PROGRAMMING EXAMPLE 5

Synchronized Multi-Channel Mixed-Signal Generation using M3xxxA PXI Instruments

PATHWAVE

In this programming example, KS2201A PathWave Test Sync Executive is used to program multiple M3xxx Arbitrary Waveform Generators (AWGs) to synchronously generate mixed signals. Each instrument can be programmed to output either a Front Panel (FP) marker pulse or a previously queued waveform. All signal channels run fully synchronized and actions across instruments can be controlled with the timing resolution of the M3xxxA AWGs which is of 10ns.

```
# Add Sync Multi-Sequence Block
sync_block = hvi.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs", 10)

# Add instructions to each local sequence within the multi-sequence block
for index in range(0, hvi.engines.count):
    # Retrieve the engine sequence from the collection of HVI Local Sequences
    # HVI Local Sequence Collection is automatically created form the user-defined HVI Engine Collect
    # Each HVI Local Sequence can be retrieved using the name alias of the corresponding HVI Engine
    sequence = sync_block.sequences[hvi.engines[index].name]

    # Add FP Trigger ON to all instruments
    instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, hvi.instruction_set.trigger_writ
    instr_trigger_ON.set_parameter(hvi.instruction_set.trigger_write.sync_mode, kthvi.SyncMode.IMMED
    instr_trigger_ON.set_parameter(hvi.instruction_set.trigger_write.trigger, fp_trigger_list[index]
    instr_trigger_ON.set_parameter(hvi.instruction_set.trigger_write.value, kthvi.TriggerValue.ON)

    # Add FP Trigger ON to all instruments
    instr_trigger_OFF = sequence.add_instruction("FP Trigger OFF", 100, hvi.instruction_set.trigger_
    instr_trigger_OFF.set_parameter(hvi.instruction_set.trigger_write.sync_mode, kthvi.SyncMode.IMME
    instr_trigger_OFF.set_parameter(hvi.instruction_set.trigger_write.trigger, fp_trigger_list[index
    instr_trigger_OFF.set_parameter(hvi.instruction_set.trigger_write.value, kthvi.TriggerValue.OFF)

    # Add AWG trigger to the HVI sequence of each module
    # "Action Execute" instruction executes the AWG trigger from HVI
    instruction1 = sequence.add_instruction("AWG trigger", 10, hvi.instruction_set.action_execute.id
    instruction1.set_parameter(hvi.instruction_set.action_execute.action, awg_action_lists[index])
```

KEYSIGHT TECHNOLOGIES

Table of Contents

KS2201A - Programming Example 5 - Synchronized Multi-Channel Mixed-Signal Generation using M3xxxA PXIe Instruments

In this programming example, KS2201A PathWave Test Sync Executive is used to program multiple M3xxx Arbitrary Waveform Generators (AWGs) to synchronously generate mixed signals. Each instrument can be programmed to output either a Front Panel (FP) marker pulse or a previously queued waveform. All signal channels run fully synchronized and actions across instruments can be controlled with the timing resolution of the M3xxx AWG which is 10ns.

Introduction

This document is organized as follows. First, a "System Setup" section explains all the mandatory software and firmware components to be installed before the programming example can run. Secondly, a "Programming Example Overview" section describes the application use case of this programming example including expected measurement results. The next section contains detailed explanations on how to use the HVI (Hard Virtual Instrument) API (Application Programming Interface) to implement the real-time algorithms of this example. Finally, the conclusions are outlined.

NOTE Please review in detail the System Requirements outlined in the next section and install all the necessary software (SW) and firmware (FW) components before executing this programming example code.

System Setup

Please review the following system requirements and install the software (SW), firmware (FW), and driver version following the instructions provided in this section. To download the programming example code and necessary files please visit www.keysight.com/find/KS2201A-programming-examples. To download the latest PathWave Test Sync Executive installer and documentation please visit www.keysight.com/find/KS2201A-downloads. The rest of software installers FPGA firmware, drivers and other components mentioned in this section can be found on www.keysight.com

System Requirements

The versions of software, FPGA firmware, drivers, and other components that are required to run this programming example are listed below. All pieces of SW and firmware listed below need to be installed on the external PC or internal chassis controller that is used to control the PXI chassis. FPGA FW of PXI instruments can be instead programmed using the "Hardware Manager" window of SD1 Software Front Panel (SFP).

1. Software versions required:
   - Microsoft Visual Studio 2017 (or later)
   - .NET Framework 4.7.2 (or later)
   - Keysight IO Libraries Suite 2021 (v18.2.27115.0 or later)
   - Keysight SD1 Drivers, Libraries and SFP (v3.3.9 or later)
   - Keysight M9032A / M9033A Drivers, Libraries and SFP (v1.0.847.0 or later)
   - Keysight PathWave Test Sync Executive 2021 (v1.15.7 or later)

2. Chassis firmware and driver:
   - Keysight Chassis M9019A firmware (v2019EnhTrig or later)
   - Keysight PXIe Chassis Family Driver (v1.7.601.0 or later)

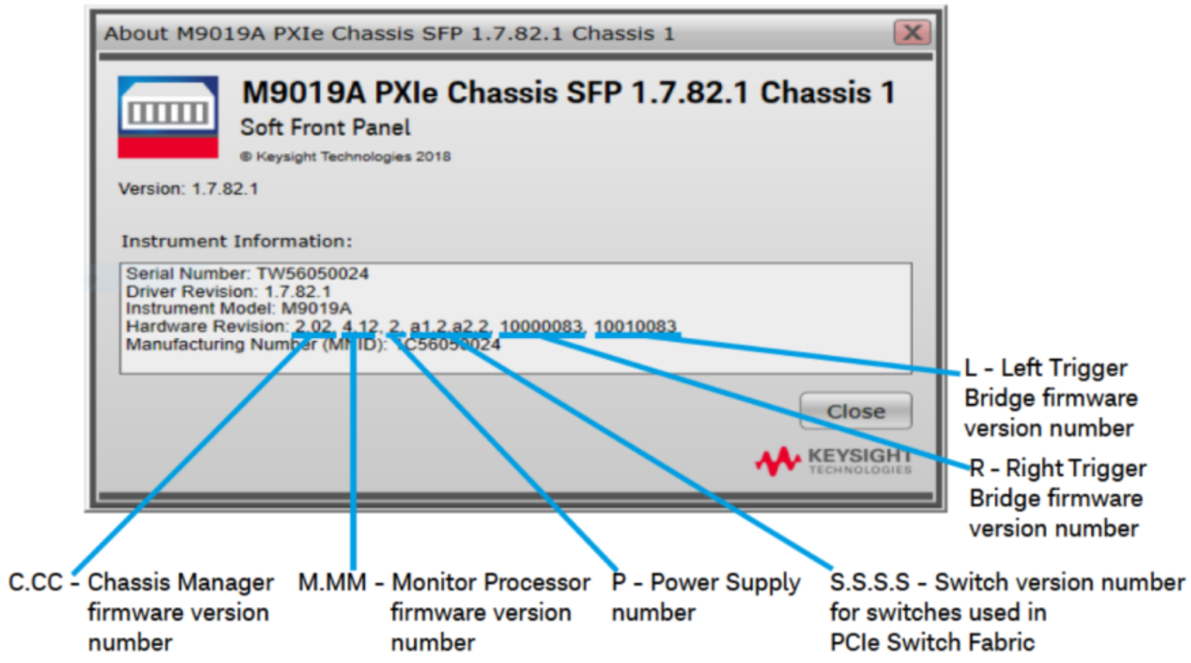
3. Keysight PXIe Instruments with FPGA firmware versions (to be installed using Keysight instrument SFP):
   - M3202A AWG FPGA firmware (v4.2.45 or later)
   - M3201A AWG FPGA firmware (v4.3.67 or later)
   - M3102A Digitizer FPGA firmware (v2.2.46 or later)
   - M9032A System Synchronization Module (v0.1.222 or later)
   - M9033A System Synchronization Module (v4.1.222 or later)

NOTE The above-mentioned list of firmware requirements includes all the Keysight PXIe instruments compatible with KS2201A. Please check the next section of this document for info about the exact instrument models necessary to run this programming example. Users can modify the example code to include additional compatible instruments.

NOTE PathWave Test Sync Executive **licenses** must be installed before running the programming example Python code. To request and install a license please consult the **PathWave Test Sync Executive User Manual** available on www.keysight.com.

How to Install Chassis Driver, SFP, and Firmware

To ensure the system compatibility described above, please install IO Libraries and chassis driver first, both are available on www.keysight.com. This programming example was tested on chassis model M9019A using the chassis driver and chassis firmware versions listed above. If you are using another chassis model, we advise you to install the same firmware version and its compatible chassis driver. When installing the Keysight Chassis Family Driver, PXIe Chassis SFP (Software Front Panel) software is automatically installed. Chassis firmware version can be checked and updated using PXIe Chassis SFP. Please see screenshots below referring to Keysight Chassis model M9019A as an example on how to check the chassis firmware version from the info in the help window of the PXIe Chassis SFP. Chassis firmware update can be performed using the Utilities window of PXIe Chassis SFP. For more info please read PXIeChassisFirmwareUpdateGuide.pdf available on www.keysight.com.


About M9019A PXIe Chassis SFP 1.7.82.1 Chassis 1

M9019A PXIe Chassis SFP 1.7.82.1 Chassis 1
Soft Front Panel
© Keysight Technologies 2018

Version: 1.7.82.1

Instrument Information:

```
Serial Number: TW56050024
Driver Revision: 1.7.82.1
Instrument Model: M9019A
Hardware Revision: 2.02, 4.12, 2, a1.2.a2.2, 10000083, 10010083
Manufacturing Number (MNID): TC56050024
```

Close
L - Left Trigger Bridge firmware version number
R - Right Trigger Bridge firmware version number
C.CC - Chassis Manager firmware version number
M.MM - Monitor Processor firmware version number
P - Power Supply number
S.S.S.S - Switch version number for switches used in PCIe Switch Fabric

M9019A Firmware Version Components

Firmware Component	2017	2018	2019StdTrig	2019EnhTrig
Chassis Manager	2.02	2.02	2.02	2.02
Monitor Processor	3.11	3.11	4.12	4.12
Switch version number for switches used in PCIe Switch Fabric	a1.2.a2.2	a1.2.a2.2	a1.2.a2.2	a1.2.a2.2
Right Trigger Bridge	0	10000083	0	10000083
Left Trigger Bridge	0	10010083	0	10010083

How to Install PathWave Test Sync Executive, SD1 SFP and M3xxxA FPGA Firmware

Note: Python 3.7.x 64-bit must be installed before installing Keysight KS2201A PathWave Test Sync Executive

After installing the chassis, the next step is to install Keysight SD1 SFP and PathWave Test Sync Executive. After installing all the necessary software, the FPGA firmware of M3xxxA PXI modules can be updated from the Hardware Manager window of the SD1 SFP. For more details on how to install SW and FPGA FW for

SD1/M3xxxA Keysight instruments, please refer to the document titled "Keysight M3xxxA Product Family Firmware Update Instructions" and the M3xxxA User Guide available on www.keysight.com

Multi-Chassis System Setup using the M9032A/M9033A PXIe System Synchronization Module

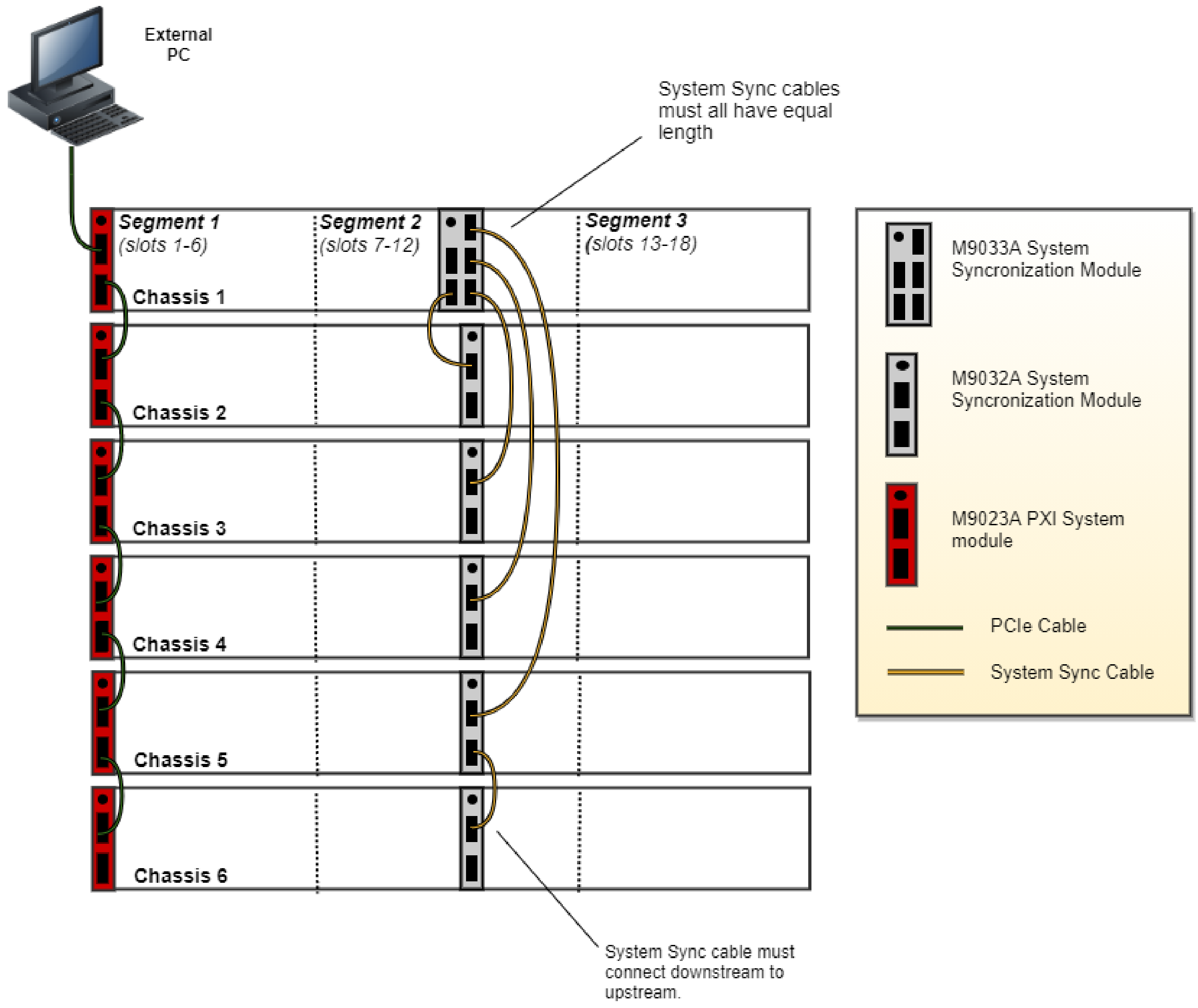
In a multi-chassis system connected with Keysight PXIe System Synchronization Modules (SSM), you must plug one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different slot number in different chassis models. SSMs are interconnected using System Sync cables.

One SSM is chosen as a leader and it is used to synchronize all the instruments included in the multi-chassis system. The SSM acting as a leader is passing the reference clock signal to the other SSMs located in the other chassis through point-to-point connections between System Sync Downstream/Upstream ports. The leader SSM is the one that has no incoming connection to his System Sync Upstream port and it only distributes on the reference clock (and other signals) from its System Sync Downstream port(s). In the example multi-chassis system depicted in the following figure, the leader SSM would be the one placed in Chassis 1.

A multi-chassis PXIe system may be configured to use many different measurement timebase reference options. For a list of those options and descriptions of how to configure them, see the section *Reference Clock Configuration* in this document. For one of those timebase reference options, one SSM is chosen as a leader and uses its internal Oven Controlled Crystal (Xtal) Oscillator (OCXO) clock to synchronize all the instruments included in the multi-chassis system.

NOTE A Multi-chassis system based on the older M9031A modules required an external reference clock generator to distribute the precise common 10 MHz reference clock signal across different chassis. In the new multi-chassis topology delivered by PathWave Test Sync Executive 2021, the SSM assumes the function of the **reference clock signal generator** , by sharing the a 100 MHz reference clock generated by an internal PLL. This PLL can be fed by different sources (as explained later in this document) including the OCXO inside the SSM, which generates a 10 MHz sine wave. An external 10 or 100 MHz reference signal can still be connected to the SSM REF Input port, to sync it together with other clusters.

The following diagram shows an example of a 6 chassis system connected with SSMs:



Programming Example Overview

In this example, Keysight PXI modular instruments are used to synchronously generate mixed signals across multiple channels on different instruments. Front Panel (FP) marker pulses can be generated in sync with arbitrary waveforms previously loaded to the AWG (Arbitrary Waveform Generator) memory thanks to the off-shelf synchronization capability provided by the HVI Synchronized Multi-Sequence Block. The mixed-signal generation can be iterated for a user-defined number of loops by using an HVI Synchronized While Statement. The duration property in HVI Sync Statements can be used to automatically set the duration of each synchronized loop.

The HVI real-time functionalities deployed to implement the use case of this application are:

1. How to create an HVI sequence using the HVI C# API.
2. Synchronized While Statement to implement synchronous measurement loops.
3. Synchronized Multi-Sequence Block to implement off-shelf synchronization capabilities
4. Duration property in HVI Sync Statements
5. Use of registers and scopes.
6. Multi-channel synchronized action execution.
7. HVI native instructions.

How to Run this Programming Example

The first step to execute this programming example is to review the configuration settings in the *ApplicationConfig.cs* file. This programming example is set up to execute in simulation mode. To execute the C# code on real HW instruments, change the option for simulated hardware to false:

```
public static bool HardwareSimulated { get; set; } = true;
```

Afterward, it is necessary to specify the actual chassis number and slot number where the real PXI instruments are located. The model number of the PXI instruments used must also be updated, if different than the instrument model used in this programming example.

This example uses PXI instruments from the Keysight M3xxx family. The first step to control such instruments is to create an object using the open() method from the SD1 API. For a complete description of the SD1 API open() method and its options please consult the SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide.

Each PXI instrument is described in the code using a module description class that contains the module model number, chassis number, slot number and options.

This programming example can be deployed on an arbitrary number of instruments to be defined using the module-descriptor class. All instruments included in the ModuleDescriptos property execute the synchronized real-time operations defined by the HVI instance. The ApplicationRole property of the ModuleDescriptor class can be used to assign to each instrument its role in the application, i.e. if it shall generate arbitrary waveforms from its AWG channels (AwgEngineName role) or marker pulses from its FP port (DioEngineName role). Please update the properties in each module-descriptor object before running the programming example:

```
public static List<ModuleDescriptor> ModuleDescriptors { get; set; } = new
List<ModuleDescriptor>
{
    new ModuleDescriptor{ ApplicationRole = DioEngineName, ModelNumber = "M3202A",
ChassisNumber = 2, SlotNumber = 4, Options = Options },
    new ModuleDescriptor{ ApplicationRole = AwgEngineName, ModelNumber = "M3201A",
ChassisNumber = 2, SlotNumber = 10, Options = Options },
    new ModuleDescriptor{ ApplicationRole = AwgEngineName, ModelNumber = "M3202A",
ChassisNumber = 2, SlotNumber = 15, Options = Options },
};
```

In the case of a multi-chassis setup, define each System Sync Module and its connections:

```
/// <summary>
/// Define list of chassis containing instruments to be used by HVI
/// </summary>
public int[] ChassisList { get; set; } = new int[] { 1, 2 };
/// Multi-chassis setup
/// Define the System Sync Modules included in your system.
/// </summary>
public static string SsmOptions { get; set; } = "DriverSetup=ForceModel=M9032A";
public static string SsmSimulationOptions { get; set; } =
"Simulate=true,DriverSetup=Model=M9033A";
```

```
public List<SystemSyncModuleDescriptor> SystemSyncModulesDescriptors { get; set; } = new
List<SystemSyncModuleDescriptor>
{
    new SystemSyncModuleDescriptor("PXI0::CHASSIS1::SLOT10::INDEX0::INSTR", SsmOptions),
    new SystemSyncModuleDescriptor("PXI0::CHASSIS2::SLOT10::INDEX0::INSTR", SsmOptions),
};
/// </summary>
/// For each SSM define which SSM is connected to its downstream connectors.
/// /// Each connectivity item is a triple (ssm1_chassis, ssm1_downstream_connector_
number, ssm2_chassis)
/// </summary>
public List<SystemSyncModuleConnection> SsmConnections { get; set; } = new
List<SystemSyncModuleConnection>
{
    new SystemSyncModuleConnection{Ssm1Chassis = 1, Ssm1DownstreamConnectorNumber = 0,
Ssm2Chassis = 2},
};
```

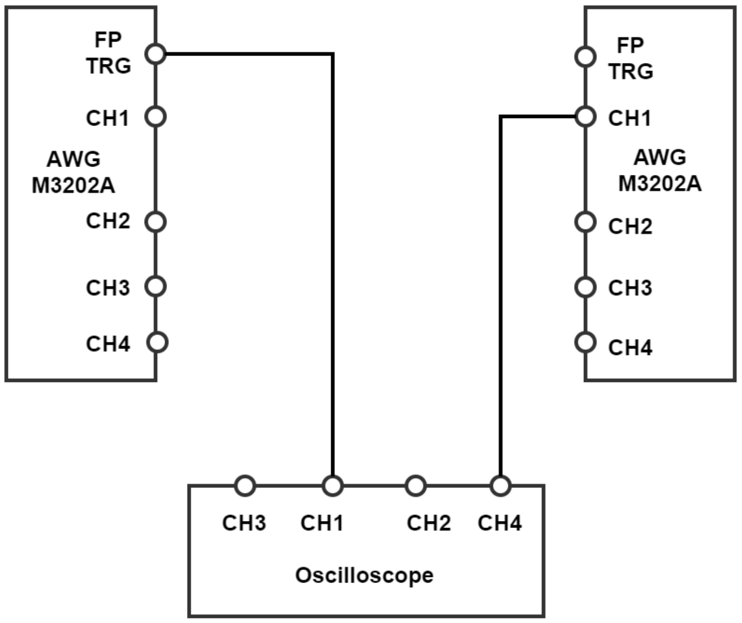
Please note that in every programming example, PXI trigger resources need to be reserved so that the HVI instance can use them for their execution. PXI lines to be assigned as trigger resources to HVI can be selected by updating the code snippet below:

```
public TriggerResourceId[] PxiSyncTriggerResources { get; set; } = new TriggerResourceId
[]
{
  TriggerResourceId.PxiTrigger3,
  TriggerResourceId.PxiTrigger4,
  TriggerResourceId.PxiTrigger5,
  TriggerResourceId.PxiTrigger6,
  TriggerResourceId.PxiTrigger7
};
```

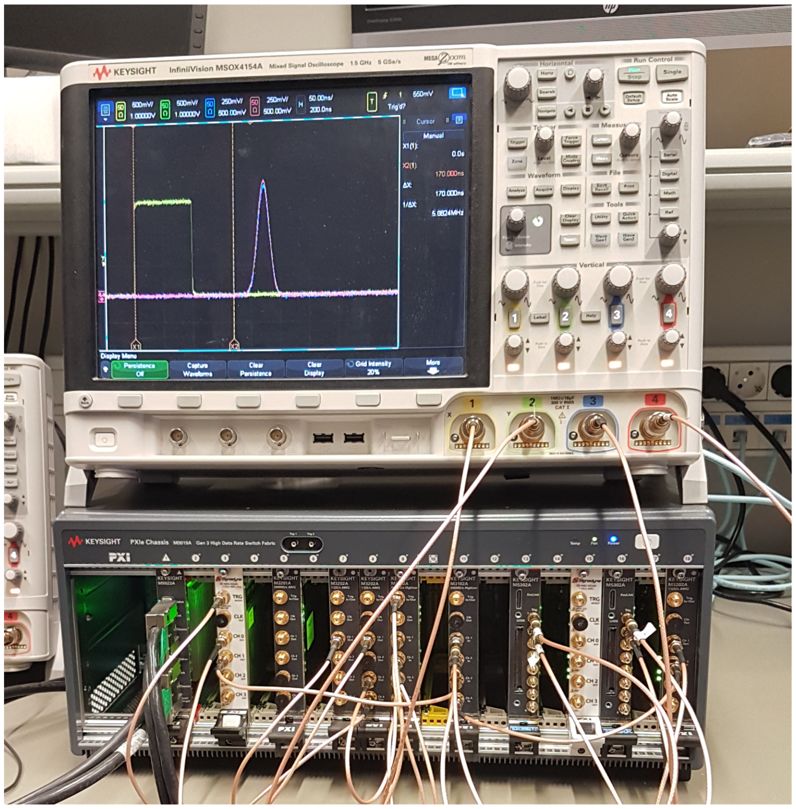
PXI lines allocated to be used as HVI trigger resources cannot be used by the programming example for other purposes. The vector pxi_sync_trigger_resources specified above must include at least the necessary number of PXI lines for the programming example to execute.

Measurement Results

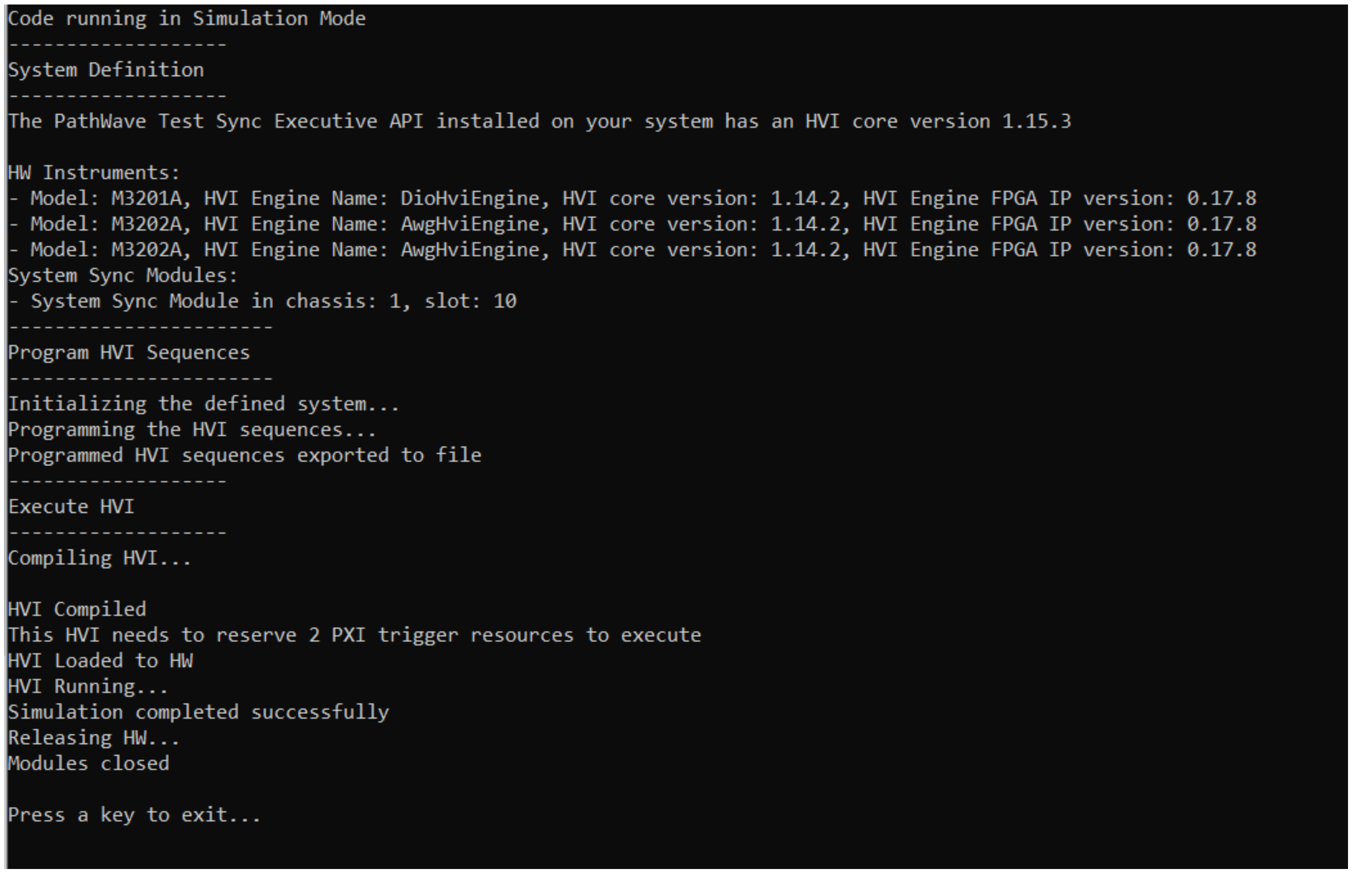
The measurement results described in this section were obtained using the measurement setup depicted below where the Front Panel (FP) connector and CH1 of two M3202A AWGs are connected to two channels of a Keysight Oscilloscope.



A photograph of the setup used for the measurement results reported in this programming example is shown below:



The screenshot below depicts the expected execution on the console window of this programming example's C# code.



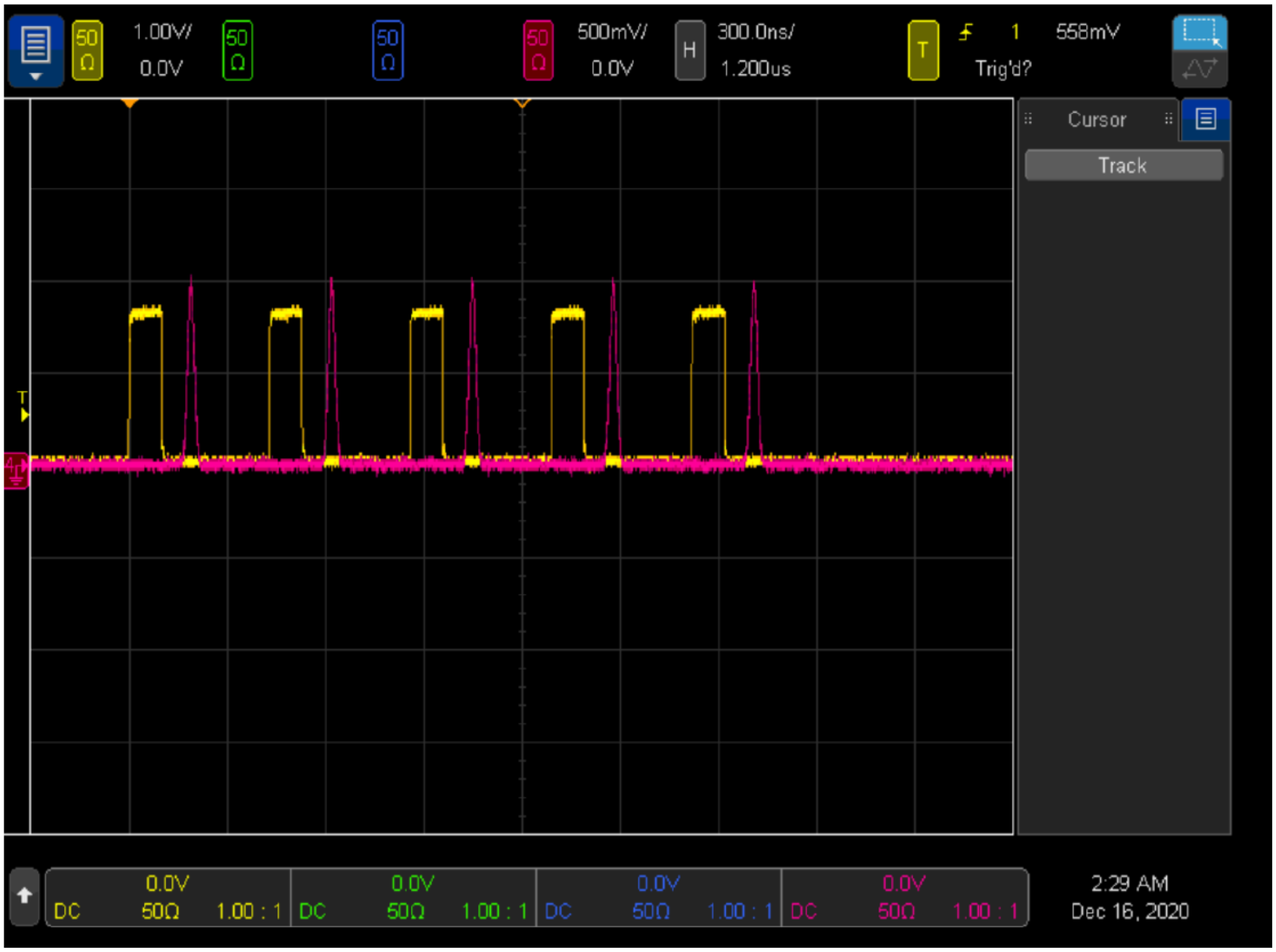
```
Code running in Simulation Mode
-------------------
System Definition
-------------------
The PathWave Test Sync Executive API installed on your system has an HVI core version 1.15.3

HW Instruments:
- Model: M3201A, HVI Engine Name: DioHviEngine, HVI core version: 1.14.2, HVI Engine FPGA IP version: 0.17.8
- Model: M3202A, HVI Engine Name: AwgHviEngine, HVI core version: 1.14.2, HVI Engine FPGA IP version: 0.17.8
- Model: M3202A, HVI Engine Name: AwgHviEngine, HVI core version: 1.14.2, HVI Engine FPGA IP version: 0.17.8
System Sync Modules:
- System Sync Module in chassis: 1, slot: 10
-----------------------
Program HVI Sequences
-----------------------
Initializing the defined system...
Programming the HVI sequences...
Programmed HVI sequences exported to file
-------------------
Execute HVI
-------------------
Compiling HVI...

HVI Compiled
This HVI needs to reserve 2 PXI trigger resources to execute
HVI Loaded to HW
HVI Running...
Simulation completed successfully
Releasing HW...
Modules closed

Press a key to exit...
```

The measurement result below shows the mixed signals that are synchronously generated by the different PXI instruments. The instrument programmed to function as a Digital I/O produces a marker pulse with a pulse width of 100 ns (yellow waveform). The second instrument is programmed to function as an arbitrary waveform generator and generates a gaussian waveform that was previously loaded to the instrument memory (red waveform). The mixed-signal generation is synchronously executed for a number of iterations that can be defined by the user. In the example measurements reported below, five signal generation loops were executed. Each mixed signal generation starts 1 us after the previous iteration because that is the value set to the duration property in the HVI SyncWhile loop.



NOTE **AWG Trigger Delay**

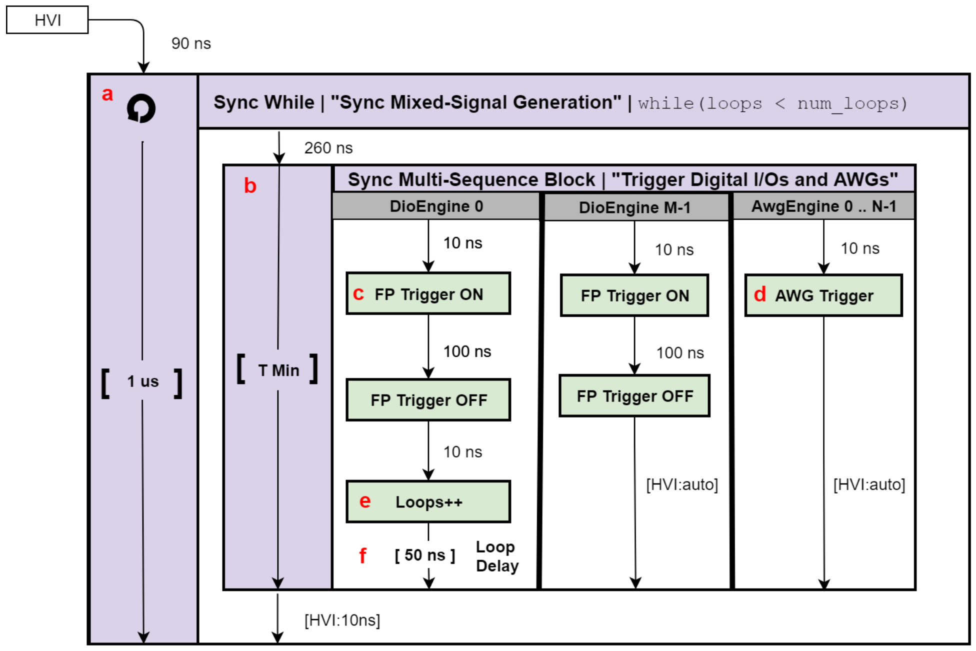
Please note, the HVI sequences represented in the HVI diagram contained in the next section specify the "AWG Trigger" instruction to happen in sync with the "FP Trigger ON" instruction. However, users must take into account that the AWG instrument requires time to process the AWG trigger action and propagate the command through its digital HW before the first waveform sample can appear at the AWG output. This processing time can be called AWG Trigger Delay, and it explains why in the previously presented scope measurements there is a delay of about 150-170 ns between the FP trigger falling edge and the first sample of the Gaussian waveform generated by the AWG. For exact values of AWG Trigger Delay and other AWG specs, please consult the documentation of Keysight M3xxx AWGs

The mixed signals can be synchronously generated by different instruments thanks to the off-shelf synchronization capabilities offered by HVI. All the details about the HVI implementation are described in the next section, together with detailed explanations of each HVI API code block.

HVI Application Programming Interface (API): Detailed Explanations

PathWave Test Sync Executive implements the next generation of HVI technology and delivers the HVI Application Programming Interface (API). This section explains how to implement the use case of this programming example using HVI API. The sequence of operations executed by each of the instruments using HVI technology is explained in the diagram below. The diagram depicts the HVI sequences executed within this programming example and the HVI statements used to program the sequences. Every HVI statement is described in detail later in this section, referencing with a letter the equivalent block in the HVI diagram and explaining in detail the corresponding HVI API code block and the HVI functionalities that it implements.

Please note that the start delays of HVI statement inserted in the following HVI diagram are set to very specific values. Unless differently specified, those values correspond to the minimum latencies that can be used for those start delays. Please consult Chapter 7 of the **PathWave Test Sync Executive User Manual** for detailed information about the timing constraint and latency of each HVI statement execution.



NOTE: Keysight M3xxxA Instruments have an FPGA clock period equal to 10 ns

NOTE The duration of each iteration of the Sync While loop used in this example is set to an arbitrary value using the *Duration* property of the SyncWhile object. The default duration of each sync statement is set to "T Min", which corresponds to the minimum duration to comply with the start delays specified by the user for each statement programmed into the local sequences contained in it.

To include HVI in an application, follow these three fundamental steps:

1. System definition: define all the necessary HVI resources, including platform resources, engines, triggers, registers, actions, events, etc.
2. Program HVI sequences: define all the statements to be executed within each HVI sequence
3. Execute HVI: compile, load to HW and execute the HVI

The following sub-sections describe in detail how these three steps are implemented for this example. For further explanations about any of the concepts, please refer to the **PathWave Test Sync Executive User Manual**.

System Definition

The definition of HVI resources is the first step of an application using HVI. The API class *SystemDefintion* enables you to define all necessary HVI resources. HVI resources include all the platform resources, engines, triggers, registers, actions, events, etc. that the HVI sequences are going to use and execute. Users need to declare them upfront and add them to the corresponding collections. All HVI Engines included in the program need to be registered into the *EngineCollection* class instance. HVI resources are described in detail in the **PathWave Test Sync Executive User Manual**. The HVI resource definitions are summarized in the code snippets below.

C#

```
var mySystem =
    DefineSystem("mySystem") // Define your system, HW platform, add HVI resources
    .AddHwPlatform(appConfig)  // Add chassis, interconnections, PXI trigger resources,
synchronization, HVI clocks
    .AddHviEngines(moduleList) // Define all the HVI engines to be included in the HVI
    .AddHviActions(moduleList, appConfig) // Define list of actions to be executed
    .AddHviTriggers(moduleList, appConfig);// Defines the trigger resources
```

Define Platform Resources: Chassis, PXI triggers, Synchronization

All HVI instances need to define the chassis and eventual chassis interconnections using the *SystemDefinition* class. In case of a multi-chassis setup, chassis interconnections using System Sync Modules can be defined using the *AddSyncModule* method of the *Interconnects* interface. PXI trigger lines to be reserved by HVI for its execution can be assigned using the *SyncResources* interface of the *SystemDefinition* class. *SystemDefinition* class also allows you to add additional clock frequencies that the HVI execution can synchronize with. For further information please consult the section "HVI Core API" of the **PathWave Test Sync Executive User Manual** .

C#

```
/// <summary>
/// Define HW platform: chassis, interconnections, PXI trigger resources,
synchronization, HVI clocks
/// </summary>
/// <param Name="mySystem">SystemDefinition object defining the system where HVI
runs.</param>
/// <param Name="appConfig">User defined configuration.</param>
/// <returns></returns>
public static SystemDefinition AddHwPlatform(this SystemDefinition mySystem,
ApplicationConfig appConfig)
{
    // Check input parameters
    AssertNotNull(mySystem, Nameof(mySystem));
    AssertNotNull(appConfig, Nameof(appConfig));
    //Add chassis resources
    // For multi-chassis setup details see programming example documentation
    foreach (int chassisNumber in appConfig.ChassisList)
    {
        if (ApplicationConfig.HardwareSimulated)
        {
            mySystem.Chassis.AddWithOptions(chassisNumber,
"Simulate=True,DriverSetup=model=M9019A");
        }
        else
        {
            mySystem.Chassis.Add(chassisNumber);
        }
    }
    // Define System Sync Modules (SSMs)
    if (appConfig.SystemSyncModulesDescriptors != null)
    {
        var interconnects = mySystem.Interconnects;
        List<Stm.ISyncModule> ssmList = new List<Stm.ISyncModule>();
        foreach (var descriptor in appConfig.SystemSyncModulesDescriptors)
        {
            if (ApplicationConfig.HardwareSimulated == true)
                ssmList.Add(interconnects.AddSyncModule(descriptor.ResourceId,
ApplicationConfig.SsmSimulationOptions));
            else
                ssmList.Add(interconnects.AddSyncModule(descriptor.ResourceId,
descriptor.Options));
        }
        // Define connections between SSMs
        if (appConfig.SsmConnections != null)
        {
            foreach (var connection in appConfig.SsmConnections)
            {
                int connectorNumber = connection.Ssm1DownstreamConnectorNumber;
                Stm.ISyncModule ssm1 = null;
                Stm.ISyncModule ssm2 = null;
                foreach (var ssm in ssmList)
                {
                    if (ssm.Chassis == connection.Ssm1Chassis)
                        ssm1 = ssm;
                    if (ssm.Chassis == connection.Ssm2Chassis)
                        ssm2 = ssm;
                }
                // Implement each user-defined connection
                try
                {
                    // Set connection. SSMs have always one upstream port
                    ssm1.Connectivity.SystemSyncDown[connectorNumber].SetConnection
(ssm2.Connectivity.SystemSyncUp[0]);
                }
                catch
                {
                    Console.WriteLine("Exception! Please check the valued defined for
SyncModule resource ids, chassis numbers and connections");
                }
            }
        }
    }
    // Assign the defined PXI trigger resources
    mySystem.SyncResources = appConfig.PxiSyncTriggerResources;
    // Assign clock frequencies that are outside the set of the clock frequencies of
each HVI engine
    // Use the code line below if you want the application to be in sync with the 10 MHz
clock
    mySystem.NonHviCoreClocks = new double[] { 10e6 };
    return mySystem;
}
```

Define HVI Engines

All HVI Engines to be included in the HVI instance need to be registered into the *EngineCollection* class instance. Each HVI Engine object added to the engine collection contains collections of its own that allow you to access the actions, events and triggers that each specific engine will control and use within the HVI.

C#

```
public static SystemDefinition AddHviEngines(this SystemDefinition mySystem,
List<SD1AwgModule> moduleList)
{
    // Check input parameters
    Program.AssertNotNull(mySystem, Nameof(mySystem));
    Program.AssertNotNull(moduleList, Nameof(moduleList));
    // For each instrument to be used in the HVI application add its HVI Engine to the
HVI Engine Collection
    foreach (var module in moduleList)
    {
        mySystem.Engines.Add(module.Instrument.Hvi.Engines.MainEngine,
module.HVIEngineName);
    }
    return mySystem;
}
```

Define HVI Actions, Events, Triggers

In this programming example, each instrument is programmed to output either a marker pulse or an arbitrary waveform at very precise instants. To do this, the FP pulse trigger and AWG trigger actions are issued from within the HVI execution. In the HVI use model, actions need to be added to the action collection of each HVI engine before they can be executed. FP trigger needs to be added to the HVI Trigger Collection and configured. This is done in this programming example as explained in the code snippets below.

C#

```
public static SystemDefinition AddHviActions(this SystemDefinition mySystem,
List<SD1AwgModule> moduleList, ApplicationConfig appConfig)
{
    // Check input parameters
    Program.AssertNotNull(mySystem, Nameof(mySystem));
    Program.AssertNotNull(moduleList, Nameof(moduleList));
    Program.AssertNotNull(appConfig, Nameof(appConfig));
    // For each AWG, define the list of HVI Actions to be executed and add such list to
its own HVI Action Collection
    foreach (var module in moduleList)
    {
        for (var chIndex = 1; chIndex <= module.NumChannels; chIndex++)
        {
            // Actions need to be added to the engine's action list so that they can be
executed
            string actionName = string.Format("{0}{1}", appConfig.AwgTriggerName,
chIndex);  // arbitrary user-defined Name
            int actionId;
            switch (chIndex)
            {
                case 2:
                    actionId = module.Instrument.Hvi.Actions.Awg2Trigger;
                    break;
                case 3:
                    actionId = module.Instrument.Hvi.Actions.Awg3Trigger;
                    break;
                case 4:
                    actionId = module.Instrument.Hvi.Actions.Awg4Trigger;
                    break;
                default:
                    actionId = module.Instrument.Hvi.Actions.Awg1Trigger;
                    break;
            }
            mySystem.Engines[module.HVIEngineName].Actions.Add(actionId, actionName);
        }
    }
    return mySystem;
}

public static SystemDefinition AddHviTriggers(this SystemDefinition mySystem,
List<SD1AwgModule> moduleList)
{
    // Check input parameters
    AssertNotNull(mySystem, Nameof(mySystem));
    AssertNotNull(moduleList, Nameof(moduleList));
    // Add to the HVI Trigger Collection of each HVI Engine the FP Trigger object of
that same instrument
    foreach (var module in moduleList)
    {
        int fpTriggerId = module.Instrument.Hvi.Triggers.FrontPanel1;
        ITriggerDefinition fpTrigger = mySystem.Engines
[module.HVIEngineName].Triggers.Add(fpTriggerId, ApplicationConfig.FpTriggerName);
        // Configure FP trigger in each Hvi Engine
        fpTrigger.Config.Direction = Direction.Output;
        fpTrigger.Config.Polarity = Polarity.ActiveHigh;
        fpTrigger.Config.SyncMode = SyncMode.Immediate;
        fpTrigger.Config.HwRoutingDelay = 0;
        // FP TriggerMode is set to Level, which does not defines a pulse length
        // FP trigger pulse length is defined by the HVI Statements that control FP
Trigger ON/OFF
        fpTrigger.Config.TriggerMode = TriggerMode.Level;
    }
            return mySystem;
        }
    }
}
```

Program HVI Sequences

HVI sequences can be programmed using the *Sequencer* class. HVI starts the execution through a global sequence (defined by the *SyncSequence* class) that takes care of synchronizing and encapsulating the local sequences corresponding to each HVI engine included in the application. In this programming example, the HVI global sync sequence contains only one sync statement, a synchronized multi-sequence block defined by the API class *SyncMultiSequenceBlock.*

C#

```
public static Sequencer ProgramMixedSignalSequence(this Sequencer sequencer,
List<SD1AwgModule> moduleList, ApplicationConfig appConfig)
{
    // Check input parameters
    Program.AssertNotNull(sequencer, Nameof(sequencer));
    Program.AssertNotNull(moduleList, Nameof(moduleList));
    Program.AssertNotNull(appConfig, Nameof(appConfig));
    // Add register
    IRegister loops = sequencer.SyncSequence.Scopes.First().Registers.Add("Loops",
RegisterSize.Short);
    // SyncWhile condition
    IConditionTerm syncWhileCondition = Condition.RegisterComparison(loops,
ComparisonOperator.LessThan, appConfig.NumLoops);
    // Add a Sync While
    ISyncWhileStatement syncWhile = sequencer.SyncSequence.AddSyncWhile("Sync Mixed-
Signal Generation", 90, syncWhileCondition);
    // Add a Sync Multi-Sequence Block (SMSB)
    ISyncMultiSequenceBlockStatement syncBlock =
syncWhile.SyncSequence.AddSyncMultiSequenceBlock("Trigger Digital I/Os and AWGs", 170);
    // Program the SMSB to trigger AWGs and FP pulses
    ProgramMimoTrigger(syncBlock, moduleList, appConfig);
    return sequencer;
}
```

Define HVI Registers

HVI registers correspond to very fast access physical memory registers in the HVI Engine located in the instrument HW (e.g. FPGA or ASIC). HVI Registers can be used as parameters for operations and modified during the sequence execution (same as Variables in any programming language). The number and size of registers is defined by each instrument. The registers that users want to use in the HVI sequences need to be defined beforehand into the register collection within the scope of the corresponding HVI Sequence. This can be done using the RegisterCollection class that is within the Scope object corresponding to each sequence. HVI Registers belong to a specific HVI Engine because they refer to HW registers of that specific instrument. Registers from one HVI Engine cannot be used by other engines or outside of their scope. Note that currently, registers can only be added to the HVI top SyncSequence scopes, which means that only global registers visible in all child sequences can be added. HVI registers are defined in this programming example by the code snippet below.

C#

```
// Add register
IRegister loops = sequencer.SyncSequence.Scopes.First().Registers.Add("Loops",
RegisterSize.Short);
```

Synchronized While (a)

This corresponds to statement (a) in the HVI diagram. Synchronized While (Sync While) statements belong to the set of HVI Sync Statements and are defined by the API class *SyncWhile* . A Sync While allows you to synchronously execute multiple local HVI sequences until a user-defined condition is met, that is, the sync while condition. Please note that for local sequences to be defined within the Sync While, it is necessary to use synchronized multi-sequence blocks. The duration of each iteration of the Sync While loop can be set using the *Duration* property and the *Time* class. Please note that the duration cannot be set to a deterministic quantity if the Sync While contains any flow control statement, i.e. If, While, Wait or WaitTime statements. Please consult Chapter 7 of the KS2201A User Manual for further information.

C#

```
// SyncWhile condition
IConditionTerm syncWhileCondition = Condition.RegisterComparison(loops,
ComparisonOperator.LessThan, appConfig.NumLoops);
// Add a Sync While
ISyncWhileStatement syncWhile = sequencer.SyncSequence.AddSyncWhile("Sync Mixed-Signal
Generation", 90, syncWhileCondition);
// Define Sync While Duration
syncWhile.Duration = new Time.Duration(1, Time.Unit.Microseconds);
```

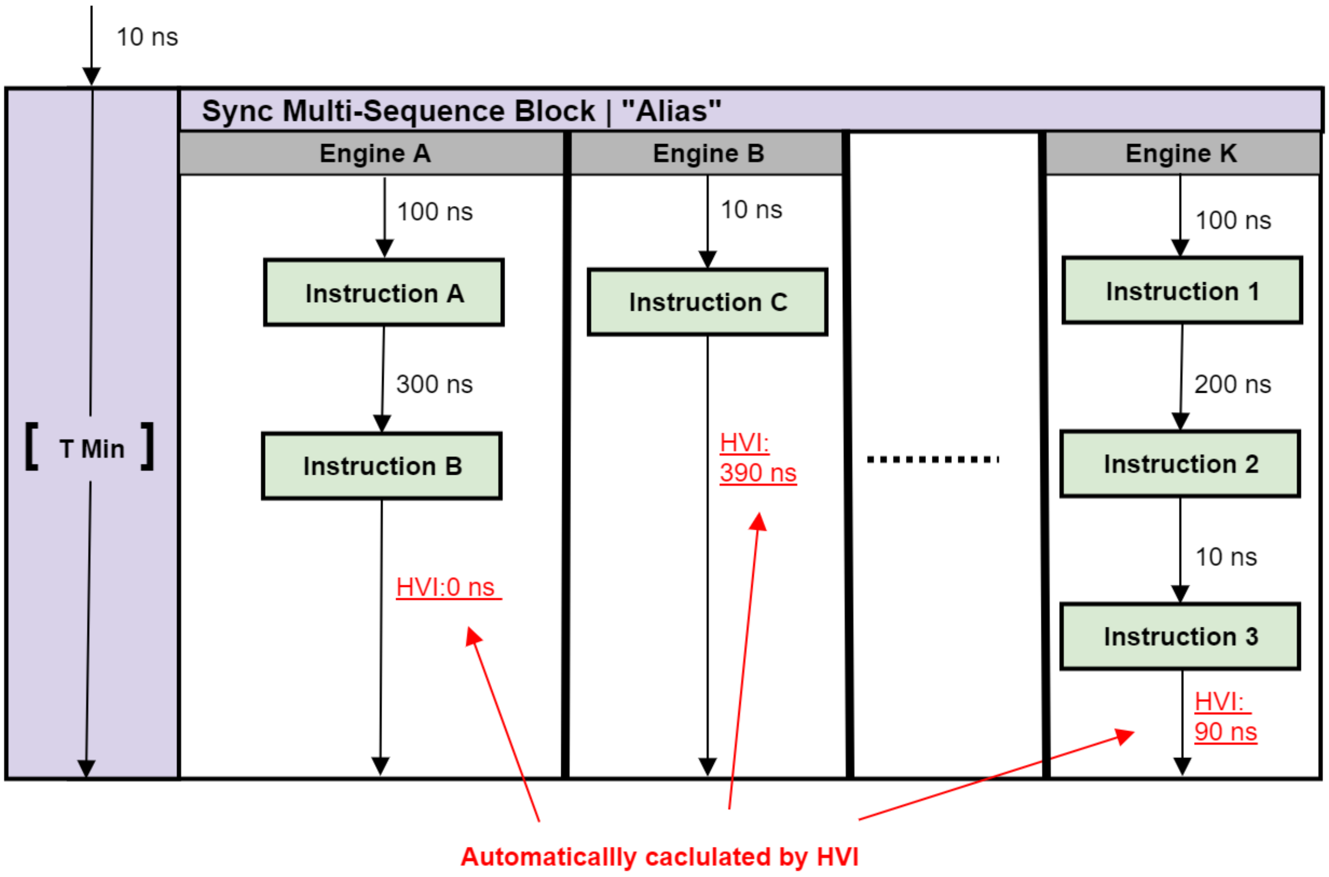
Synchronized Multi-Sequence Block (b)

This block synchronizes all the HVI engines that are part of the sync sequence and enables the user to program each HVI Engine to do specific operations by exposing a local sequence for each engine. By calling the API method A*ddMultiSequenceBlock()* a synchronized multi-sequence block is added to the Sync (global) Sequence. The duration of the Sync Multi-Sequence Block (SMSB) can be set using the *Duration* property and the *Time* class. In this example the SMSB duration is set to minimum, which means that the SMSB will last according to the start delays specified by the user for each statement programed into the local sequences contained in it. Please note that the duration cannot be set to a deterministic quantity if the SMSB contains any flow control statement, i.e. If, While, Wait or WaitTime statements. Please consult Chapter 7 of the KS2201A User Manual for further information.

C#

```
// Add a Sync Multi-Sequence Block (SMSB)
ISyncMultiSequenceBlockStatement syncBlock =
syncWhile.SyncSequence.AddSyncMultiSequenceBlock("Trigger Digital I/Os and AWGs", 260);
// Define SMSB Duration
syncBlock.Duration = new Time.Minimum();
```

Within the Synchronized Multi-Sequence Block (SMSB), users can define which statement each local engine is going to execute in parallel with the other engines. Local HVI sequences start and end synchronously their execution within the sync multi-sequence block. Users can define the exact amount of time each local HVI statement starts to execute with respect to the previous one. HVI automatically calculates the execution time of each local sequence and adjusts the execution of all local sequences within the multi-sequence block so that they can deterministically end altogether within the synchronized multi-sequence block. See the general case example in the figure below for additional details.



<u>NOTE:</u> Keysight M3xxxA Instruments have an FPGA clock period equal to 10 ns

Please note that the Sync Multi-Sequence Block has an execution duration time labeled as "T Min" in the figure above. The "T Min" default value for any sync statement corresponds to the minimum time necessary to complete the operations included inside. KS2201A Update 1.0 release provides the *Duration* property in Sync Statement objects that allows users to set an arbitrary duration value larger than "T Min". The timing at the end of each local sequence is automatically adjusted by HVI according to the duration specified by the user for the SMSB. In the case of duration "T min", HVI will automatically add no time to the local sequence with the longest duration and adjust the other sequences accordingly, as in the example depicted in the figure above. The resolution for HVI-defined time adjustment at the end of a sync multi-sequence block corresponds to the 10 ns FPGA clock period for an application including instruments that are all within the Keysight M3xxx family. For further explanations about the timing of HVI sequence execution please refer to the **KS2201A PathWave Test Sync Executive User Manual** available on www.keysight.com

HVI Instruction: Front Panel Trigger ON/OFF (c)

This block executes a native HVI instruction. Native HVI instructions are common to every Keysight product. The API method *add_instruction()* allows you to add the wanted instruction within the HVI sequence. Instruction parameters are set using the API method *SetParameter()* . All HVI Native instructions and parameters are defined in the *hvi.InstructionSet* interface.

C#

```
// Retrieve FP Trigger prefiously defined in the HVI Trigger Collection
ITrigger fpTrigger = sequence.Engine.Triggers[appConfig.FpTriggerName];
// Retrive TriggerWrite instruction from HVI Native InstructionSet
IInstructionTriggerWrite triggerWrite = sequence.InstructionSet.TriggerWrite;
// Write FP Trigger ON
IInstructionStatement instrTriggerOn = sequence.AddInstruction("FP Trigger ON", 10,
triggerWrite.Id);
instrTriggerOn.SetParameter(triggerWrite.Trigger.Id, fpTrigger);
instrTriggerOn.SetParameter(triggerWrite.SyncMode.Id, triggerWrite.SyncMode.IMMEDIATE);
instrTriggerOn.SetParameter(triggerWrite.Value.Id, triggerWrite.Value.ON);
```

Action Execute: AWG Trigger (d)

Actions to be used within an HVI sequence need to be added to the instrument HVI engine using the API "Add" method of the *ActionCollection* class. Once the wanted actions are added within the list of the instruments' HVI engine actions, an instruction to execute them can be added to the instrument's HVI sequence using the HVI API class *InstructionsActionExecute* . One or multiple actions can be executed at the same time within the same "Action Execute" instruction.

C#

```
// Execute AWG trigger from the HVI sequence of each module
// "Action Execute" instruction executes the AWG trigger from HVI
IInstructionStatement instr = sequence.AddInstruction("AWG trigger", 10,
sequence.InstructionSet.ActionExecute.Id);
instr.SetParameter(sequence.InstructionSet.ActionExecute.Action.Id,
sequence.Engine.Actions.ToArray());
```

Register Increment (e)

This type of instruction can be found in statements   (e). A register increment can be implemented within an HVI sequence using an instance of the API instruction class *InstructionsAdd* . The same instruction can be used to add registers and constant values (operands) and put the result in another register (result). The register to be incremented was previously defined in the scope of the corresponding HVI engine.

C#

```
// Retrieve register object
var loops = sequence.Scope.Registers["Loops"];
// Increment loop counter
var instruction = sequence.AddInstruction("Loops++", 10,
sequence.InstructionSet.Add.Id);
instruction.SetParameter(sequence.InstructionSet.Add.Destination.Id, loops);
instruction.SetParameter(sequence.InstructionSet.Add.LeftOperand.Id, loops);
instruction.SetParameter(sequence.InstructionSet.Add.RightOperand.Id, 1);
```

Delay Statement (f)

This type of statement can be found in statements (f). Inserting an instance of *DelayStatement* class causes an HVI sequence to wait for a fixed amount of time that is known at compilation time and it is not expected to change during HVI execution. The amount of time is specified in nanoseconds. The Delay Statment functions like the start delay parameter used in each method that programs a statement into an HVI sequence. The main difference is that a start delay allows specifying a delay before a statement, whereas the delay statement allows to specify it afterward, for example at the end of a Sync Multi-Sequence Block, as it is used in this programming example. To specify a Variable delay that can change during HVI execution, one shall use the WaitTime statement instead.

C#

```
// Add a delay statement to allow the register increment to complete its execution
var instrDelay = sequence.AddDelay("Delay", 100);
```

Export the Programmed HVI Sequences to File

KS2201A provides a feature to export the programmed HVI sequences, which can be used both as a development and debug tool. The sequences can be exported using the *ToString()* method of the SyncSequence class, as illustrated in the code snippet below. An example text file containing the HVI sequences exported from this programming example is provided together with this example's files.

```
// Generate HVI sequence description text
var output = sequencer.SyncSequence.ToString(OutputFormat.Debug);
```

Compile, Load, Execute the HVI Instance

Once the HVI sequences are programmed by defining all the necessary HVI statements, you can compile, load and execute the HVI. Compile, load and run functionalities can be accessed from the *Hvi* class.

Compile HVI

The compilation operation is performed by calling the compile() API method. This operation processes all the info related to the HVI application, including the necessary HVI resources and the HVI statements included in the HVI sequences. The compilation generates a binary compiled output that can be loaded to the hardware instruments for their HVI engine to execute it. As an output, the compile() API method provides an object that can tell the user how many PXI sync resources are necessary to be reserved to execute the HVI application.

C#

```
// Compile HVI sequences
var hvi = sequencer.Compile();
Console.WriteLine("HVI Compiled");
Console.WriteLine("This HVI application needs to reserve {0} PXI trigger resources to
execute", hvi.CompileStatus.SyncResources.Count());
```

Load HVI to Hardware

The API method load_to_hw() loads to each HVI engine the binary output obtained from the HVI compilation so that the HVI engine programmed into their digital HW (FPGA or ASIC) can execute it.

C#

```
// Load HVI to HW: load sequences, configure actions/triggers/events, lock resources,
etc.
hvi.LoadToHw();
```

Execute HVI

HVI execution is controlled by the run() API method. HVI can be run in a blocking or non-blocking mode. In this programming example, the blocking mode is used. In this mode the SW execution is blocked at the HVI execution code line for a fixed amount of time specified by the timeout input parameter. The SW execution can be blocked until the HVI sequences finish their execution if timeout = hvi.no_timeout is used as an input parameter.

C#

```
// Execute HVI in blocking mode: SW waits until HVI sequences ends their execution
// Eventually enter a timeout for the HVI execution to be stopped: timeout = timedelta
(seconds=0), hvi.run(timeout)
hvi.Run(IHvi.NoTimeout);
```

Release Hardware

API method release_hw() shall be called once the HVI execution is finished to release all the HW resources that were reserved during the HVI execution, including the PXI trigger resources that had been locked by HVI for its execution.

C#

```
// Release HW resources once HVI execution is completed
hvi.ReleaseHw();
```

Further HVI API Explanations

Detailed explanations of each class and functionality of the HVI API can be found in the PathWave Test Sync Executive User Manual or in the C# help file that is provided with the HVI installer.

Conclusions

This programming example explained how to use Pathwave Test Sync Executive and HVI (Hard Virtual Instrument) technology to synchronously generate mixed signals from multiple PXI instruments. Each instrument can be configured to generate a marker pulse or a previously loaded arbitrary waveform. The programming example use case illustrated here can be tested on any AWG of the Keysight M3xxxA PXI family. HVI technology was deployed using the HVI API (Application Programming Interface). Example measurement results demonstrated synchronized multi-channel mixed-signal generation with sub-ns precision.