15	M9311A	PXI23::0::0::INSTR
	17	M9310A	PXI21::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument:

PXI23::0::0::INSTR;PXI21::0::0::INSTR;PXI22::0::0::INSTR;PXI20::0::0::INSTR

Verify that VSA 1 is Connected, Passes Self-Test, and Contains Up to Date Firmware

1. Select Start > All Programs > Keysight > M9391 > **M9391 SFP** and run the soft front panel of the M9391A PXIe VSA - connect to VSA #1 and the M9300A.
2. Run self-test.
3. Check firmware and update if necessary.
4. Close the **Firmware Update** dialog box if no firmware updates are necessary.If required, install the updates, shut down the computer, cycle power on the M9018A PXIe Chassis, and repeat this procedure to verify connection, perform self-test, and verify that no further firmware updates are necessary.



Select Instrument:

Show: Instruments and Modules

☐ Simulation Mode

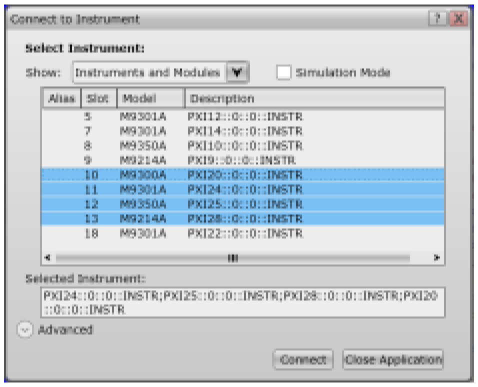
Alias	Slot	Model	Description
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	8	M9350A	PXI10::0::0::INSTR
	9	M9214A	PXI9::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	12	M9350A	PXI25::0::0::INSTR
	13	M9214A	PXI28::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument:

PXI14::0::0::INSTR;PXI10::0::0::INSTR;PXI9::0::0::INSTR;PXI20::0::0::INSTR

Verify that VSA 2 is Connected, Passes Self-Test, and Contains Up to Date Firmware

1. Select Start > All Programs > Keysight > M9391 > **M9391 SFP** and run the soft front panel of the M9391A PXIe VSA - connect to VSA #2 and the M9300A.
2. Run self-test.
3. Check firmware and update if necessary.
4. Close the **Firmware Update** dialog box if no firmware updates are necessary.If firmware updates are required, install the updates, shut down the computer, cycle power on the M9018A PXIe Chassis, and repeat this procedure to verify connection,



Select Instrument:

Show: Instruments and Modules

☐ Simulation Mode

Alias	Slot	Model	Description
	5	M9301A	PXI12::0::0::INSTR
	7	M9301A	PXI14::0::0::INSTR
	8	M9350A	PXI10::0::0::INSTR
	9	M9214A	PXI9::0::0::INSTR
	10	M9300A	PXI20::0::0::INSTR
	11	M9301A	PXI24::0::0::INSTR
	12	M9350A	PXI25::0::0::INSTR
	13	M9214A	PXI28::0::0::INSTR
	18	M9301A	PXI22::0::0::INSTR

Selected Instrument:

PXI24::0::0::INSTR;PXI25::0::0::INSTR;PXI28::0::0::INSTR;PXI20::0::0::INSTR perform self-test, and verify that no further firmware updates are necessary.

Appendix - Using LO Distribution Network in Multi-Channel Systems

You can use the four output ports of an M9301A Synthesizer to provide an LO signal for up to four M9381A Vector Signal Generators or M9391A Vector Signal Analyzers. For more than four VSGs or VSAs, you can use a single output port of the M301A Synthesizer routed through a V2802A LO Distribution Network to split the signal. For more information, refer to the *Sharing the M9301A Synthesizer's LO* topic in the *Keysight M9391A and M9381A Startup Guide*.

Initialization Settings for LO Distribution Network

When using an LO Distribution Network you must configure the following option in the "`DriverSetup=`" section of the`OptionString` during the initialization process:

Option Name	Description	Default
ExtLoDistributionUnit	Set this option to true, or use a supported model number such as V2802A. Setting this option prepares the driver for use with the LO Distribution Network by loading the LO Level field alignment data. CAUTION The V2802A LO Distribution Network has approximately +4 dBm of nominal gain across its range of operation. The M9311A Modulator module is highly sensitive to excessive power levels at the LO input port. Therefore, be sure to set the `ExtLoDistributionUnit` option to true at initialization whenever the system is configured with an LO Distribution Network to prevent possible damage to the module(s).	False

LO Level Field Alignment

The LO Level field alignment adjusts the signal level from a shared M9301A Synthesizer module to offset the effects of additional loss from non-standard cabling, additional gain from V2802A LO Distribution Network, and temperature variation. This alignment should be performed on all the instrument drivers sharing a M9301A Synthesizer module following configuration changes, such as LO cable changes, or selecting a different M9301A Synthesizer module to use as the shared synthesizer in a system.

This alignment must be performed on the Master channel first, followed by all the slave channels in the system and is invoked via the`LOLevelAlignment.AlignLoLevel`method for M9381A VSG and the`Align`method with the`AgM9391AlignmentTypeLOLevel`option for M9391A VSA.

VSG Code Snippet

```
mMaster.Calibration.LOLevelAlignment.AlignLoLevel();
foreach( var slave in mSlaves )
{
        slave.Calibration.LOLevelAlignment.AlignLoLevel();
}
```

VSA Code Snippet

```
mMaster.Calibration.Align(
AgM9391AlignmentTypeEnum.AgM9391AlignmentTypeLOLevel );
foreach( var slave in mSlaves )
{
        slave.Calibration.Align(
AgM9391AlignmentTypeEnum.AgM9391AlignmentTypeLOLevel );
}
```

NOTE

When performing alignments, the LO level alignment should be the first alignment invoked whenever an LO distribution Network is being used in the system. All subsequent alignments rely on the proper LO signal level to be present at the LO input port. For the M9391A VSA, this alignment will automatically be performed first when the `Align`method with the `AgM9391AlignmentTypeComprehensive`option is invoked, or if the user is performing a Keysight 89600 VSA full calibration.

The results of the alignment are retained on the host controller and are used in subsequent sessions. The alignment data is unique to the M9301A Synthesizer module used as the shared synthesizer, and the M9311A Modulator module, or M9350A Downconverter module that was used during the alignment. Reconfiguring the system to use a different M9301A Synthesizer module as the shared synthesizer, or changing the shared roles of the instruments in the system causes the driver to report an error and recommend the user to rerun the alignment.

After the first alignment, a series of validations are performed at startup to verify that the appropriate alignment data exists when operating in shared LO configuration.

The validations include:

- Alignment data exists for the Master as well as the Slave channels.
- The time elapsed since the last alignment is less than 30 days. Further, it is verified that the Slave alignment and Master alignment were performed at the same time, i.e., the user did not just perform the alignment on the Master which would invalidate the Slave alignment data.
- The temperature of the Modulator/Downconverter is within 10 deg. of the last alignment.
- The driver version matches the version used during the alignment.

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.

This Page Intentionally Left Blank

KEYSIGHT TECHNOLOGIES

M9300-90080
www.keysight.com