PROGRAMMING EXAMPLE 2

Synchronous Mixed-Signal Measurements using M3xxxA PXI Instruments

In this programming example a M3102A digitizer performs sequenced acquisition of mixed signals generated by multiple M320xA Arbitrary Waveform Generators (AWGs). The first AWG generates a train of RF pulses, and the other AWGs output previously queued arbitrary waveforms. By using PathWave Test Sync Executive, each cycle of digitizer measurements is precisely synchronized with the AWG output signals.

```
# 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_wri
    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])
```

Table of Contents

KS2201A - Programming Example 2 - Synchronous Mixed-Signal Measurements using M3xxxA PXI Instruments

In this programming example, an M3102A digitizer performs sequenced acquisition of heterogeneous signals generated by multiple M320xA arbitrary waveform generators (AWGs). The first AWG generates a train of RF pulses, and the other AWGs output a queued arbitrary waveform. By using PathWave Test Sync Executive, each cycle of digitizer measurements are precisely synchronized with the AWG output signals.

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)

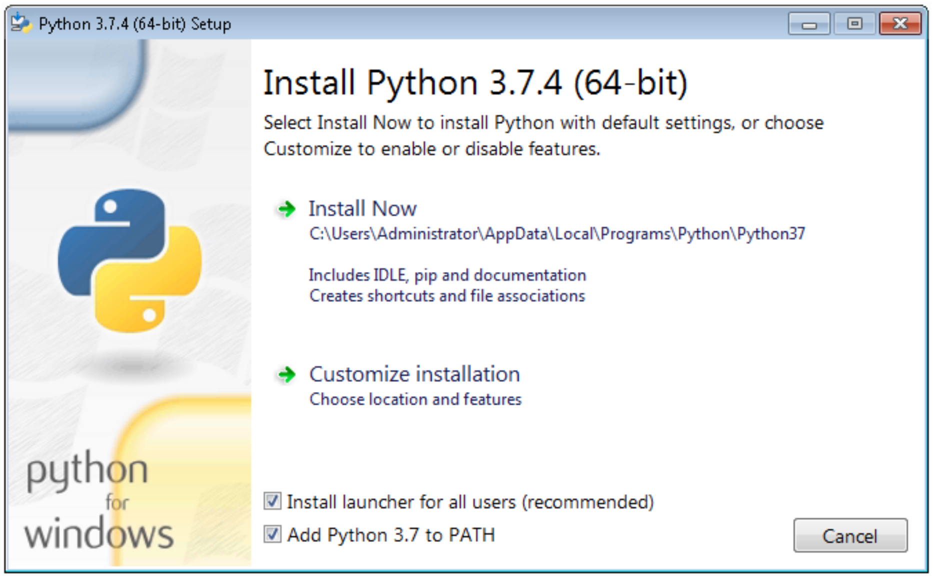
3. Keysight PXIe Instruments with FPGA firmware versions (to be installed using Keysight instrument SFP):
   - M3202A AWG FPGA firmware (v4.2.45 or later)
   - M3201A AWG FPGA firmware (v4.3.67 or later)
   - M3102A Digitizer FPGA firmware (v2.2.46 or later)
   - M9032A System Synchronization Module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



Install Python 3.7.4 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\Administrator\AppData\Local\Programs\Python\Python37

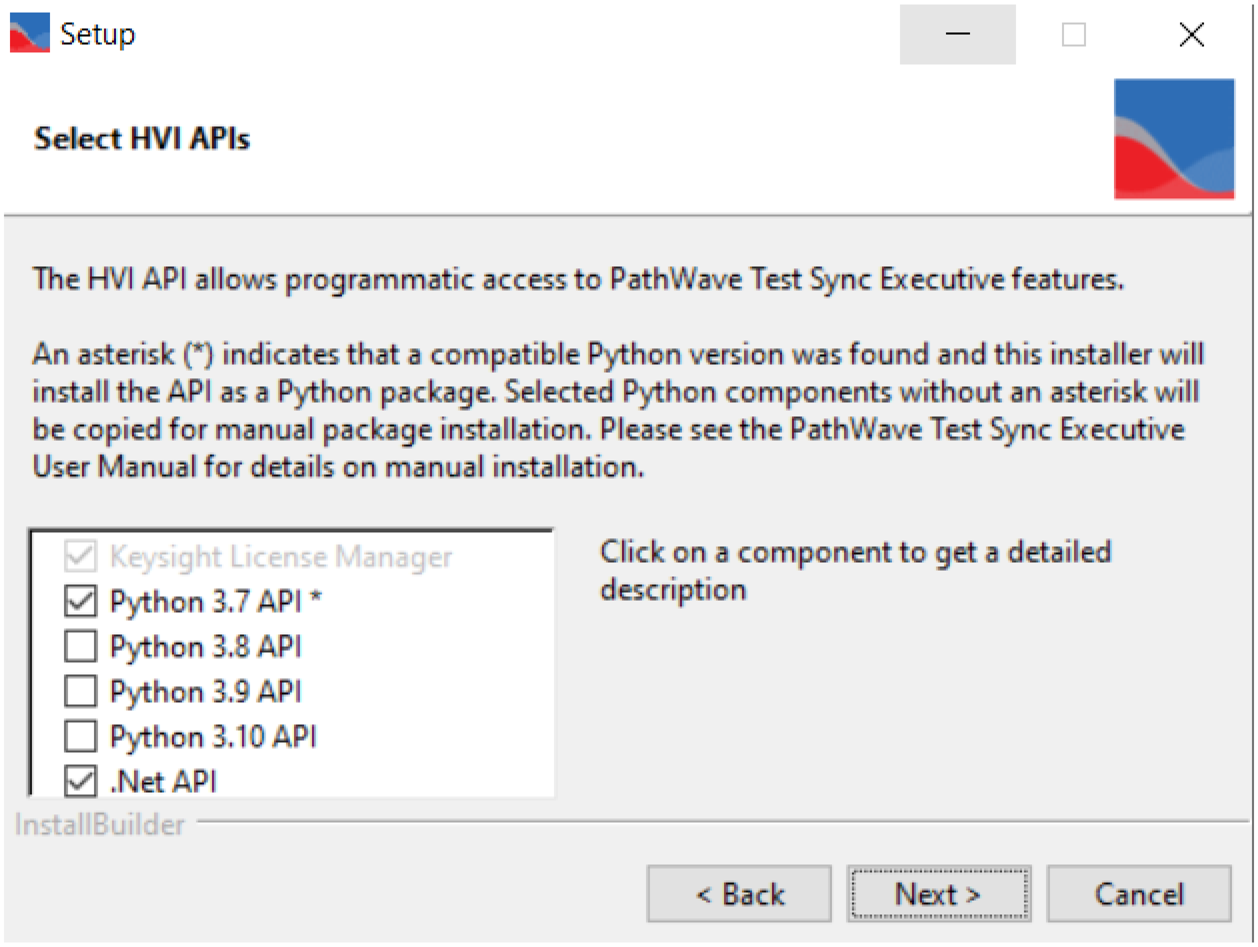
Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

[✓] Install launcher for all users (recommended)
[✓] Add Python 3.7 to PATH

Cancel

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.



Setup

Select HVI APIs

The HVI API allows programmatic access to PathWave Test Sync Executive features.

An asterisk (*) indicates that a compatible Python version was found and this installer will install the API as a Python package. Selected Python components without an asterisk will be copied for manual package installation. Please see the PathWave Test Sync Executive User Manual for details on manual installation.

- [✓] Keysight License Manager
- [✓] Python 3.7 API *
- [ ] Python 3.8 API
- [ ] Python 3.9 API
- [ ] Python 3.10 API
- [✓] .Net API

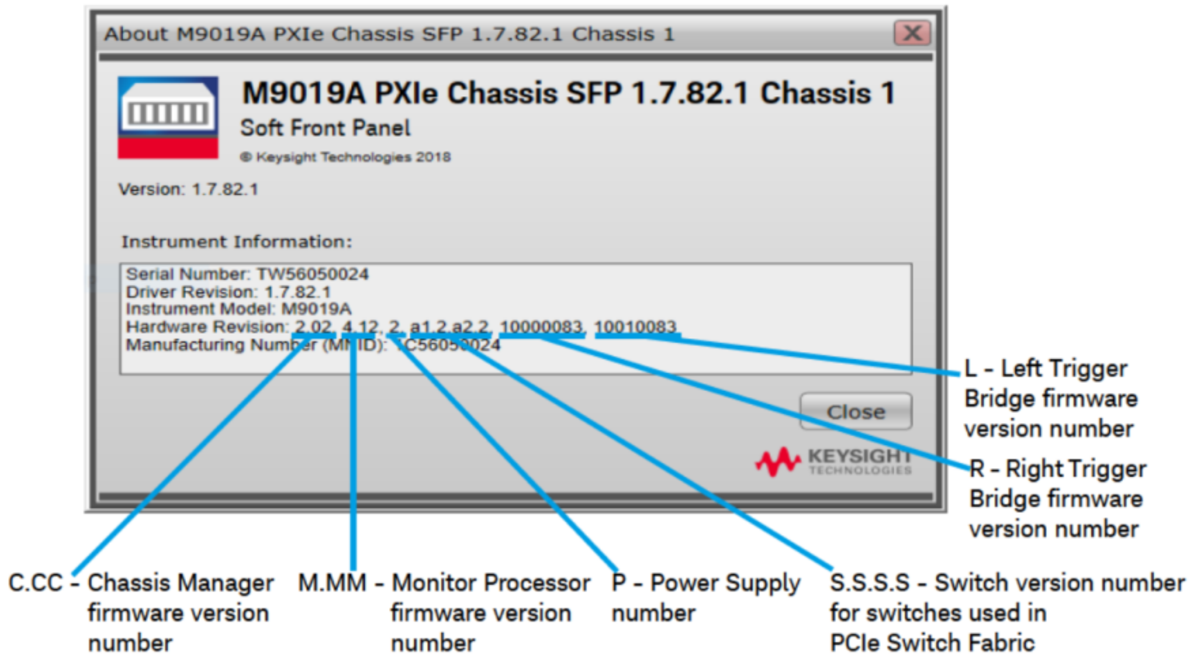
Click on a component to get a detailed description

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 .


About M9019A PXIe Chassis SFP 1.7.82.1 Chassis 1

M9019A PXIe Chassis SFP 1.7.82.1 Chassis 1

Soft Front Panel

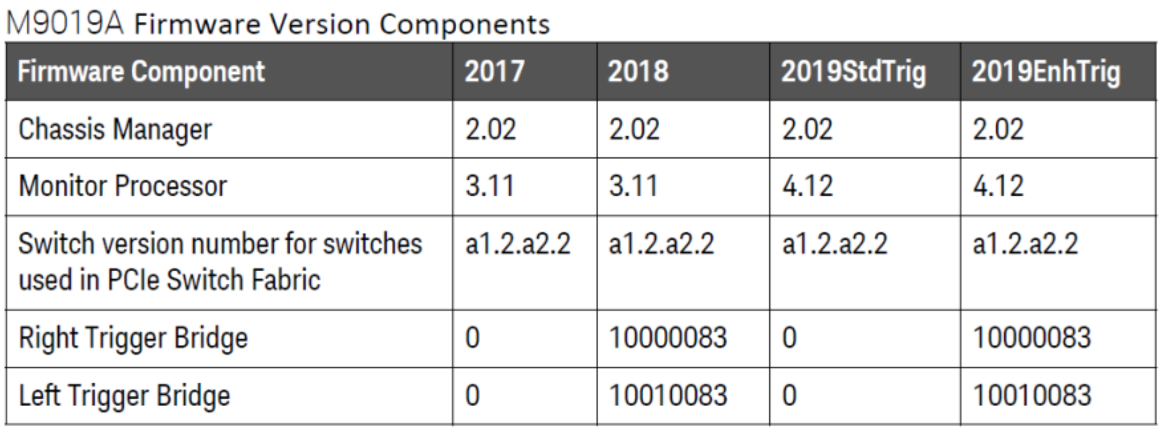
© Keysight Technologies 2018

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 (MN ID): 1C56050024
```

Close
KEYSIGHT TECHNOLOGIES
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

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

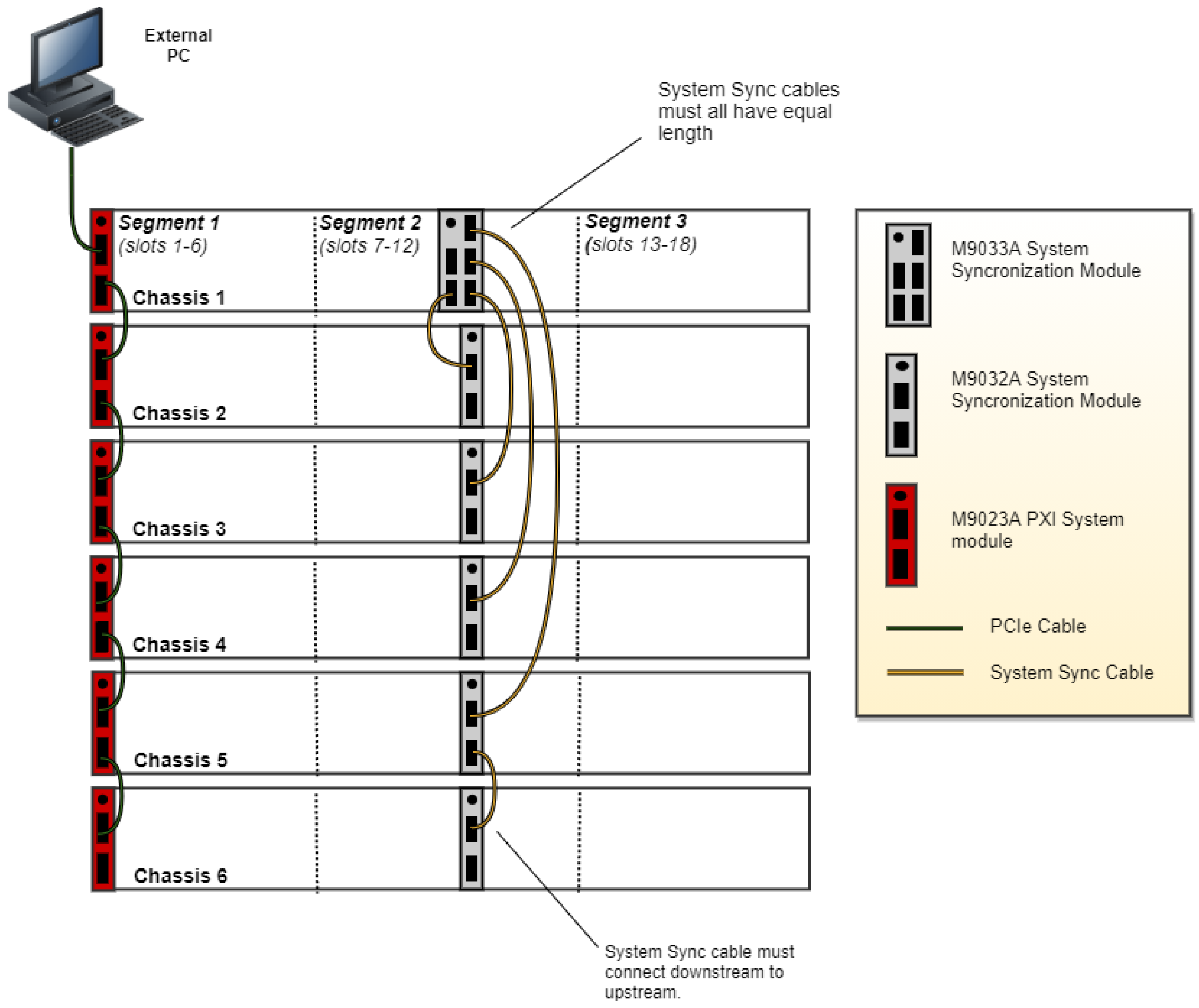
In a multi-chassis system connected with Keysight PXIe System Synchronization Modules (SSM), you must plug one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different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

In this example, AWG signal generation is controlled at FPGA level using local flow control loops. This way, a train of RF pulses can be generated and previously loaded arbitrary waveforms can be queued and played. An HVI Synchronized While Statement controls the digitizer acquisitions and enables synchronization of each acquisition cycle to capture the AWG outputs generated within each loop of the HVI Synchronized While Statement.

More specifically, this programming example illustrates the following HVI real-time functionalities:

1. Synchronized While Global Statement.
2. Wait-for-event Statement.
3. Use of registers and scopes.
4. Local Flow Control Statements: WHILE loop, IF.
5. HVI Product-specific Instructions.

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. Update the model numbers of the PXI instruments used, if they are different than the instrument models 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. Please update the properties in each *module-descriptor* object before running the programming example:

```
# Define module descriptors below with your instruments information1
self.digitizer_descriptor = ModuleDescriptor('M3102A', 1, 9, self.options, self.dig_
engine)
self.rf_gen_descriptor = ModuleDescriptor('M3202A', 1, 8, self.options, self.rf_gen_
engine) # instrument to be used as an RF Pulse Gen.
self.awg_descriptors = [ModuleDescriptor('M3202A', 1, 7, self.options, "")]
```

```
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]
```

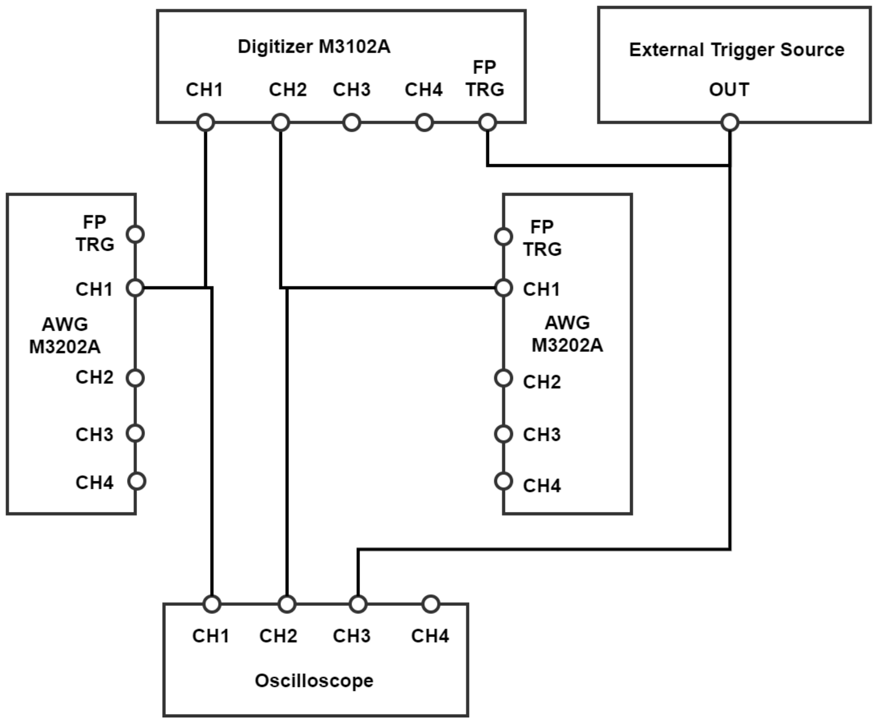
PXI lines allocated to be used as HVI trigger resources cannot be used by the programming example for other purposes. The vector pxi_sync_trigger_resources specified above must include at least the necessary number of PXI lines for the programming example to execute. Please check the programming example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

Application-specific parameters that are necessary to configure the digitizer and the AWGs are listed in dedicated classes. Before running the programming example, update, if necessary, the AWG and digitizer parameters contained in the code classes listed below. Measurement results reported in this document were obtained using the parameter values reported in the following code snippets.

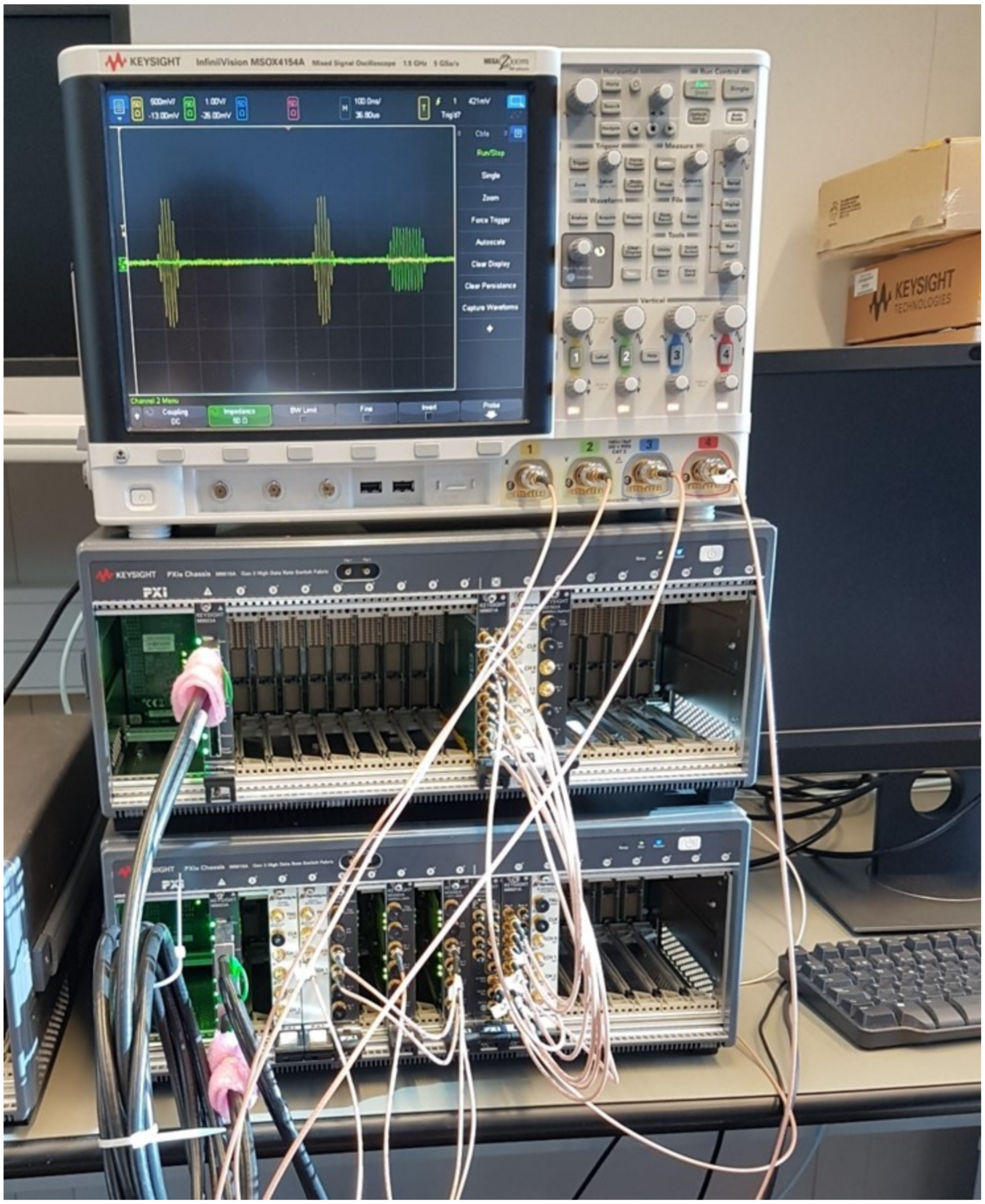
```
"""AWG Parameters
"""# AWG settings for all channels
self.sync_mode = keysightSD1.SD_SyncModes.SYNC_NONE
self.queue_mode = keysightSD1.SD_QueueMode.ONE_SHOT
self.awg_mode = keysightSD1.SD_Waveshapes.AOU_AWG
self.start_delay = 0 # x10 [ns]
self.prescaler = 0
self.wfm_A_cycles = 3
self.wfm_B_cycles = 2
self.amplitude = 1
self.wfm_A = 0
self.wfm_B = 1
# Trigger settings
self.trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG
# Latency values for M3202A AWGqueueWfm() [ns]
# Latencies depend on AWG FPGA FW. Check M3xxx User Guide for further info
self.queue_wfm_latency = 100 # [ns] Minimum start delay necessary to execute an
AWGqueueWfm() instruction
self.awg_trigger_latency = 2000 # [ns] Minimum latency necessary between an AWGqueueWfm
() instruction and an AWGtrigger action.
self.wfm_length = 100 # [ns]
"""
RF pulse generator parameters
"""self.all_ch_mask = 0xF # binary mask defining which channels to use
self.offset = 0 # [V]
self.frequency = 10e6 # [Hz]
self.num_loops = 3 # sync while loops
self.num_pulses = 5
self.ON_value = 1.0 # [V]
self.OFF_value = 0.0 # [V]
self.n_AWG = 1 # channel number to be used as RF Gen
self.pulse_ontime = 200 # [ns]
self.pulse_delay = 100 # [ns]
"""
Digitizer parameters
"""self.all_ch_mask = 0xF
self.dig_sampling_time = 2 # [ns] 1/sample_rate, sample_rate = 500 MSa/s for Digitizer
M3102A
self.acquisition_points_per_cycle = 1500
self.prescaler = 0
self.fullscale = 2 # [V] enter x Volts to set the full scale to [-x, x] Volts
self.acquisition_time_per_cycle = self.acquisition_points_per_cycle*self.dig_sampling_
time
self.num_cycles = self.num_loops #insert -1 for infinite cycles
self.acquisition_points = int(self.acquisition_points_per_cycle*self.num_loops)
self.acquisition_delay = 150 # x2 [ns]
self.trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG
self.mask = self.all_ch_mask
```

Measurement Results

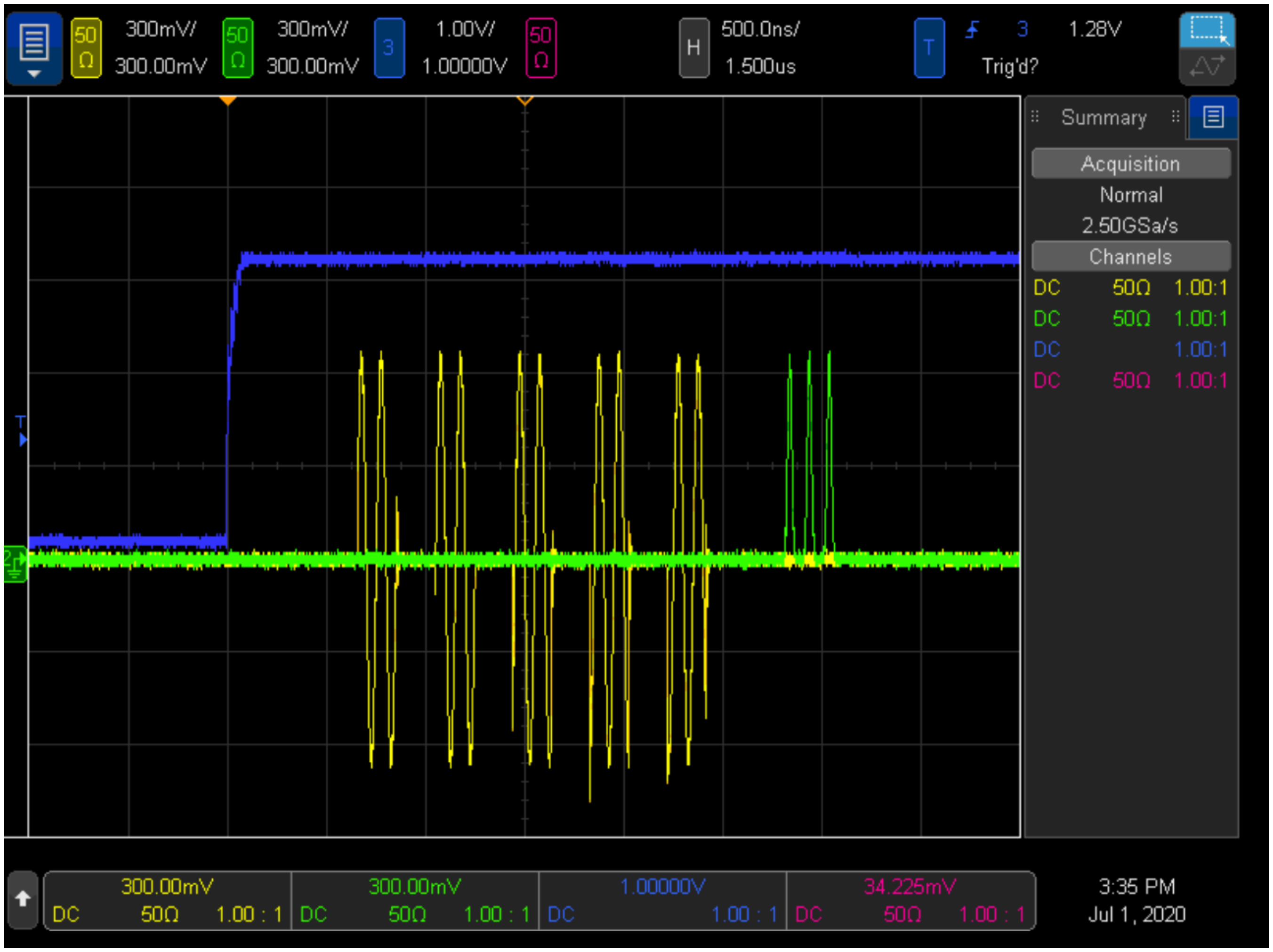
This section describes the measurement results obtained by deploying this programming example on a setup including two M9019A PXI chassis, an M3102A digitizer and two M3202A AWGs. A block diagram of the measurement setup used in this document is reported below.



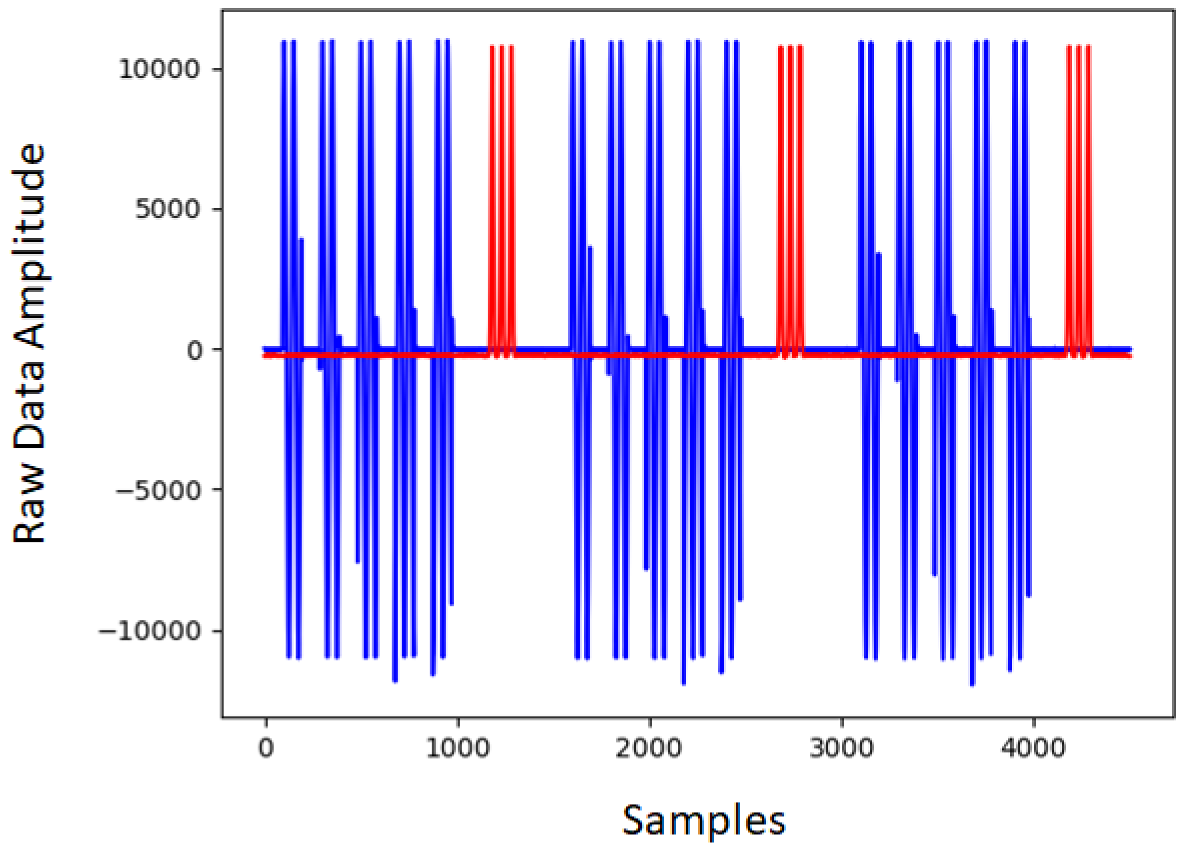
A photograph of the measurement setup used for the measurement results reported in this section is also reported below:



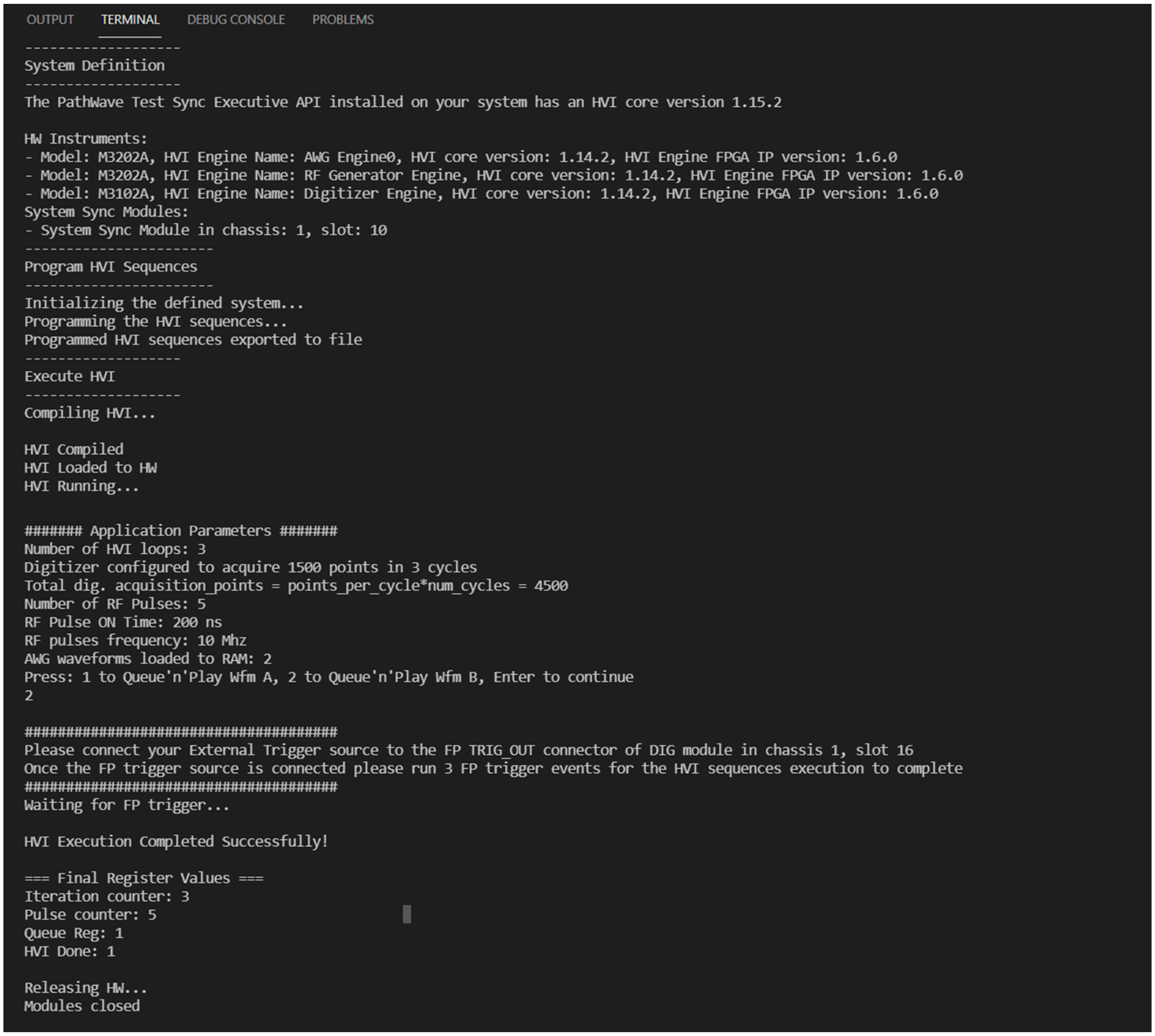
The oscilloscope measurements below show measurement results obtained using a digitizer M3102A and two AWG M3202As. All instruments have the -HV1 option enabled that allows to use them to execute HVI applications. In the scope measurement, we can observe the external trigger signal sent to the digitizer Front Panel (FP) TRG Port (**blue** waveform). The FP trigger provides the condition necessary for the wait statement to continue the HVI sequence execution and generate a series of RF pulses from the first AWG (**yellow** waveform) and queue'N'play an arbitrary waveform from the second AWG (**green** waveform in the scope measurement screenshot).



The plot below depicts digitizer data acquired over three multiple cycles. Each acquisition cycle corresponds to an iteration of the HVI Sync While statement described in the next section. The **blue** trace represents data acquired by DAQ1 channel connected to the AWG used as an RF pulse generator. The **red** trace represents DAQ2 channel measurements obtained from the AWG output that generates arbitrary waveforms selected by the user.

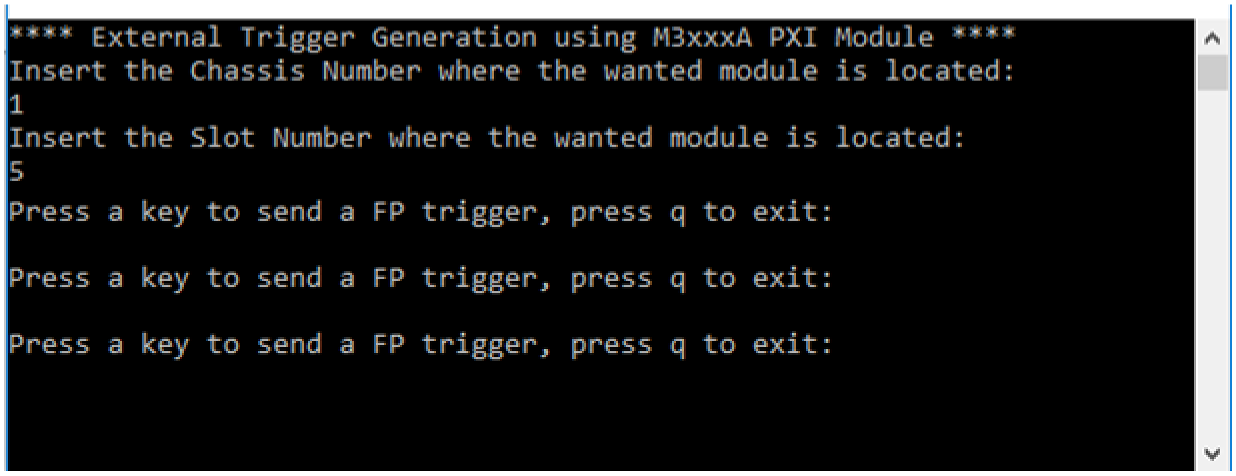


The screenshot below depicts the expected execution of this programming example's Python code.



```
-------------------
System Definition
-------------------
The PathWave Test Sync Executive API installed on your system has an HVI core version 1.15.2

HW Instruments:
- Model: M3202A, HVI Engine Name: AWG Engine0, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
- Model: M3202A, HVI Engine Name: RF Generator Engine, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
- Model: M3102A, HVI Engine Name: Digitizer Engine, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
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
HVI Loaded to HW
HVI Running...

####### Application Parameters #######
Number of HVI loops: 3
Digitizer configured to acquire 1500 points in 3 cycles
Total dig. acquisition_points = points_per_cycle*num_cycles = 4500
Number of RF Pulses: 5
RF Pulse ON Time: 200 ns
RF pulses frequency: 10 Mhz
AWG waveforms loaded to RAM: 2
Press: 1 to Queue'n'Play Wfm A, 2 to Queue'n'Play Wfm B, Enter to continue
2

######################################
Please connect your External Trigger source to the FP TRIG_OUT connector of DIG module in chassis 1, slot 16
Once the FP trigger source is connected please run 3 FP trigger events for the HVI sequences execution to complete
######################################
Waiting for FP trigger...

HVI Execution Completed Successfully!

=== Final Register Values ===
Iteration counter: 3
Pulse counter: 5
Queue Reg: 1
HVI Done: 1

Releasing HW...
Modules closed
```

With this programming example, it is provided an executable GenExtTrigger_M3xxx.exe that can be used to generate the FP triggers from any M3xxx AWG module that is external to the HVI application. It is used here to emulate the external trigger. An example execution on the console terminal of this independent executable for FP trigger generation is displayed in the screenshot below.

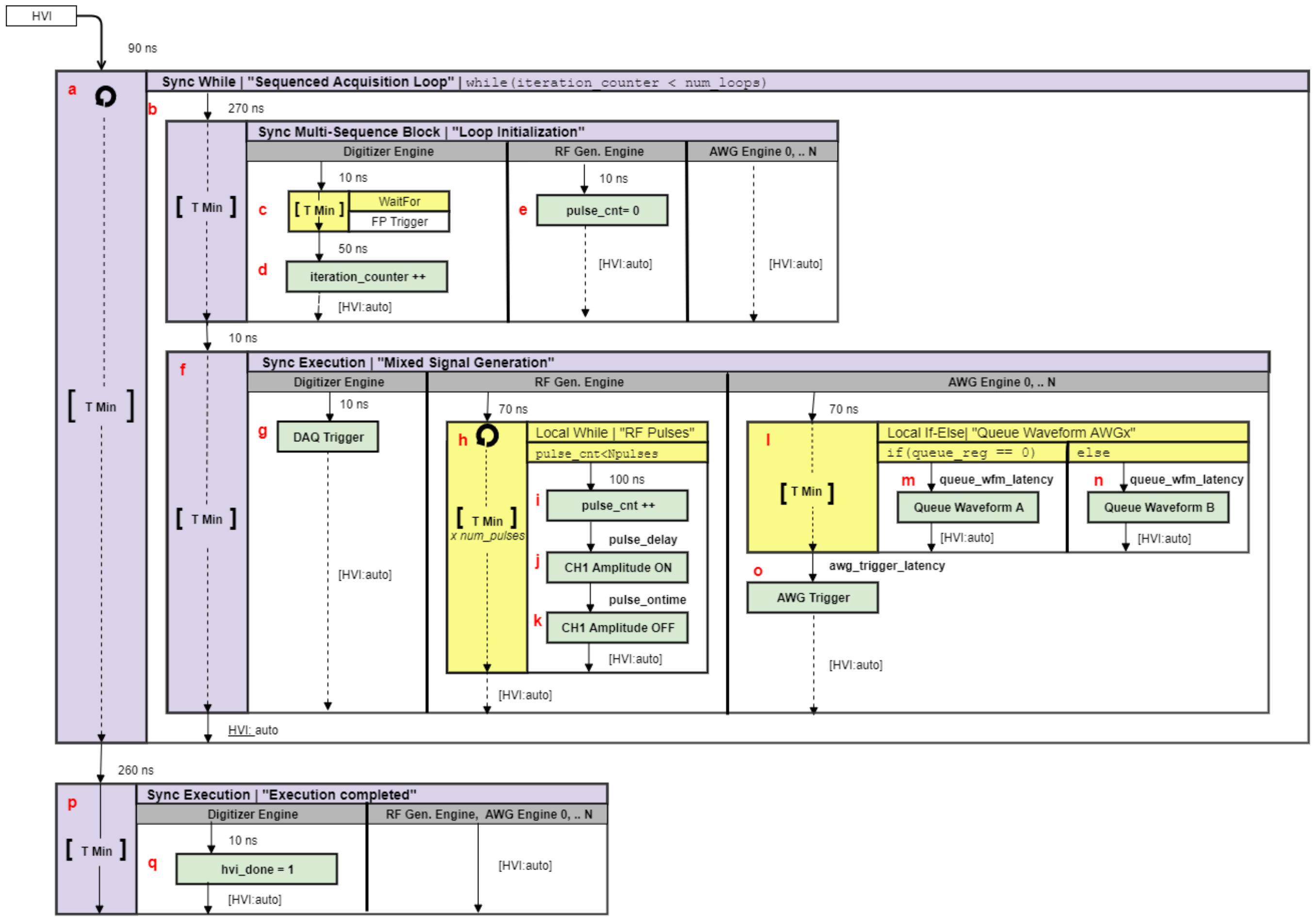


```
**** External Trigger Generation using M3xxxA PXI Module ****
Insert the Chassis Number where the wanted module is located:
1
Insert the Slot Number where the wanted module is located:
5
Press a key to send a FP trigger, press q to exit:

Press a key to send a FP trigger, press q to exit:

Press a key to send a FP trigger, press q to exit:
```

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: 10 ns is the FPGA clock period for M3xxxA instruments

NOTE The duration of each iteration of the Sync While loop used in this example is unknown due to the unknown execution time of the local flow control statements (If, While, Wait) used inside the loop. The unknown duration is represented by the dotted arrows in the HVI diagram. Due to its unknown duration, it is not possible to use the Sync While duration property to specify how long each loop iteration should last.

NOTE Fixed delays can be parametrized in HVI sequences by using Python Variables. For example, the Python Variables *pulse_delay* and *pulse_ontime* are used to parametrize the RF pulse generation. Users can update them before execution using the *ApplicationConfig* class. To implement a Variable delay in an HVI sequence, the WaitTime statement shall be used instead. More information can be found in the KS2201A User Manual.

NOTE AWG queue waveform and AWG trigger operations require a minimum latency to correctly execute which is specified using Python Variables *queue_wfm_latency* and *awg_trigger_latency*. These Variables can be also updated using the *ApplicationConfigclass*. Using lower values than what specified in this example may cause misbehaviors during HVI execution. This may happen for example because the AWG FPGA FW is not being allowed enough time to real-time queue the waveforms before the command to reproduce them (AWG trigger) is issued. AWG latency information is documented in the M3xxx AWG documentation and in the SD1 3.x documentation.

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(sys_def, module_dict, config):
    """    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, module_dict)
    # Define list of actions to be executed
    define_hvi_actions(sys_def, module_dict, config)
    # Defines the trigger resources
    define_hvi_triggers(sys_def, 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 it to access the actions, events and triggers that each specific engine will control and use within the HVI. In this programming example in particular two HVI engines are used, one for the AWG, the other for the digitizer.

Python

```
# HVI engines
self.awg_engine = "AWG Engine"self.rf_gen_engine = "RF Generator Engine"self.dig_engine
= "Digitizer Engine"

def define_hvi_engines(sys_def, 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 in module_dict.keys():
        sys_def.engines.add(module_dict[engine_Name].instrument.hvi.engines.main_engine,
engine_Name)
```

Define HVI Actions, Events, Triggers

In this programming example, both the AWG and the digitizer need to trigger waveforms or acquisition very precisely. To do that the AWG trigger and DAQ 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. This is done in this programming example as explained in the code snippets below.

Python

```
# HVI actions
self.awg_trigger = "AWG_Trigger"self.daq_trigger = "DAQ_Trigger"# HVI triggers
self.fp_trigger = "FP Trigger"

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
            if engine_Name == config.dig_engine:
                action_Name = config.daq_trigger + str(ch_index) # arbitrary user-
defined Name
                instrument_action = "daq{}_trigger".format(ch_index) # Name decided by
instrument API
            else:
                action_Name = config.awg_trigger + 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 the FP trigger to be used as a wait condition by the digitizer "    # Add
to the HVI Trigger Collection of each HVI Engine the FP Trigger object of that same
instrument
    fp_trigger_id = module_dict[config.dig_engine].instrument.hvi.triggers.front_panel_1
    fp_trigger = sys_def.engines[config.dig_engine].triggers.add(fp_trigger_id,
config.fp_trigger)
    # Trigger configuration
    # NOTE: Trigger to be used as WaitEvent conditions must be configured as
kthvi.Direction.INPUT
    # DriveMode (e.g. PushPull/OpenDrain) is defined by the instrument and cannot be
changed by the user
    fp_trigger.config.direction = kthvi.Direction.INPUT
    fp_trigger.config.polarity = kthvi.Polarity.ACTIVE_HIGH
    fp_trigger.config.hw_routing_delay = 0
    fp_trigger.config.trigger_mode = kthvi.TriggerMode.LEVEL
```

Program HVI Sequences

Once the HVI resources are defined, you program the HVI sequence of measurement actions to be executed by each HVI engine. HVI sequences can be programmed using the *Sequencer* class. HVI execution happens 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 consists of a synchronized while statement containing two synchronized multi-sequence blocks.

Python

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

```
def program_mixed_sig_meas_sequence(sequencer, module_dict, config):
    """    This method programs the HVI sequence of this application.
    Different HVI statements are encapsulated as much as possible in separated SW
methods to help users visualize
    the programmed HVI sequences.
    The programming example documentation on www.keysight.com contains an HVI diagram
that graphically represents the programmed HVI sequence.
    """    # Define registers within the scope of the outmost sync sequence
    define_registers(sequencer, module_dict, config)
    # Define sync while condition
    iteration_counter = sequencer.sync_sequence.scopes[config.dig_engine].registers
[config.iteration_counter]
    sync_while_condition = kthvi.Condition.register_comparison(iteration_counter,
kthvi.ComparisonOperator.LESS_THAN, config.num_loops)
    # Add Sync While Statement
    sync_while = sequencer.sync_sequence.add_sync_while("Sequenced Acquisition Loop",
90, sync_while_condition)
    # Program Sequenced Acquisition Loops
    program_sequenced_meas_loop(sync_while.sync_sequence, module_dict, config)
    # Add 3rd Sync Multi-Sequence Block
    sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("Execution
Completed", 230)
    # Program execution completed
    program_execution_completed(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

```
# HVI registers
self.iteration_counter = "Iteration Counter"self.pulse_counter = "Pulse
Counter"self.queue_reg = "Queue Reg"self.reg_wfm_A = "Wfm A"self.reg_wfm_B = "Wfm
B"self.hvi_done = "HVI Done"



def define_registers(sequencer, module_dict, config):
    " Defines all registers for each HVI engine in the scope af the global sync sequence
"    # Digitizer registers
    iteration_counter = sequencer.sync_sequence.scopes[config.dig_engine].registers.add
(config.iteration_counter, kthvi.RegisterSize.SHORT)
    iteration_counter.initial_value = 0
    hvi_done = sequencer.sync_sequence.scopes[config.dig_engine].registers.add
(config.hvi_done, kthvi.RegisterSize.SHORT)
    hvi_done.initial_value = 0
    # RF Gen registers
    pulse_counter = sequencer.sync_sequence.scopes[config.rf_gen_engine].registers.add
(config.pulse_counter, kthvi.RegisterSize.SHORT)
    pulse_counter.initial_value = 0
    # AWG 1:N Registers
    for engine_Name in module_dict.keys():
        if engine_Name!=config.rf_gen_engine and engine_Name!=config.dig_engine:
            queue_reg = sequencer.sync_sequence.scopes[engine_Name].registers.add
(config.queue_reg, kthvi.RegisterSize.SHORT)
            queue_reg.initial_value = 0
            reg_wfm_A = sequencer.sync_sequence.scopes[engine_Name].registers.add
(config.reg_wfm_A, kthvi.RegisterSize.SHORT)
            reg_wfm_A.initial_value = config.wfm_A
            reg_wfm_B = sequencer.sync_sequence.scopes[engine_Name].registers.add
(config.reg_wfm_B, kthvi.RegisterSize.SHORT)
            reg_wfm_B.initial_value = config.wfm_B
```

Synchronized While

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. For local sequences to be defined within the Sync While, it is necessary to use synchronized multi-sequence blocks.

Python

```
# Define sync while condition
sync_while_condition = kthvi.Condition.register_comparison(iteration_counter,
kthvi.ComparisonOperator.LESS_THAN, rf_pulse_params.num_loops)
# Add Sync While Statement
sync_while = sequencer.sync_sequence.add_sync_while('Sequenced Acquisition Loop', 60,
sync_while_condition)
```

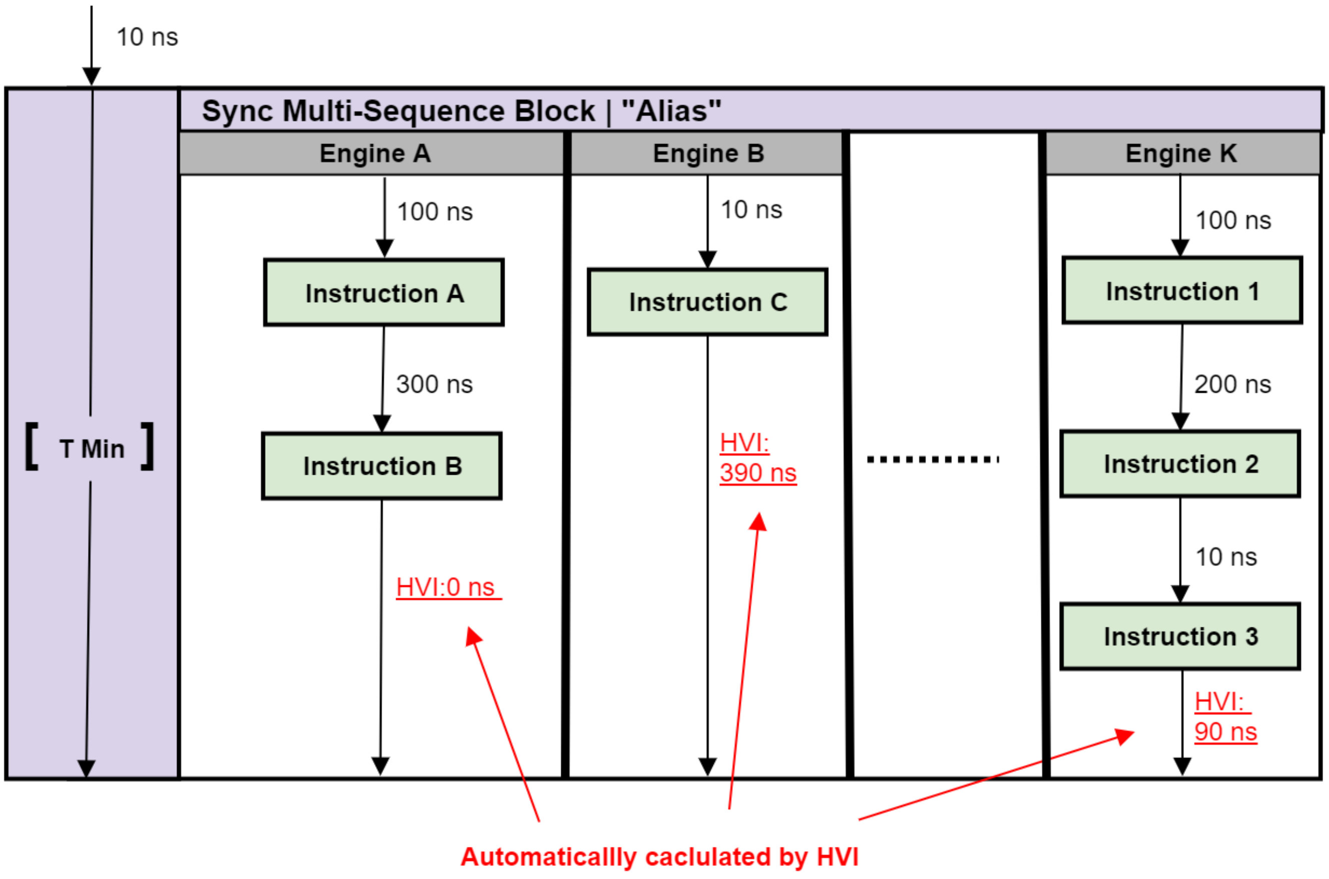
Synchronized Multi-Sequence Block

This corresponds to statements (b, f, p) in the HVI diagram. Synchronized multi-sequence blocks are defined by the API class *SyncMultiSequenceBlock.* This type of sync statement synchronizes all the HVI engines that are part of the sync sequence. It allows you 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.

Python

```
# Add 1st Sync Multi-Sequence Block to the Sync While sequence
sync_block_1 = sync_sequence.add_sync_multi_sequence_block("Loop Initialization", 270)
```

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

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

Wait Statement

This corresponds to statement (c) in the HVI diagram. The wait statement is a local flow control statement that can be implemented using the API class *WaitStatement.* This sequence block sets an instrument to wait for a condition. The condition can be defined by a trigger, an event, or any combination of them through the usage of logical operators. In this programming example, the wait is used to set the digitizer to wait for an external front panel trigger. The wait statement is set to wait for a trigger falling edge using the .wait mode *WaitMode.TRANSITION* combined with a trigger configuration as *ACTIVE_LOW* . The sync mode SyncMode.IMMEDIATE sets the wait event to let the execution continue immediately, i.e. as soon as the trigger event is received.

Python

```
# Define the condition for the wait statement
trigger_event = dig_sequence.engine.triggers[config.fp_trigger]
wait_condition = kthvi.Condition.trigger(trigger_event)
# Add a Wait For Event
wait_event = dig_sequence.add_wait("Wait for FP Trigger", 10, wait_condition)
wait_event.set_mode(kthvi.WaitMode.TRANSITION, kthvi.SyncMode.IMMEDIATE)
```

HVI Native Instruction: Register Increment

This corresponds to statement (d, i) in the HVI diagram. 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 needs to be added previously to the scope of the corresponding HVI engine.

Python

```
# Increment the sync while iteration counter for each external trigger event that is
received
instruction = sequence.add_instruction('Increment Counter', 50, sequence.instruction_
set.add.id)
instruction.set_parameter(sequence.instruction_set.add.destination.id, iteration_
counter)
instruction.set_parameter(sequence.instruction_set.add.left_operand.id, iteration_
counter)
instruction.set_parameter(sequence.instruction_set.add.right_operand.id, 1)
```

HVI Native Instruction: Register Assign

This corresponds to statements (e, q) in the HVI diagram. A register assign statement can be used to initialize a register to an initial value using the instruction class InstructionsAssign from Python HVI API. The same instruction can be used to assign a register value (source) to another register (destination). Each register can also be initialized outside an HVI sequence using the API method KtviRegister.set_initial_value.

Python

```
# In sync_block_1 Initialize pulse_counter = 0 in RF Gen Engine
instruction = sequence.add_instruction('Initialize Pulse Counter', 10,
sequence.instruction_set.assign.id)
instruction.set_parameter(sequence.instruction_set.assign.destination.id, pulse_counter)
instruction.set_parameter(sequence.instruction_set.assign.source.id, 0)
```

Action Execute: DAQ, AWG Trigger

This corresponds to statement (g, o) in the HVI diagram. 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

```
# Action-execute
daq_trigger_all = dig_sequence.engine.actions
inst_daq_trigger = dig_sequence.add_instruction("DAQ Trigger", 10, dig_
sequence.instruction_set.action_execute.id)
inst_daq_trigger.set_parameter(dig_sequence.instruction_set.action_execute.action.id,
daq_trigger_all)
```

Local While

This corresponds to statement (h) in the HVI diagram *WhileStatement* class allows you to add a local WHILE loop sub-sequence within the main HVI sequence of any instrument engine. The WHILE sub-sequence runs until the WHILE condition is met. The condition can be defined using the API class *ConditionalExpression* . Once the WHILE loop sub-sequence is created, it can be programmed using the same API methods and classes used to program the main HVI sequence.

Python

```
# Local WHILE: generate RF pulses
local_while_condition = kthvi.Condition.register_comparison(pulse_counter,
kthvi.ComparisonOperator.LESS_THAN, config.num_pulses)
while_loop = rf_gen_sequence.add_while("RF pulses", 70, local_while_condition)
while_sequence = while_loop.sequence
#
#Increment pulse_counter
instruction = while_sequence.add_instruction("Increment Pulse Counter", 20, while_
sequence.instruction_set.add.id)
instruction.set_parameter(...)
```

HVI Instrument-Specific Instruction

Instrument-specific instructions are in statements (j, k, m, n) of the HVI diagram. This block executes a product-specific HVI instruction. Native HVI instructions are common to every Keysight product. 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 product-specific instructions and parameters are defined in the hvi.InstructionSet interface of each product. Instructions, actions, events and in general all the HVI definitions specific of M3xxx instruments can be found in the M3xxx User Guide available on www.keysight.com.

Python

```
# Set CH1 amplitude to ON_value
instruction = while_sequence.add_instruction('Set CH1 amplitude to ON_value', 100,
module_dict[hvi_eng_Names.rf_gen_engine].instrument.hvi.instruction_set.set_
amplitude.id)
instruction.set_parameter(module_dict[hvi_eng_Names.rf_gen_
engine].instrument.hvi.instruction_set.set_amplitude.channel.id, rf_pulse_params.n_AWG)
instruction.set_parameter(module_dict[hvi_eng_Names.rf_gen_
engine].instrument.hvi.instruction_set.set_amplitude.value.id, rf_pulse_params.ON_value)
```

IF-ELSEIF-ELSE Statement

This corresponds to statement (I) in the HVI diagram. *IfStatement* class allows you to add an IF-ELSEIF-ELSE loop within the main HVI sequence of any instrument engine. The IF-ELSEIF-ELSE loop contains one (or more) IF branches and an ELSE branch. The instructions and/or statements contained in each IF or ELSE branch are executed if the condition of each branch is met. The condition of each branch can be defined using the API class *ConditionalExpression* . Branch sub-sequence can be programmed using the same API methods and classes used to program the main HVI sequence, by means of the API classes *IfBranch* and *ElseBranch* .

Python

```
# Configure IF condition
if_condition = kthvi.KtHviCondition.register_comparison(queue_reg[index],
kthvi.ComparisonOperator.EQUAL_TO, 0)
# Set flag that enables to match the execution time of all the IF branches
enable_ifbranches_time_matching = True
# Add If statement
if_statement = awg_sequence.add_if('Queue Wfm AWG' + str(index), 10, if_condition,
enable_ifbranches_time_matching)
# Program IF branch
if_sequence = if_statement.if_branch.sequence
# Add statements in if-sequence
instruction = if_sequence.add_instruction(instrLabel, start_delay, module
[index].hvi.instructions.queue_waveform.id)
instruction.set_parameter(...)
...
# Eventually add Else-If-branches (not used in this programming example)
else_if_condition_1 = ...
else_if_branch_1 = ...
...
# Else-branch
# Program Else branch
else_sequence = else_branch.sequence
# Add statements in Else-sequence
instruction = else_sequence.add_instruction(...)
...
```

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 non-blocking mode is used. By using this execution mode, SW execution can interact through registers read/write with the HVI sequence execution.

Python

```
# Execute HVI in non-blocking mode
# This mode allows SW execution to interact with HVI execution
hvi.run(hvi.no_wait)
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 synchronously execute sequences of measurement actions over multiple M3xxxA PXI instruments. Validation measurements showed how to synchronize an M3102A digitzer and an arbitrary number of M320xA AWGs to iteratively acquire heterogeneous signals generated over multiple cycles.

This information is subject to change without notice. © Keysight Technologies, 2020-2021, Published in USA, December 9 2021,KS2201-90002