PROGRAMMING EXAMPLE 6

# Synchronized MIMO Measurements using M5302A Digital I-O and M3xxxA PXI Instruments

PATHWAVE

In this programming example, PathWave Test Sync Executive is used to program multiple M5302A Digital I/O (DIO) and M3xxxA PXI instruments. By using HVI (Hard Virtual Instrument) capabilities, DIO instruments can output a pulsed signal from any of their Front Panel (FP) SMB trigger ports and M320xA AWGs can synchronously play a previously queued waveform. Multiple M3102A Digitizers can also be included in the same HVI to synchronously capture all the generated analog and digital signals. This way the example can showcase a Multiple-Input Multiple-Output (MIMO) measurement setup having all his input and output channels fully synchronized.


KEYSIGHT TECHNOLOGIES

# Table of Contents

## KS2201A - Programming Example 6 - Synchronized MIMO Measurements using M5302A Digital I-O and M3xxxA PXI Instruments

In this programming example, PathWave Test Sync Executive is used to program multiple M5302A Digital I/O (DIO) and M3xxx PXI instruments. By using HVI (Hard Virtual Instrument) capabilities, DIO instruments can output a pulsed signal from any of their Front Panel (FP) SMB trigger ports and M320xA AWGs can synchronously play a previously queued waveform. Multiple M3102A Digitizers can also be included in the same HVI to synchronously capture all the generated analog and digital signals. This way the example can showcase a Multiple-Input Multiple-Output (MIMO) measurement setup having all his input and output channels fully synchronized.

# 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 Python 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 the 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).

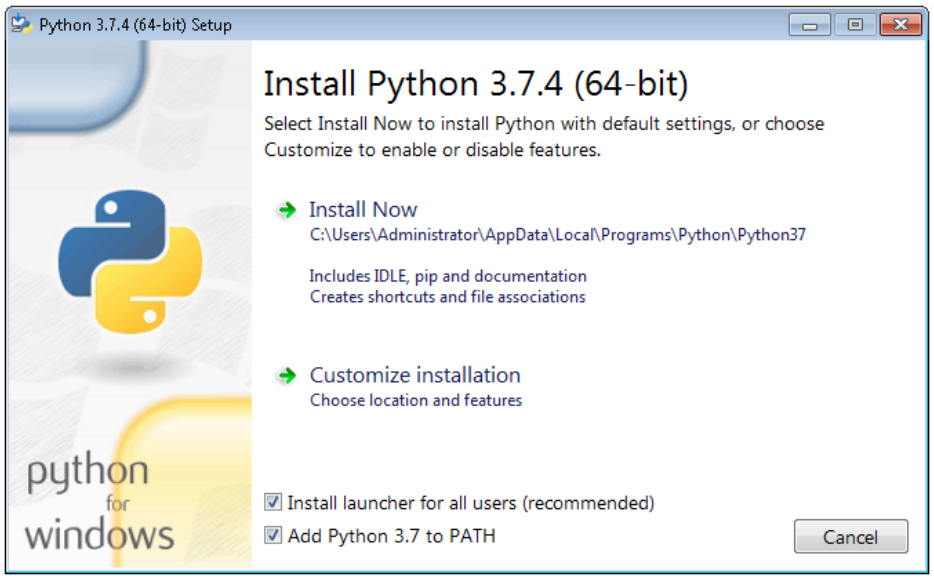
1. Software versions required:
   - Python 3.x 64-bit (or later), including Python packages time, numpy, matplotlib
   - Keysight IO Libraries Suite 2021 (v18.2.27115.0 or later)
   - Keysight SD1 Drivers, Libraries and SFP (v3.3.12 or later)
   - Keysight M5302A Drivers, Libraries and SFP (v1.0.11616 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.

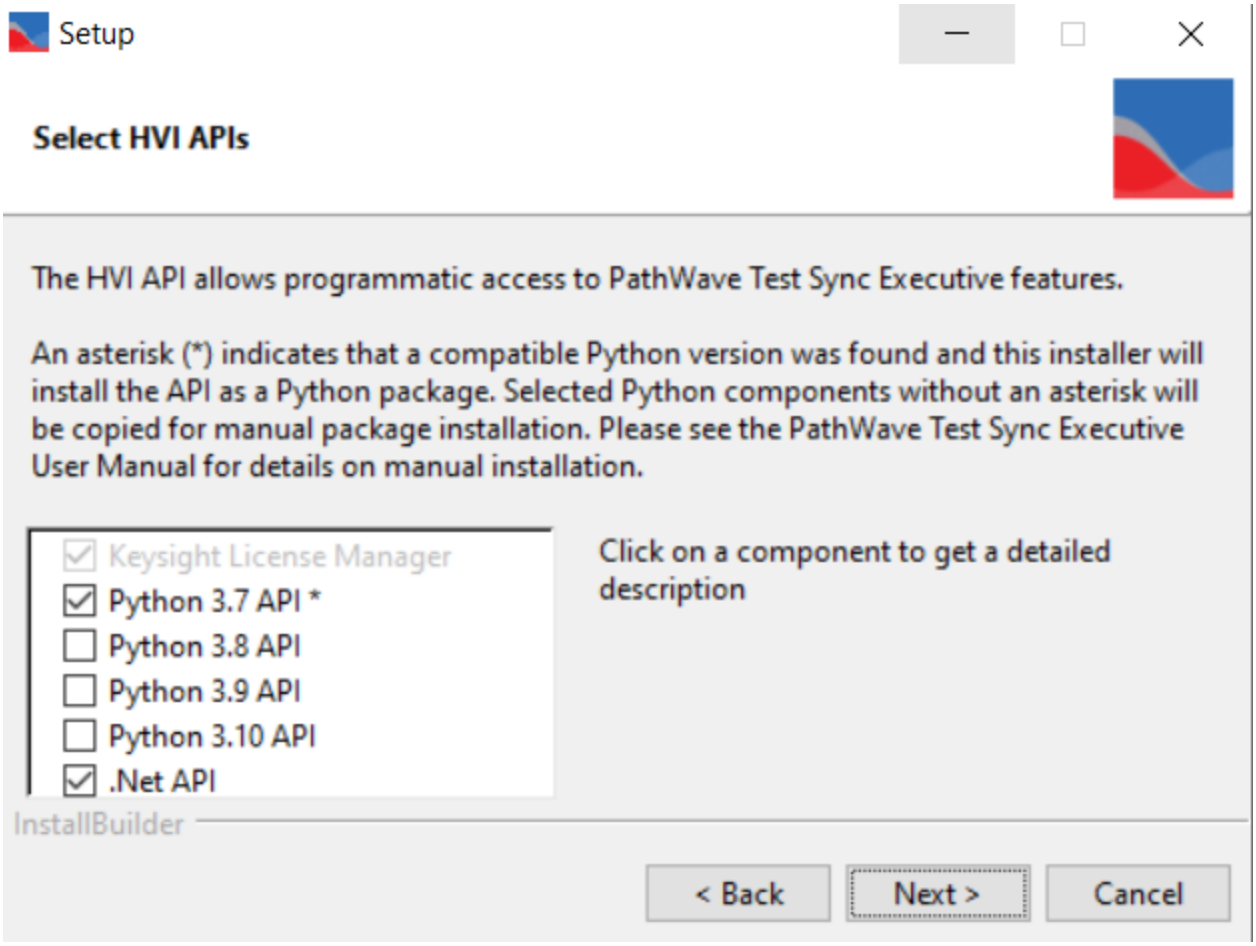
NOTE PathWave Test Sync Executive **licenses** must be installed before running the programming example Python code. To request and install a license please consult the **PathWave Test Sync Executive User Manual** available on www.keysight.com.

## How to Install Python 3.x 64-bit

This programming example requires you to install Python 64-bit version equal or greater than 3.7.x for all users. The Python installer can be downloaded from the Python official webpage https://www.python.org. Make sure you add Python 3.x to the PATH system Variable. This can be done at the installation step by checking the right check-boxes as shown in the screenshot below.




Once Python is installed, you can install KS2201A. When running the KS2201A installer, it will detect which Python 3.x 64-bit is installed in your system and is compatible with the `keysight_hvi` package delivered by the installer. The detected compatible version(s) will appear with a check in its checkbox. In the screenshot example below the Python 3.7 API is checked and will be installed. If you wish to install other instances of the `keysight_hvi` package, compatible with other Python 3.x 64-bit versions, then please manually check other additional checkboxes at this step of the installation procedure.

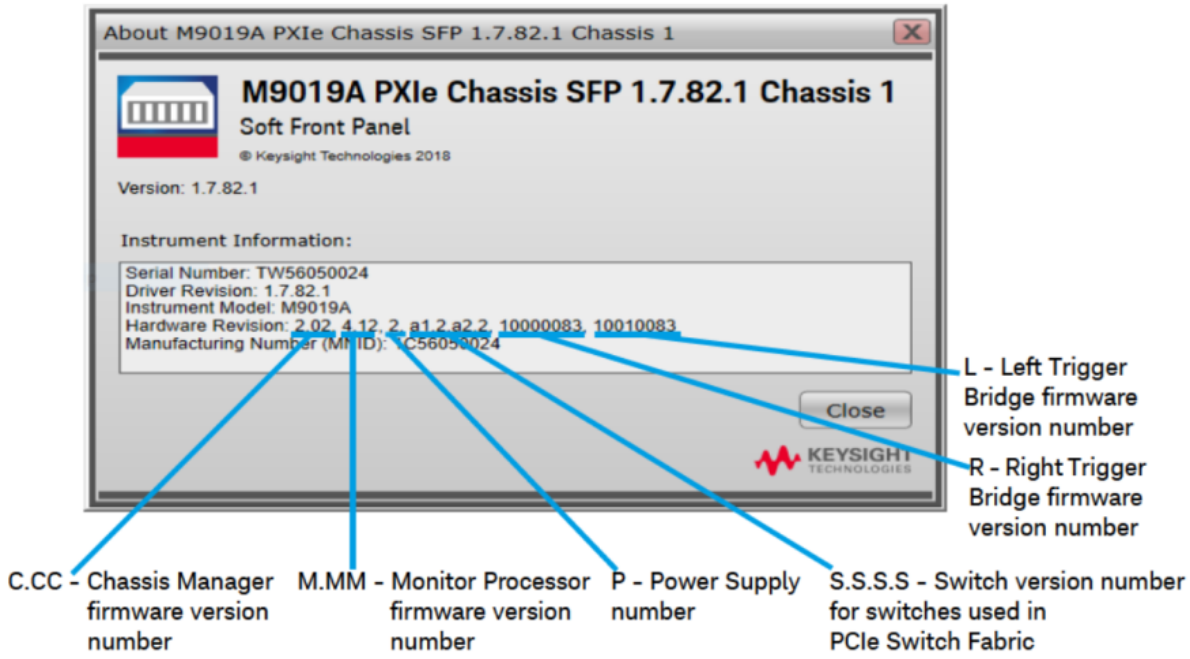



NOTE PathWave Test Sync Executive programming examples require the Python packages ***time*** , ***numpy*** and ***matplotlib*** . These packages can be installed using the Python package installer pip. For more information about pip and how to use it, please visit https://pypi.org/project/pip/.

NOTE Users installing Python through a distribution that is different than the one available from the Python official webpage https://www.python.org (e.g. Anaconda distribution) need to make sure that their PATH environment Variable includes the path to set up the HVI API Python library. This can be done by adding to the programming example Python code a line that includes that path, for example: sys.path.append(C:\Program Files\Keysight\PathWave Test Sync Executive <year>\api\python) where year shall be replaced with the year of the release you are using, for example <year> = 2021.

## 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 .



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

## How to Install KF9000B PathWave FPGA

The 3rd and 8th programming examples include PathWave FPGA project files designed using **KF9000B PathWave FPGA 2021**. To install and obtain a license for KF9000B PathWave FPGA 2021 (or a later version) please consult the product webpage on www.keysight.com. PathWave FPGA also requires Xilinx Vivado software to run. For further information please consult the PathWave FPGA User Manual on www.keysight.com.

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

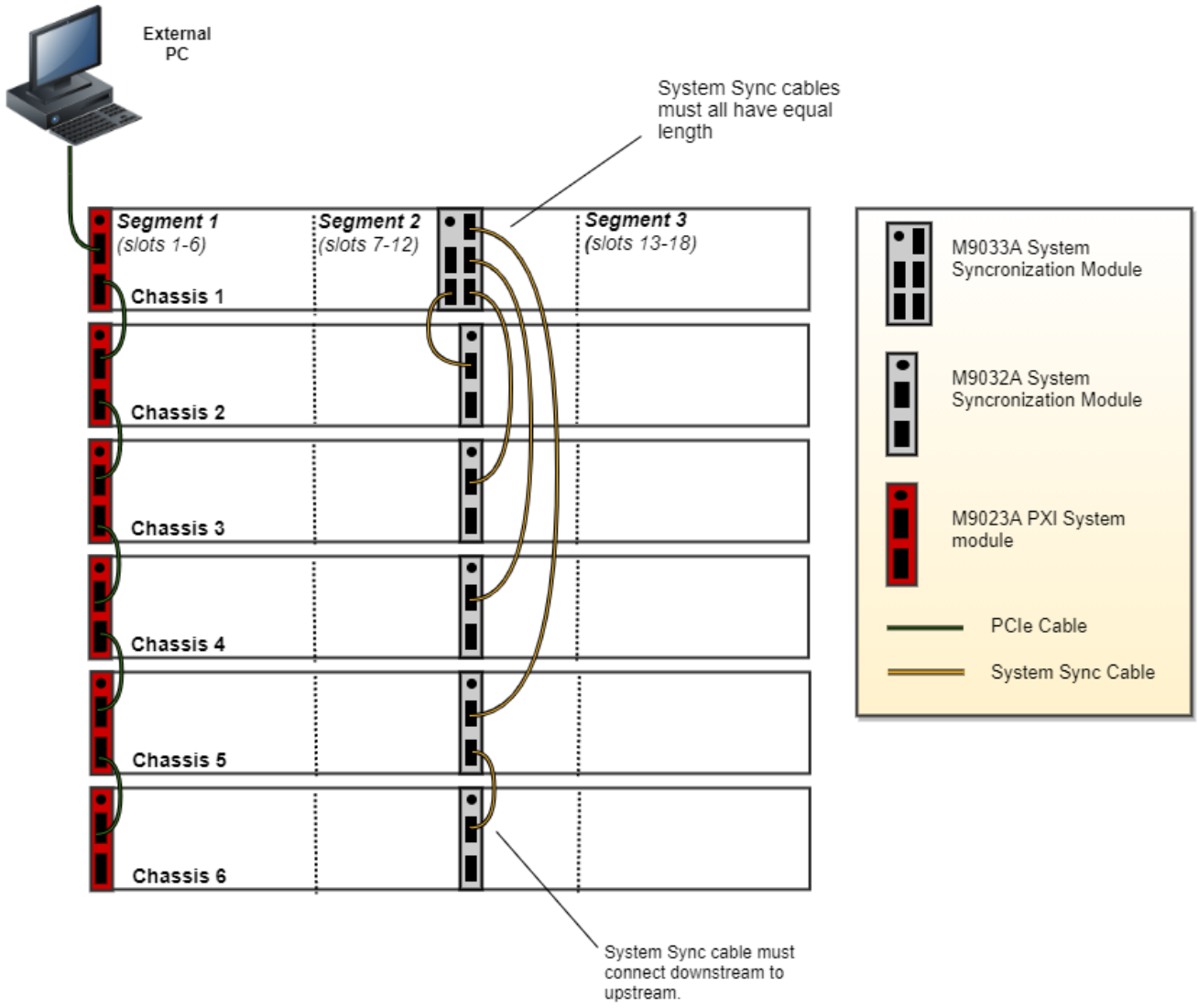
In a multi-chassis system connected with Keysight PXIe System Synchronization Modules (SSM), you must plug one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different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




For further information please refer to the Quick Start Guide: Multi-Chassis System Setup available on www.keysight.com.

# Programming Example Overview

This programming example illustrates how to deploy HVI to synchronously generate electronic signals from multiple channels across an arbitrary number of different instruments. The instruments used in this programming example are M320xA AWGs (Arbitrary Waveform Generators) and Digital I/O instruments. The example targets MIMO (Multiple-Input Multiple-Output) use case scenarios including MIMO transceiver testing for 5G (5th Generation) telecommunications and multi-qBit (quantum bit) experiments for quantum engineering.

This programming example illustrates the following HVI functionalities:

1. Synchronization of M5302A Digital I/Os and M3xxx PXI instruments using Synchronized Multi-Sequence Blocks.
2. Duration property in HVI Sync Statements
3. Execution of HVI Native Instructions and HVI Actions
4. Synchronized multiple trigger write functionality

## How to Run this Programming Example

This programming example is set up to execute in simulation mode. To execute the Python code on real HW instruments, change the option for simulated hardware to False:

```
# Simulated HW Option
hardware_simulated = 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 used PXI instruments must be updated if it is different from the instrument model used in this programming example. This example uses PXI instruments from the Keysight M3xxx and M5xxx families. The first step to control such instruments is to create an object using the instrument API. For a complete description of the instrument object creation options, please consult the SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide and the M5302A PXIe Digital IO Module 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 (or resource ID) 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 Python code execute the synchronized real-time operations defined by the HVI instance. Please update the properties in each module-descriptor object before running the programming example:

```
# Define module descriptors below with your instruments information
self.awg_module_descriptors = [
    AwgModuleDescriptor('M3202A', 1, 3, self.sd1_options, self.awg_engine_Name),
    AwgModuleDescriptor('M3202A', 1, 18, self.sd1_options, self.awg_engine_Name)]
self.dio_module_descriptors = [
    DioModuleDescriptor('PXI0::34-0.0::INSTR', self.dio_query, self.dio_reset, self.dio_
options, self.dio_model_number, self.dio_engine_Name),
    DioModuleDescriptor('PXI0::42-0.0::INSTR', self.dio_query, self.dio_reset, self.dio_
options, self.dio_model_number, self.dio_engine_Name)]
```

```
class ModuleDescriptor:
"Descriptor for module objects"
def __init__(self, model_number, chassis_number, slot_number, options, engine_Name):
self.model_number = model_number
self.chassis_number = chassis_number
self.slot_number = slot_number
self.options = options
self.engine_Name = engine_Name
```

The chassis to be used in the programming example must be specified and listed by chassis number:

```
# Update list of chassis numbers included in the programming example
self.chassis_list = [1, 2]
```

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

```
# Multi-chassis setup
# Define the System Sync Modules included in your system.
self.ssm_options = ''
self.ssm_simulation_options = 'Simulate=true,DriverSetup=Model=M9033A'
self.system_sync_modules_descriptors = [
    SystemSyncModuleDescriptor('PXI0::CHASSIS1::SLOT10::INDEX0::INSTR', self.ssm_
options),
    SystemSyncModuleDescriptor('PXI0::CHASSIS2::SLOT10::INDEX0::INSTR', self.ssm_
options)]
# 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)
self.ssm_connections = [
    SystemSyncModuleConnection(ssm1_chassis=1, ssm1_downstream_connector_number=1, ssm2_
chassis=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:

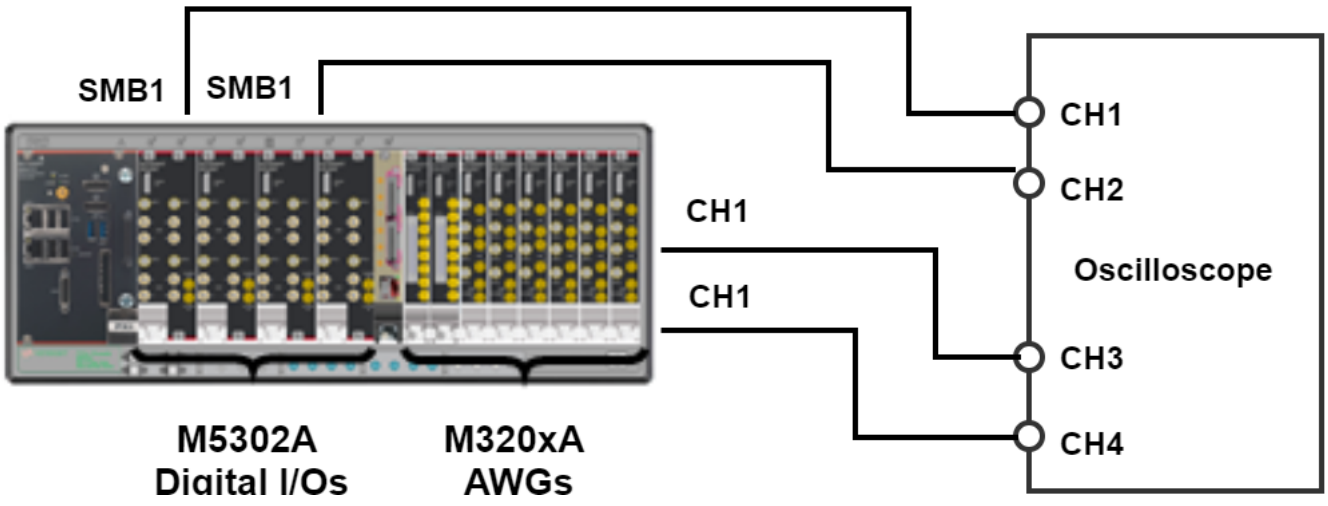
```
# Assign triggers to HVI object to be used for HVI-managed synchronization, data
sharing, etc # NOTE: In a multi-chassis setup ALL the PXI lines listed below need to be
shared among each M9031 board pair by means of SMB cable connections
self.pxi_sync_trigger_resources = [ kthvi.TriggerResourceId.PXI_TRIGGER0,
kthvi.TriggerResourceId.PXI_TRIGGER1, kthvi.TriggerResourceId.PXI_TRIGGER2,
kthvi.TriggerResourceId.PXI_TRIGGER3]
```

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. Please check the programming example code for the actual number of PXI lines that needs to be reserved. The HVI compiler also returns, for a given HVI sequence, the number of necessary PXI lines that must be reserved.

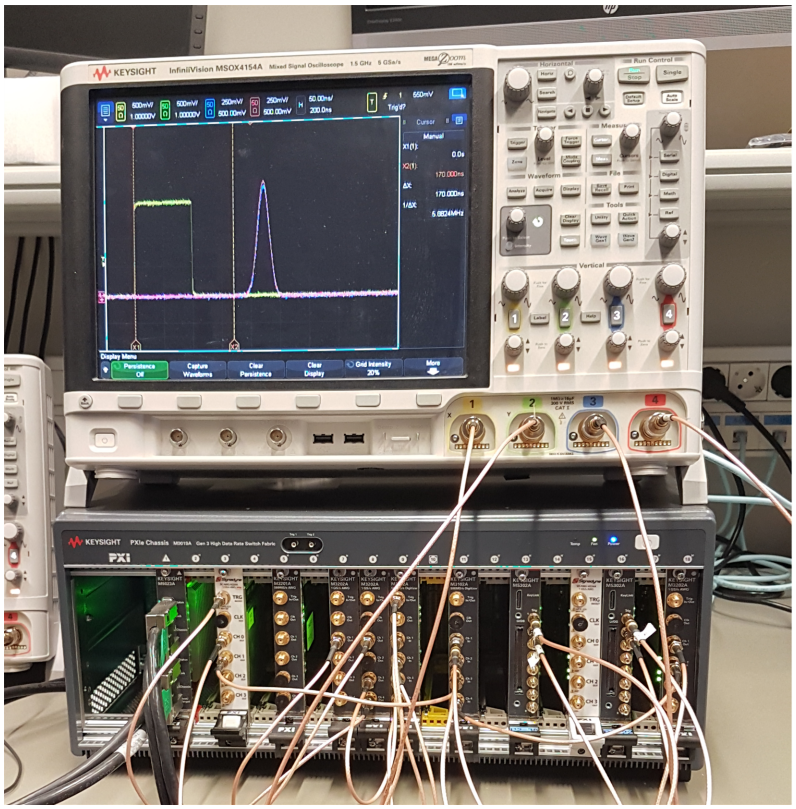
## Measurement Results

This programming example contains a simple HVI global sync sequence that executes a sync multi-sequence block on an arbitrary number of AWG instruments. All local sequences are synchronously executed by all instruments' HVI engines to first trigger a pulse from the front panel TRG port and then output a waveform from all the AWG channels.

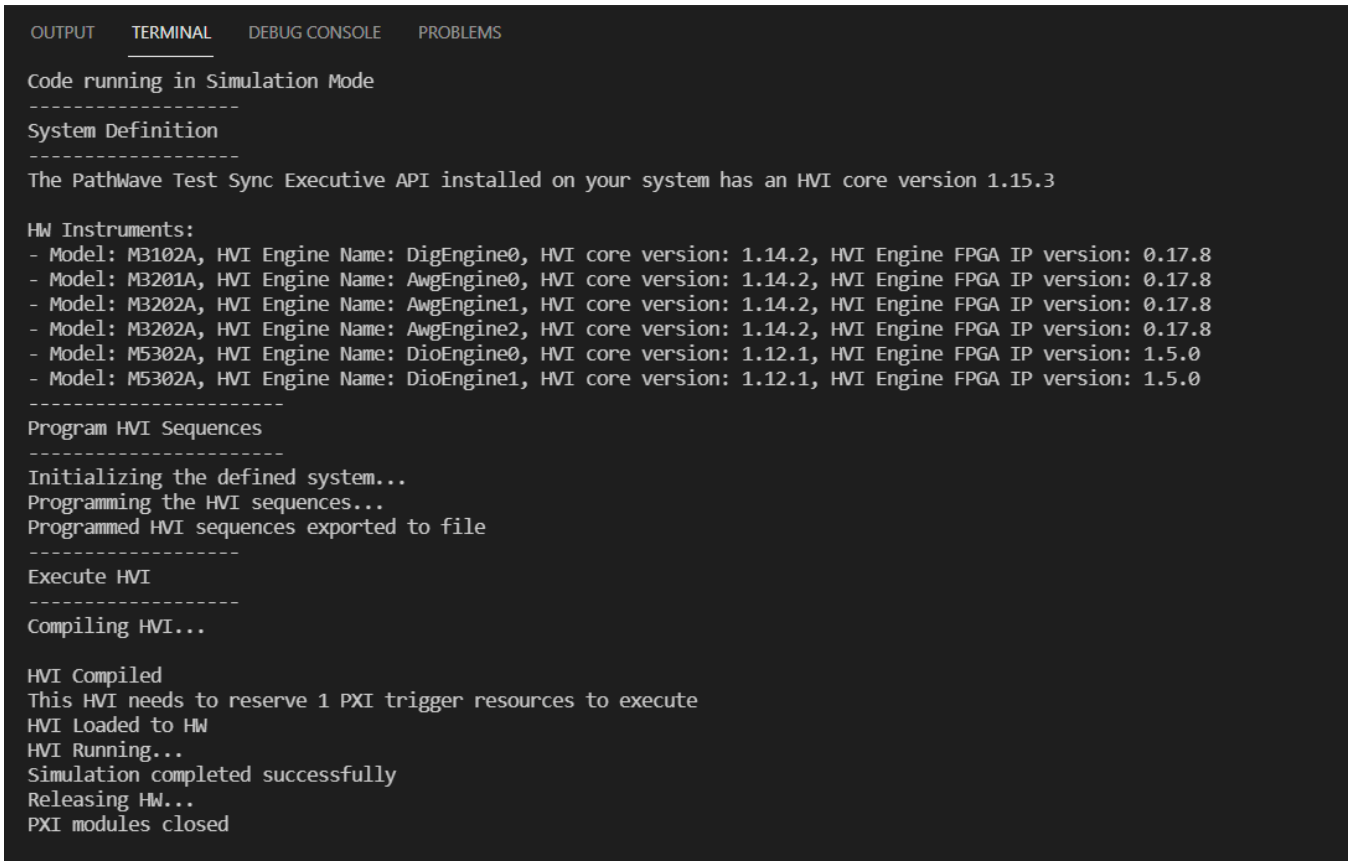
The programming example capabilities are illustrated through some example measurement results obtained using the measurement setup depicted below where the Front Panel (FP) connector and CH1 of two M3202A AWGs are connected to the four channels of a Keysight Oscilloscope.



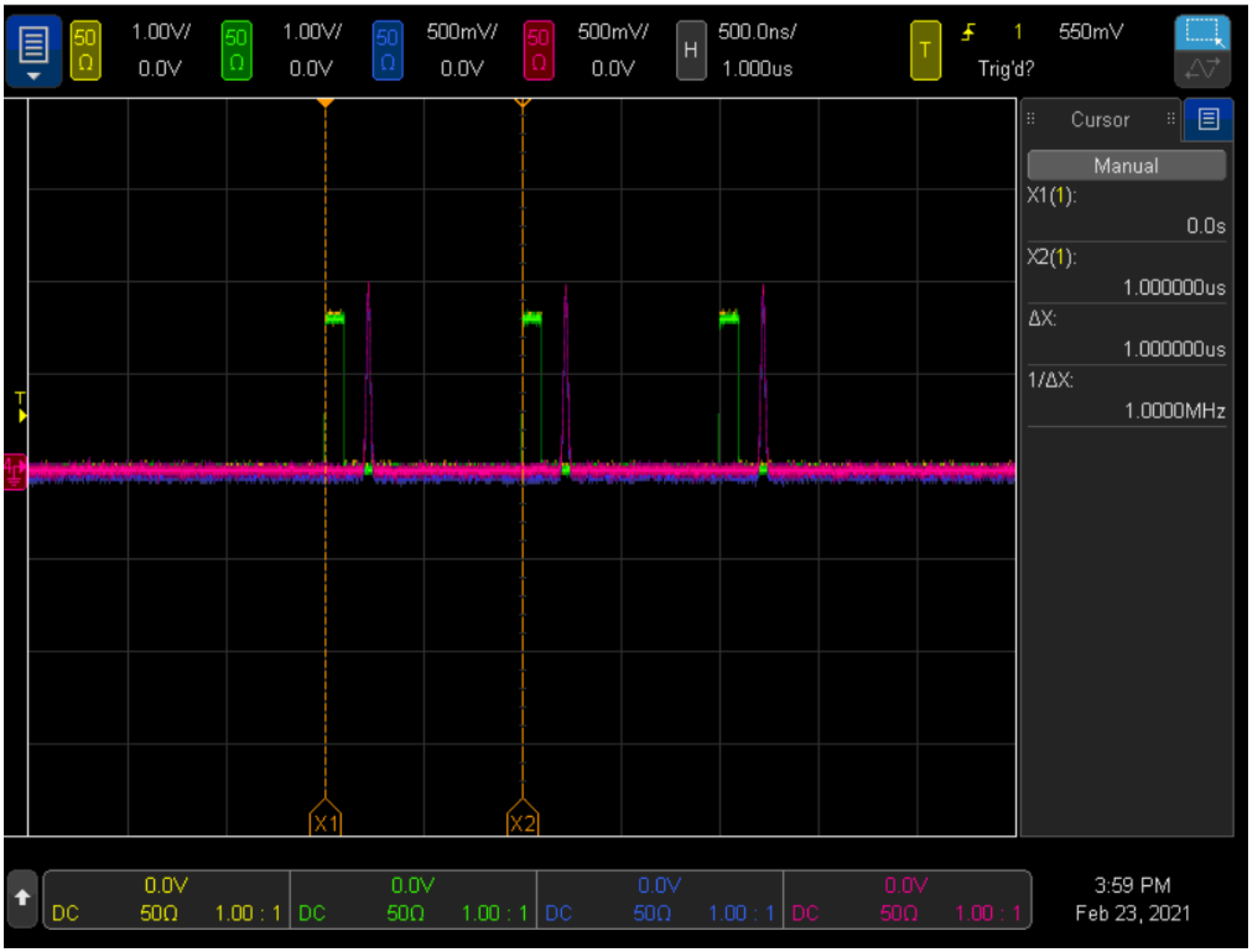
A photograph of the measurement setup used for the measurement results reported in this programming example is also reported below:

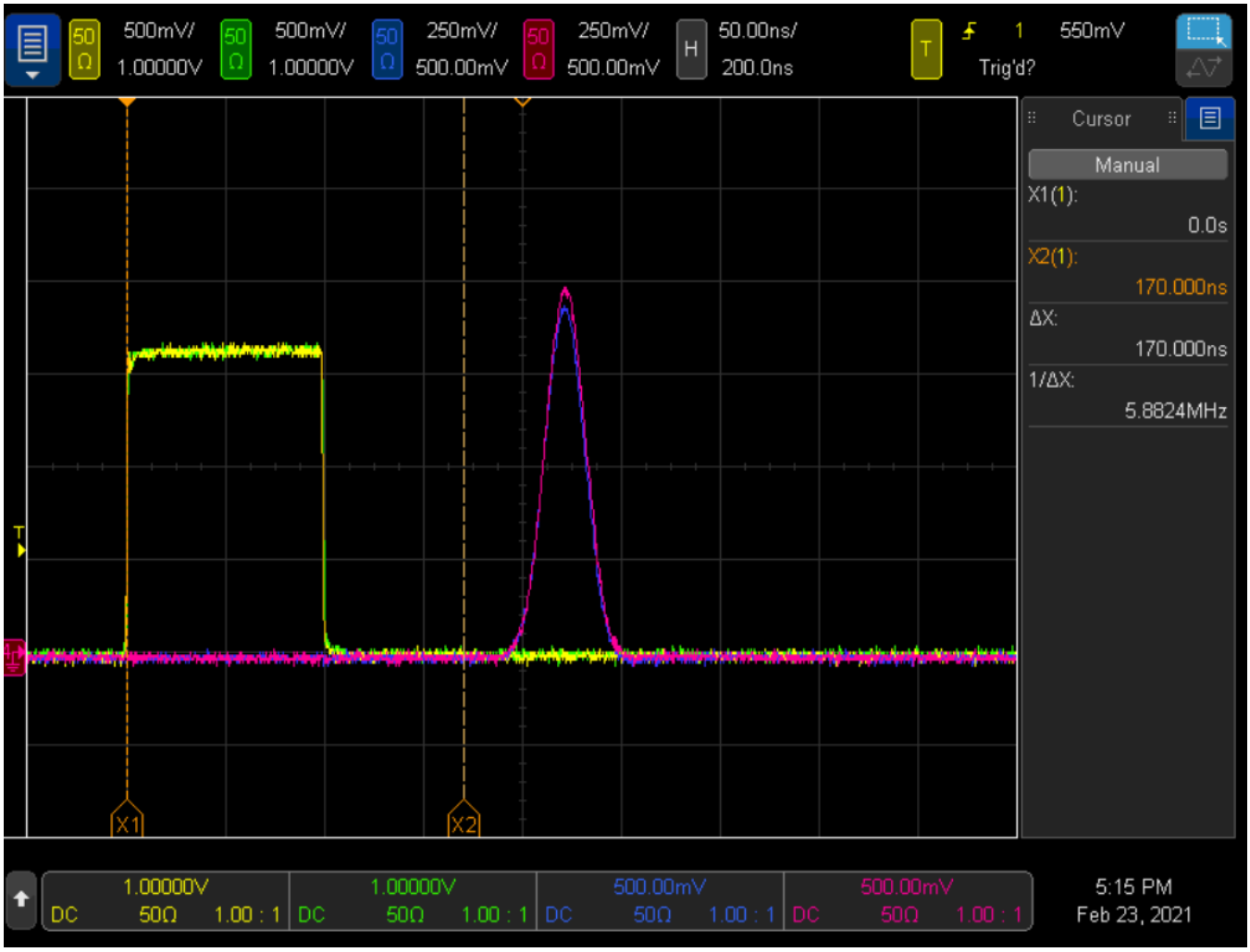


The screenshot below depicts the expected execution on the console window of this programming example's Python code.



By running the Python code, the example measurements depicted below were obtained. The scope measurements below show measurement results obtained using two AWG M3202A with HV1 option. In the scope measurement, we can observe the two synchronized FP trigger pulses (yellow and green waveforms) output in a synchronized manner by two independent M5302A Digital I/O instruments. The FP trigger pulses are followed by two waveforms (red and blue waveforms) triggered by the "AWG Trigger" action executed from the HVI sequence. The execution can be repeated for a number of synchronized loops that can be configured by the user by setting the num_loops property of the ApplicationConfig class. Each loop iteration can be configured using the loop_duration property of the ApplicationConfig class, which was set to 1 us to obtain the example measurement results in the figure below.

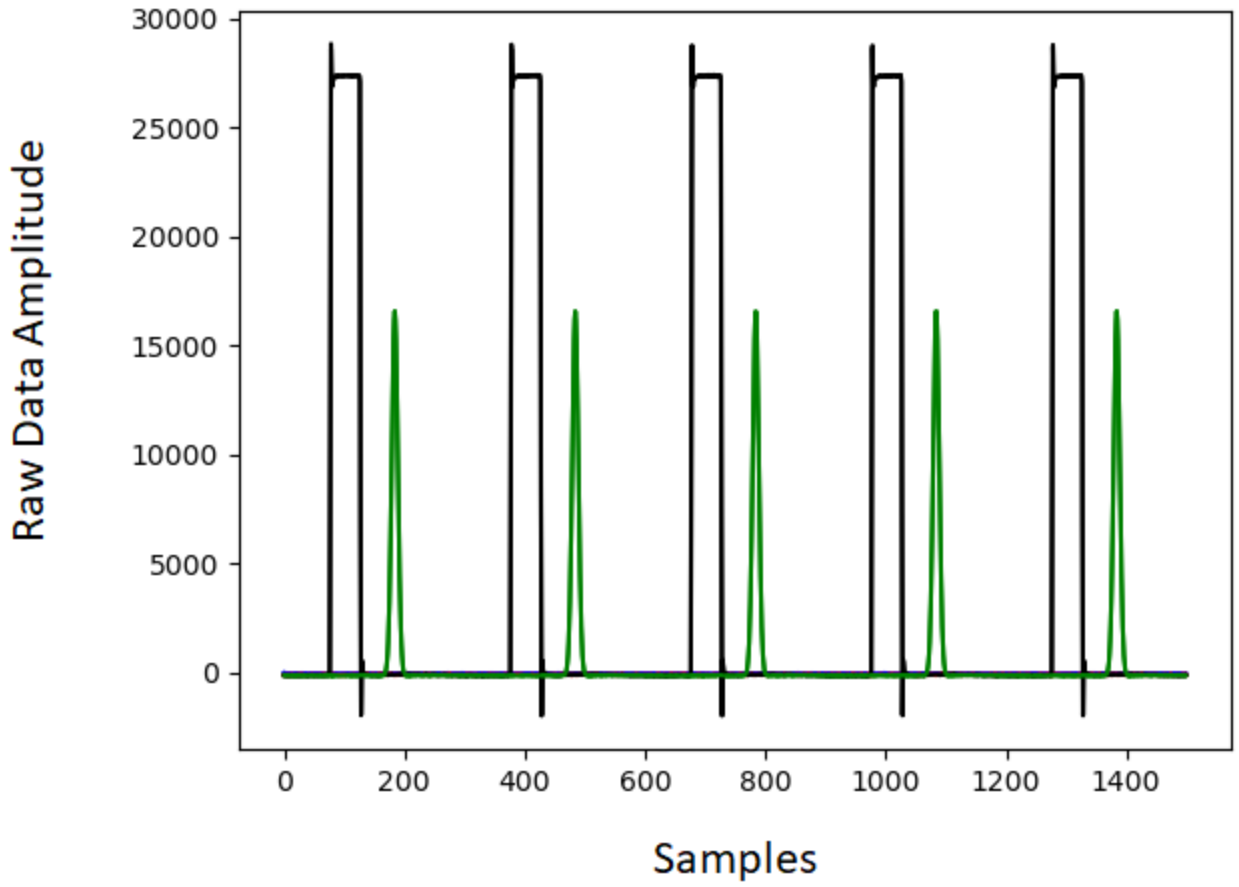





NOTE **AWG Trigger Delay**

Please note, the HVI sequences represented in the HVI diagram contained in the next section specifies the "AWG Trigger" instruction to be executed by all AWG instruments in sync with the "SMB Triggers ON" instruction executed by Digital I/Os. 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 170 ns between the FP SMB trigger rising 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 available on www.keysight.com.

This programming example includes the possibility to include M3102A PXI digitizers into the HVI so that each digitizer channel can be triggered within the HVI SyncWhile loop and capture an acquisition cycle in sync with the AWGs or Digital I/Os that are outputting the analog or digital waveforms in the same measurement loop iteration. The figure below shows an example digitizer acquisition configured with 5 acquisition cycles of 300 acquisition points for each cycle. Each acquisition cycle is triggered by a "DAQ Trigger" action execution controlled by HVI.



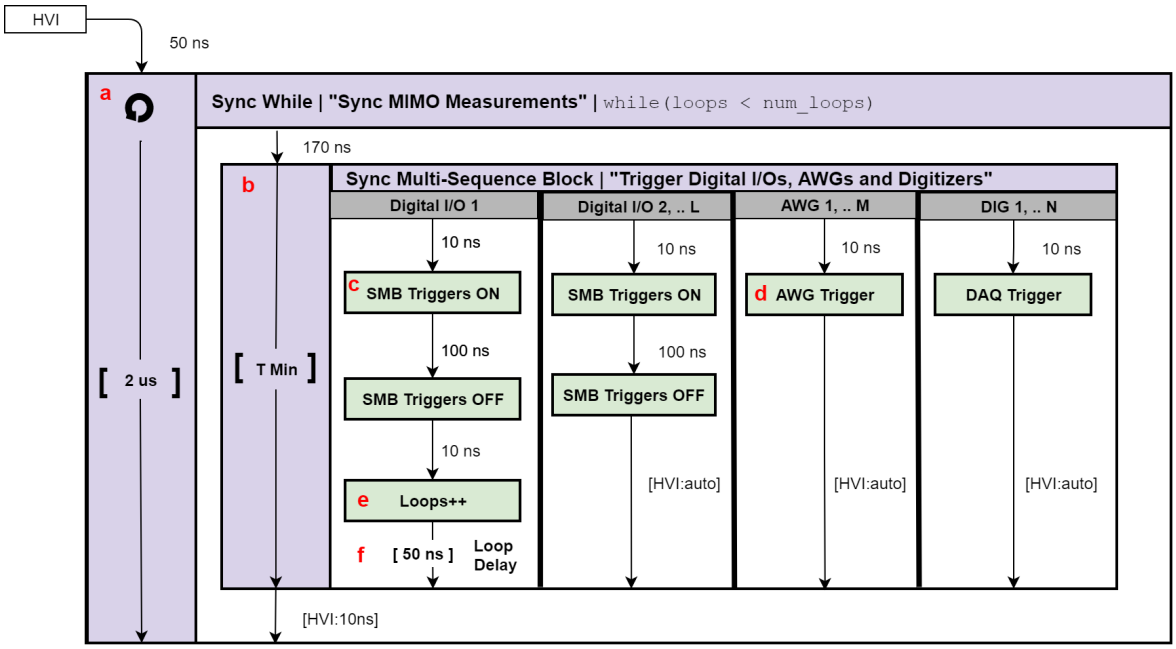

NOTE **Digitizer Trigger Re-Arm Latency**

Please note that the HVI diagram contained in the next section specifies the "DAQ Trigger" instruction to be executed by each Digitizer (in its local sequence), in sync with the "FP Triggers ON" and "AWG Trigger" instructions executed by the Digital I/O and AWG respectively. Two consecutive "DAQ Trigger" actions cannot happen if the digitizer has not re-armed the acquisition trigger logic. For the M3102A Digitizer such Trigger Re-Arm Latency was found to be around 1.8 us and that is why the duration of the "Sync MIMO Measurements" SyncWhile is set to 2 us. For exact values of AWG Trigger Delay and other AWG specs, please consult the documentation of Keysight M3xxx Digitizers available on www.keysight.com.

# 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:** Keysight M5xxxA Instruments have an FPGA clock period equal to 1/(300 MHz) = 3.333... 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.

NOTE HVI allows specifying the statement in each engine local sequence with a resolution corresponding to the FPGA clock period of that HVI engine, which is 10 ns for the M3xxx PXI instruments and 3.333... ns for M5xxx instruments . HVI Sync statements instead, can use start delays that are integer multiples of the Least Common Multiple (LCM) of the FPGA clock periods of all the HVI Engines included in the HVI, which corresponds in this case to LCM(10, 10/3) = 10 ns

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 up front and add them to the corresponding collections. All HVI Engines included in the programming 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.

**Python**

```
# Create system definition object
my_system = kthvi.SystemDefinition("MySystem")
```

```
def define_hvi_resources(config, sys_def, awg_module_dict, dio_module_dict):
    """    Configures all the necessary resources for the HVI application to execute: HW
platform, engines, actions, triggers, etc.
    """    # Define HW platform: chassis, interconnections, PXI trigger resources,
synchronization, HVI clocks
    define_hw_platform(sys_def, config)
    # Define all the HVI engines to be included in the HVI
    define_hvi_engines(sys_def, awg_module_dict, dio_module_dict)
    # Define list of actions to be executed
    define_hvi_actions(sys_def, awg_module_dict, config)
    # Defines the trigger resources
    define_hvi_triggers(sys_def, dio_module_dict, config)
```

## Define Platform Resources: Chassis, PXI triggers, Synchronization

All HVI instances need to define the chassis and eventual chassis interconnections using the *SystemDefinition* class. System Sync Modules can be defined using the *add_sync_module* method of the interconnects interface. PXI trigger lines to be reserved by HVI for its execution can be assigned using the *sync_resources* interface of the *SystemDefinition* class. The *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** .

```
def define_hw_platform(sys_def, config):
    """    Define HW platform: chassis, interconnections, PXI trigger resources,
synchronization, HVI clocks
    """    # Define chassis resources
    # For multi-chassis setup details see programming example documentation
    for chassis_number in config.chassis_list:
        if config.hardware_simulated:
                        # This simulation options require to install the chassis driver:
            # sys_def.chassis.add_with_options(chassis_number,
'Simulate=True,DriverSetup=Model=M9019A')
                        # As an alternative, the GenericPxieChassis allows to run simulations without
installing the chassis driver
            sys_def.chassis.add_with_options(chassis_number,
'Simulate=True,DriverSetup=Model=GenericPxieChassis')
        else:
            sys_def.chassis.add(chassis_number)

# Define System Sync Modules (SSMs)
    if config.system_sync_modules_descriptors:
        interconnects = sys_def.interconnects
        ssm_list = []
        for descriptor in config.system_sync_modules_descriptors:
            if config.hardware_simulated:
                ssm = interconnects.add_sync_module(descriptor.resource_id, config.ssm_
simulation_options)
            else:
                ssm = interconnects.add_sync_module(descriptor.resource_id,
descriptor.options)
            ssm_list.append(ssm)
```

```
# Define connections between SSMs
if config.ssm_connections:
    for connection in config.ssm_connections:
        connector_number = connection.ssm1_downstream_connector_number
        for ssm in ssm_list:
            if ssm.chassis == connection.ssm1_chassis:
                ssm1 = ssm
            if ssm.chassis == connection.ssm2_chassis:
                ssm2 = ssm
        # Implement each user-defined connection
        try:
            # Set connection. SSMs have always one upstream port
            ssm1.connectivity.systemsync_downstream[connector_number].set_connection
(ssm2.connectivity.systemsync_upstream[0])
        except:
            exit("Exception! Please check the valued defined for SyncModule resource
ids, chassis numbers and connections")

    # Assign the defined PXI trigger resources
    sys_def.sync_resources = config.pxi_sync_trigger_resources
    # 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
    sys_def.non_hvi_core_clocks = [10e6]
```

## 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.

**Python**

```
"""Define Names of HVI engines, actions, triggers, registers
"""# The primary engine is assigned to be the first one in the awg_module_descriptors
list
self.primary_engine_Name = ""self.awg_engine_Name = "AwgEngine"self.dio_engine_Name =
"DioEngine"self.awg_trigger_action_Name = "AwgTrigger"self.smb_trigger_Name =
"SmbTrigger"self.loop_register_Name = "Loops"self.num_loops = 3



def define_hvi_engines(sys_def, awg_module_dict, dio_module_dict):
    """    Define all the HVI engines to be included in the HVI
    """    # For each instrument to be used in the HVI application add its HVI Engine to
the HVI Engine Collection
    for engine_Name, module in awg_module_dict.items():
        sys_def.engines.add(module.hvi.engines.main_engine, engine_Name)
    for engine_Name, module in dio_module_dict.items():
        sys_def.engines.add(module.hvi.engines.primary, engine_Name)
```

## Define HVI Actions, Events, Triggers

In this programming example, each AWG needs to trigger both an FP pulse and a waveform very precisely. To do this, the 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 triggers need to be added to the HVI Trigger Collection and configured. This is done in this programming example as explained in the code snippets below.

**Python**

```
def define_hvi_actions(sys_def, module_dict, config):
    """
    Defines AWG trigger actions for each module, to be executed by the "action execute"
instruction in the HVI sequence
    Create a list of AWG trigger actions for each AWG module. The list depends on the
number of channels
    """    # For each AWG, define the list of HVI Actions to be executed and add such
list to its own HVI Action Collection
    for engine_Name in module_dict.keys():
        for ch_index in range(1, module_dict[engine_Name].num_channels + 1):
            # Actions need to be added to the engine's action list so that they can be
executed
            action_Name = config.awg_trigger_action_Name + str(ch_index) # arbitrary
user-defined Name
            instrument_action = "awg{}_trigger".format(ch_index) # Name decided by
instrument API
            action_id = getattr(module_dict[engine_Name].instrument.hvi.actions,
instrument_action)
            sys_def.engines[engine_Name].actions.add(action_id, action_Name)

def define_hvi_triggers(sys_def, module_dict, config):
    """    Defines and configure the eight FP SMB trigger outputs of each DIO instrument
    """    # Add to the HVI Trigger Collection of each DIO HVI Engine the SMB Trigger
object of each channel of that same instrument
    for engine_Name in module_dict.keys():
        for ch_index in range(1, module_dict[engine_Name].num_channels + 1):
            smb_trigger_ch_index = "smb{}".format(ch_index)
            smb_trigger_id = getattr(module_dict[engine_Name].hvi.triggers, smb_trigger_
ch_index)
            smb_trigger = sys_def.engines[engine_Name].triggers.add(smb_trigger_id,
config.smb_trigger_Name+str(ch_index))
            # Configure SMBx trigger in each hvi.engines[index]
            smb_trigger.config.direction = kthvi.Direction.OUTPUT
            smb_trigger.config.polarity = kthvi.Polarity.ACTIVE_HIGH
            smb_trigger.config.sync_mode = kthvi.SyncMode.IMMEDIATE
            smb_trigger.config.hw_routing_delay = 0
            # NOTE: the trigger mode is set to LEVEL. The length of the trigger pulse is
defined by the HVI Instructions TriggerWrite
            smb_trigger.config.trigger_mode = kthvi.TriggerMode.LEVEL
```

## 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.*

**Python**

```
# Create sequencer object
sequencer = kthvi.Sequencer("MySequencer", my_system)
```

```
def program_mimo_meas_sequence(sequencer, config):
    """
    Programs the MIMO Measurements HVI sync sequence
    """    # Define loop register
    loops = sequencer.sync_sequence.scopes[config.primary_engine_Name].registers.add
(config.loop_register_Name, kthvi.RegisterSize.SHORT)
    loops.initial_value = 0
    # Define Sync While condition
    condition = kthvi.Condition.register_comparison(loops,
kthvi.ComparisonOperator.LESS_THAN, config.num_loops)
    # Add a Sync While
    sync_while = sequencer.sync_sequence.add_sync_while("Sync MIMO Measurements", 50,
condition)
    # Set duration of each Sync While iteration
    sync_while.duration = kthvi.time.Duration(config.loop_duration,
kthvi.time.Unit.NANOSECONDS)
    # Add a Sync Multi-Sequence Block (SMSB)
    sync_block = sync_while.sync_sequence.add_sync_multi_sequence_block("Trigger Digital
I/Os, AWGs and Digitizers", 170)
    # Set SMSB duration
    sync_block.duration = kthvi.time.Minimum()
    # Program the SMSB to trigger AWGs
    program_mimo_trigger(sync_block, config)
```

## 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.

**Python**

```
# Define loop register
loops = sequencer.sync_sequence.scopes[config.primary_engine_Name].registers.add
(config.loop_register_Name, kthvi.RegisterSize.SHORT)
loops.initial_value = 0
```

## 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.

**Python**

```
# Define Sync While condition
condition = kthvi.Condition.register_comparison(loops, kthvi.ComparisonOperator.LESS_
THAN, config.num_loops)
# Add a Sync While
sync_while = sequencer.sync_sequence.add_sync_while("Sync MIMO Measurements", 50,
condition)
# Set duration of each Sync While iteration
sync_while.duration = kthvi.time.Duration(config.loop_duration,
kthvi.time.Unit.NANOSECONDS)
```

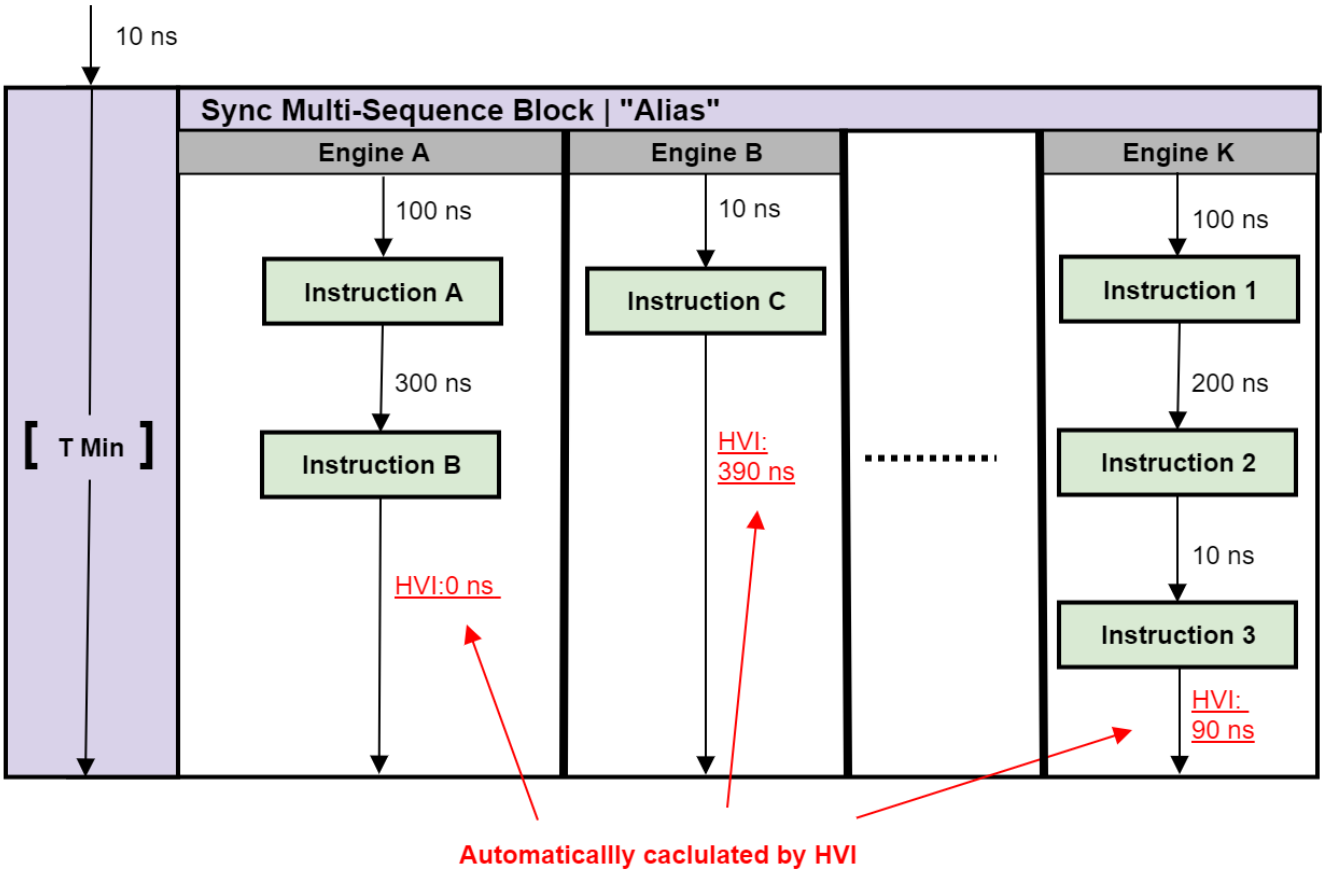
### Synchronized Multi-Sequence Block (b)

This block synchronizes all the HVI engines that are part of the sync sequence and allows to program each HVI Engine to do specific operations by exposing a local sequence for each engine. By calling the API method *add_multi_sequence_block()* 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.

**Python**

```
# Add a Sync Multi-Sequence Block (SMSB)
sync_block = sync_while.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs and
Digital I/Os", 250)
# Set SMSB duration
sync_block.duration = kthvi.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.




**NOTE:** **Keysight M3xxxA Instruments have an FPGA clock period equal to 10 ns**
**NOTE:** **Keysight M5xxxA Instruments have an FPGA clock period equal to 10/3 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 Triggers 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 *set_parameter()* . All HVI Native instructions and parameters are defined in the hvi.InstructionSet interface.

**Python**

```
# Create vector of values to be written to each SMB trigger
on_values = []
for trigger in range(0, smb_triggers.count):
    on_values.append(trigger_write.value.on)
# Write SMB Triggers ON
instr_trigger_ON = sequence.add_instruction("SMB Triggers ON", 10, trigger_write.id)
instr_trigger_ON.set_parameter(trigger_write.trigger.id, smb_triggers)
instr_trigger_ON.set_parameter(trigger_write.sync_mode.id, trigger_write.sync_
mode.immediate)
instr_trigger_ON.set_parameter(trigger_write.value.id, on_values)
```

### 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.

**Python**

```
# Execute AWG trigger from the HVI sequence of each module
# "Action Execute" instruction executes the AWG trigger from HVI
action_list = sequence.engine.actions
instruction1 = sequence.add_instruction("AWG trigger", 10, sequence.instruction_
set.action_execute.id)
instruction1.set_parameter(sequence.instruction_set.action_execute.action.id, action_
list)
```

### 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.

**Python**

```
# Retrieve register
loops = sequence.scope.registers[config.loop_register_Name]
# Increment register
instruction = sequence.add_instruction("Loops++", 10, sequence.instruction_set.add.id)
instruction.set_parameter(sequence.instruction_set.add.destination.id, loops)
instruction.set_parameter(sequence.instruction_set.add.left_operand.id, loops)
instruction.set_parameter(sequence.instruction_set.add.right_operand.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.

**Python**

```
# Delay statement
sequence.add_delay("Loop Delay", 50)
```

### Export the Programmed HVI Sequences to Text Format

KS2201A provides a feature to export the programmed HVI sequences to text format, which can be used both as a development and debug tool. The sequences can be exported using the to_string() method of the SyncSequence class, as illustrated in the code snippet below. Once exported to text format, the HVI sequences can be written to a text file or displayed on the console output. 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
output = sequencer.sync_sequence.to_string(kthvi.OutputFormat.DEBUG)
print("Programmed HVI sequences exported to file")
```

## 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.

**Python**

```
# Compile HVI sequences
hvi = sequencer.compile()
print(hvi.compile_status.to_string())
print("HVI Compiled")
print("This HVI programming example needs to reserve {} PXI trigger resources to
execute".format(len(hvi.compile_status.sync_resources)))
```

### 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.

**Python**

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

### 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.

**Python**

```
# 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(hvi.no_timeout)
print("HVI Running...")
```

### 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.

**Python**

```
# Unlock and release HW resources
hvi.release_hw()
print("Releasing HW...")
```

## 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 Python help file that is provided with the HVI installer, available at: C:\Program Files\Keysight\PathWave Test Sync Executive <year>\api\python\Help\index.htm, where <year> shall be replaced with the year of the release you are using, for example <year> = 2021.

# Conclusions

This programming example explained how to use PathWave Test Sync Executive and HVI (Hard Virtual Instrument) technology to synchronize multiple AWGs and Digital I/O (DIO) instruments to concurrently issue a marker pulse from all the DIO Front Panel (FP) SMB trigger ports and play a previously loaded waveform from all the AWG channels. The programming example use case illustrated here can be tested using a M5302A Digital I/O and any AWG of the Keysight M3xxxA PXI family. HVI technology was deployed using the HVI API (Application Programming Interface). Example measurement results demonstrated synchronized FP trigger marker output and waveform output from multiple AWGs and DIO instruments.