

Multi-Channel Sync Playback using M32xxA Arbitrary Waveform Generators

PATHWAVE

In this programming example, PathWave Test Sync Executive is used to program multiple M32xxA AWGs to synchronously output first a FP (Front Panel) trigger pulse and afterwards a previously queued waveform. All modules run fully synchronized and actions across modules can be controlled with the timing resolution of the M32xxA AWGs which is of 10ns.

PATHWAVE

Test Sync Executive



Table of Contents

KS2201A - Programming Example 1 - Multi-Channel Sync Playback using M320xA Arbitrary Waveform Generators	3
System Setup	3
System Requirements	3
How to install Python 3.7.x 64-bit	4
How to Install Chassis Driver, SFP and Firmware	4
How to Install PathWave Test Sync Executive, SD1 SFP and M3xxxA FPGA Firmware	5
How to run this programming example	6
Multi-Channel Sync Playback using M320xA Arbitrary Waveform Generators	7
Overview	7
Measurement Results	10
Getting Started with HVI Application Programming Interface (API)	11
System Definition	12
Define Platform Resources: Chassis, PXI triggers, Synchronization	13
Define HVI engines	14
Define HVI actions, events, triggers	14
Program HVI Sequences	15
Synchronized Multi-Sequence Block (a)	15
HVI Instruction: Front Panel Trigger ON/OFF (b)	17
Action Execute: AWG Trigger (c)	17
Compile, Load, Execute the HVI	17
Compile HVI	18
Load HVI to Hardware	18
Execute HVI	18
Release Hardware	18
Further HVI API Explanations	19
Multi-Chassis Setup Implementation	19
Add Chassis	21
Add M9031A Boards	21
10 MHz Clock Reference Source	21
Conclusions	25

KS2201A - Programming Example 1 - Multi-Channel Sync Playback using M320xA Arbitrary Waveform Generators

In this programming example, PathWave Test Sync Executive is used to program multiple M3xxxA AWGs to synchronously output first a FP (Front Panel) trigger pulse and afterwards a previously queued waveform. All modules run fully synchronized and actions across modules can be controlled with the timing resolution of the M3xxxA AWGs which is of 10ns.

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 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:
 - Keysight IO Libraries Suite 2020 (v18.1.25310.1 or later)
 - Keysight SD1 Drivers, Libraries and SFP (v3.00.95 or later)
 - Keysight PathWave Test Sync Executive Update 0.2 (v1.00.18 or later)
2. Chassis firmware and driver:
 - Keysight Chassis M9019A firmware (tested on v2018, v2019EnhTrig)
 - Keysight PXIe Chassis Family Driver (tested on v1.7.82.1)
3. M3xxxA with -HVx HW option and following FPGA firmware versions (to be installed using Keysight SD1 SFP):
 - M3202A AWG FPGA firmware (v4.00.95 or later)
 - M3201A AWG FPGA firmware (v4.02.65 or later)

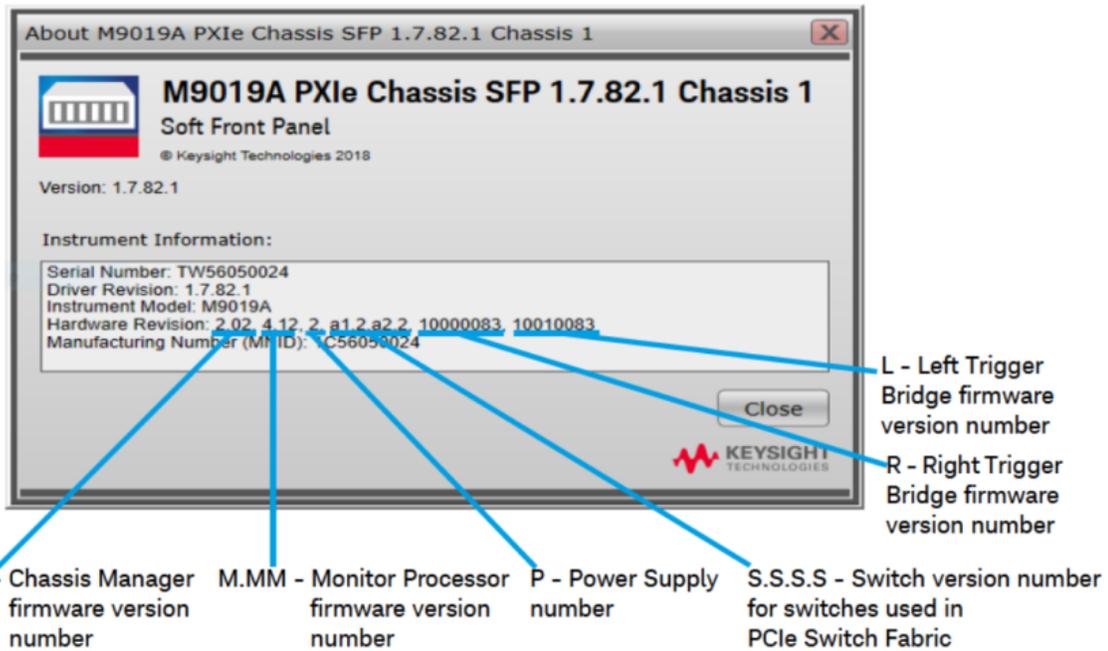
How to install Python 3.7.x 64-bit

This programming example requires you to install Python 64-bit version 3.7.x for all users. The Python installer can be downloaded from the Python webpage. Make sure you add Python 3.7.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.



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 run this programming example

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

```
# Simulated HW Option
hardware_simulated = True
```

Afterwards, it is necessary to specify the actual chassis number and slot number where the real PXI instruments are located. Model number of the used PXI instruments shall be updated, if different than the instrument model used in this programming example. This example uses PXI instruments from the Keysight M3xxxA family. The first step to control such instruments is to create an object using the `open()` method from the SD1 API. For a complete description of the SD1 API `open()` method and its options please consult the **SD1 3.x Software for M320xA / M330xA Arbitrary Waveform Generators User's Guide**.

Each PXI instrument is described in the code using a module description class that contains the module model number, chassis number, slot number and options. This programming example can be deployed on an arbitrary number of AWGs to be defined using the *module-descriptor* class. Please update the properties in each *module-descriptor* object before running the programming example:

```
# Update module descriptors below with your instruments information
module_descriptors = [
    config.ModuleDescriptor('M3202A', 2, 10, config.options, ""),
    config.ModuleDescriptor('M3202A', 2, 4, config.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 need to be also specified and listed by chassis number. In case of multi-chassis setup, please specify the connection between each pair of M9031 modules using the *M9031_descriptor* class.

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

# Multi-chassis setup
# In case of multiple chassis, chassis PXI lines need to be shared using M9031 PXI modules.
# M9031 module positions need to be defined in the program.
M9031_descriptors = [config.M9031Descriptor(1, 11, 2, 10)]
```

```

class M9031Descriptor:
    "Describes the interconnection between each pair of M9031A modules"    def __init__(
(self, first_M9031_chassis_number, first_M9031_slot_number, second_M9031_chassis_number,
second_M9031_slot_number):
    self.chassis_1 = first_M9031_chassis_number
    self.slot_1 = first_M9031_slot_number
    self.chassis_2 = second_M9031_chassis_number
    self.slot_2 = second_M9031_slot_number

```

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
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.

Multi-Channel Sync Playback using M320xA Arbitrary Waveform Generators

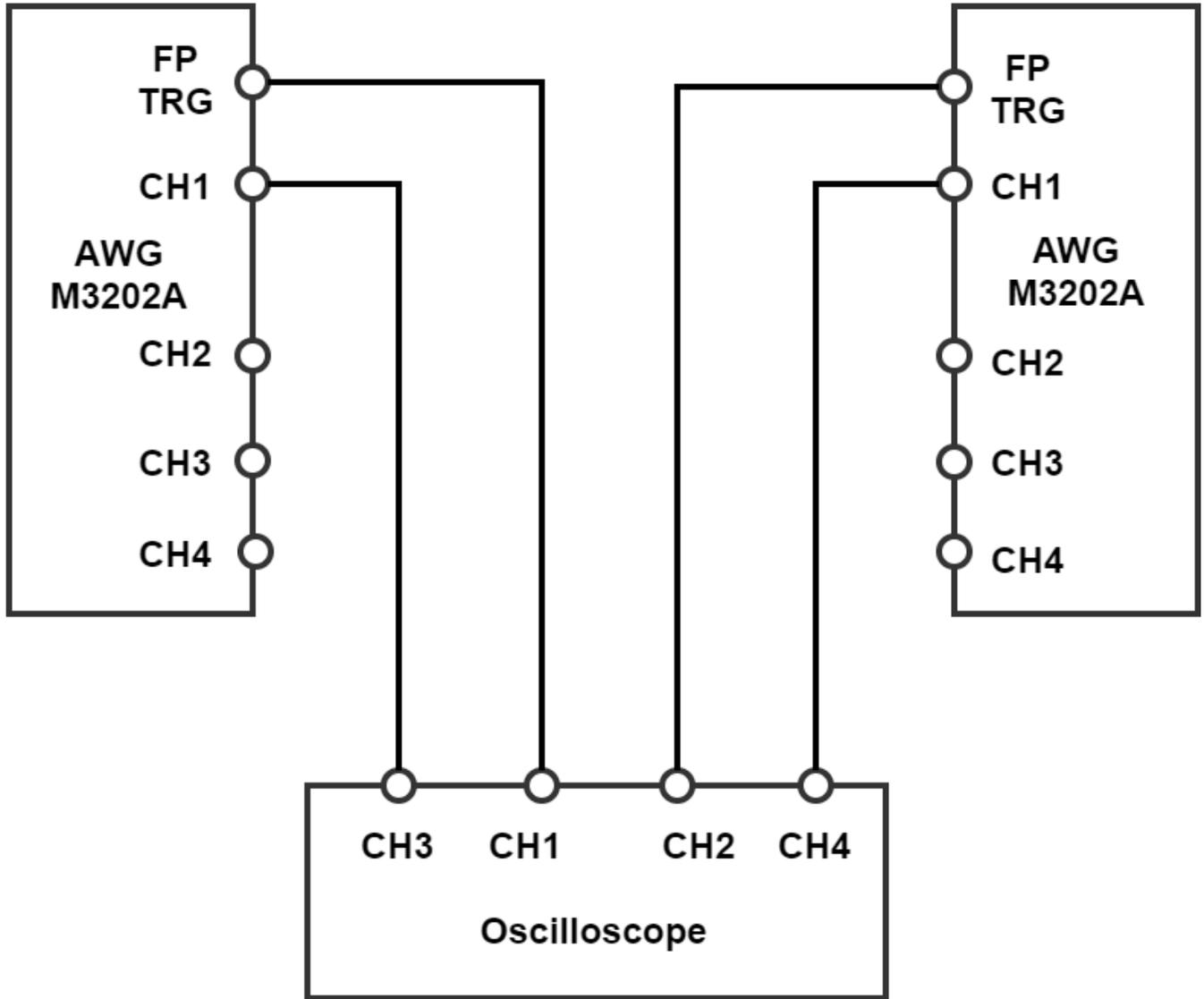
Overview

This programming example illustrates how to deploy HVI to synchronously generate electronic signals from multiple channel across an arbitrary number of different 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.

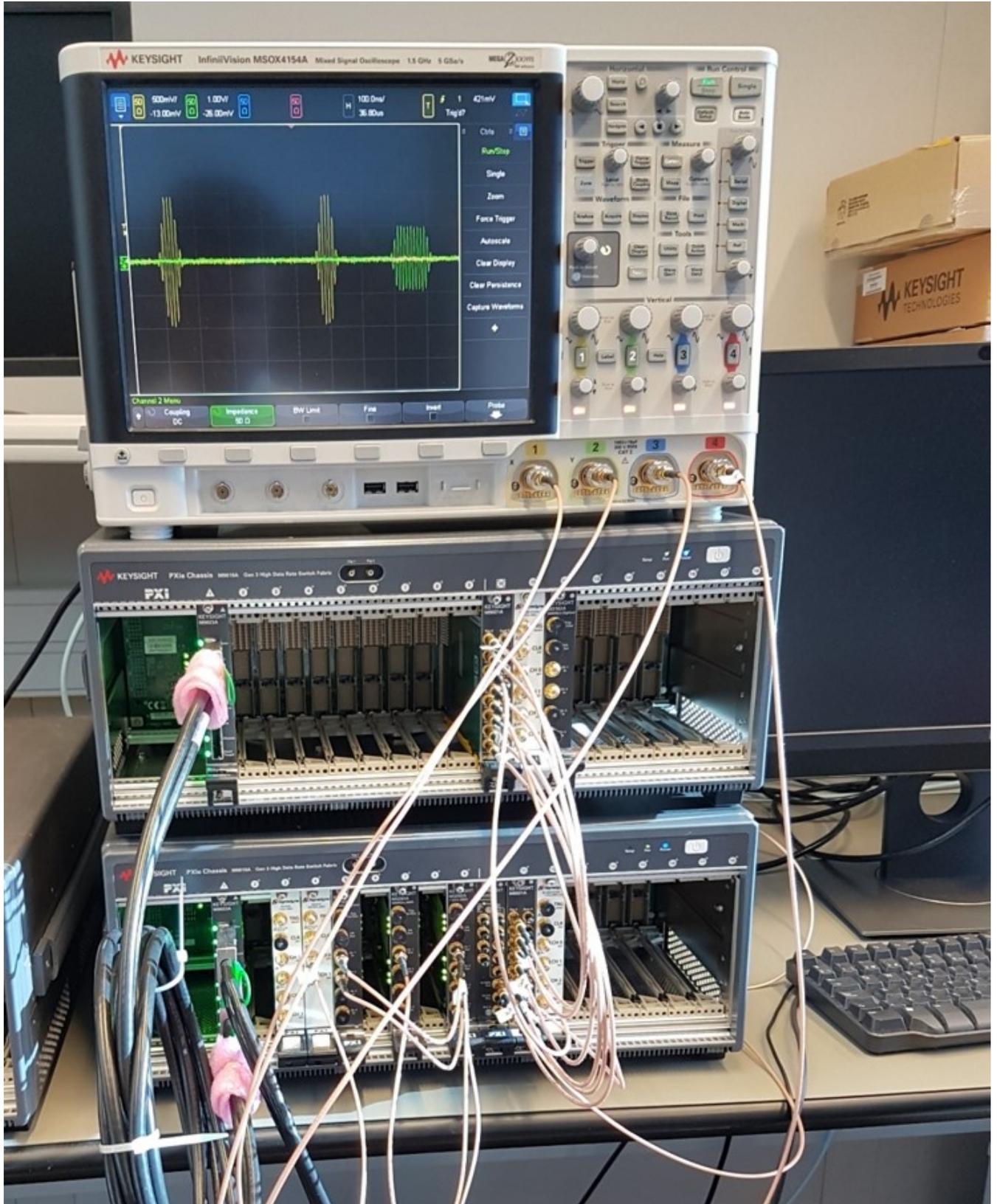
This programming example illustrates the following HVI functionalities:

1. How to create an HVI sequence using the HVI API
2. Synchronized Multi-Sequence Block
3. Module synchronization using Synchronized Multi-Sequence Blocks
4. HVI Native Instructions
5. Instrument action execution within HVI sequences

The programming example capabilities will be 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.

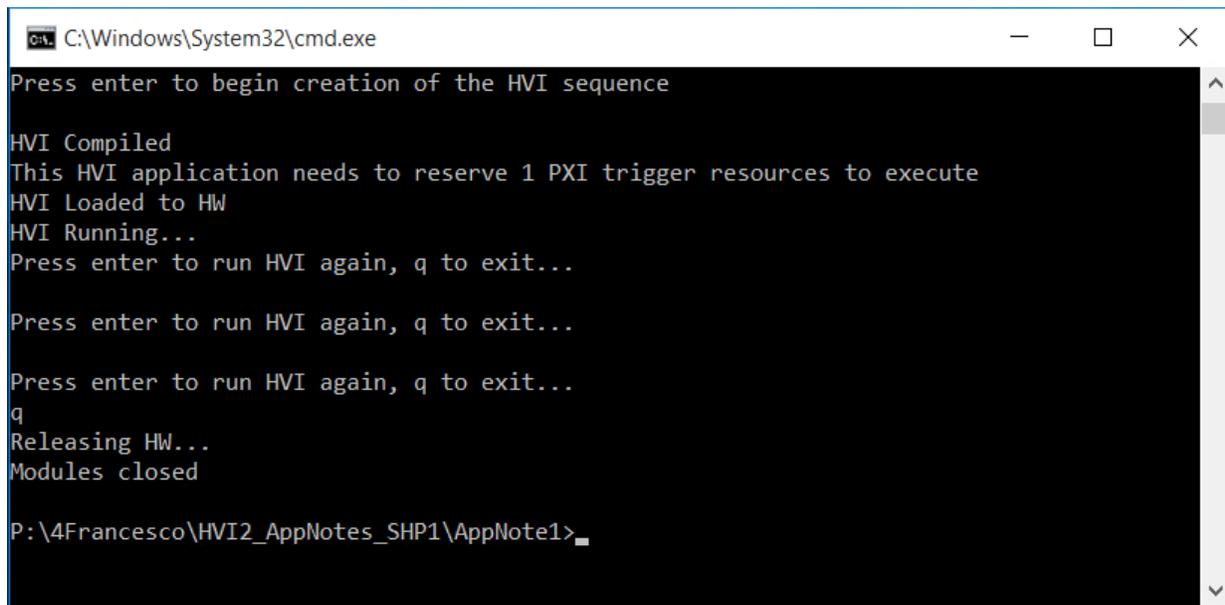


A photograph of the measurement setup used for the measurement results reported in this programming example is also reported below:



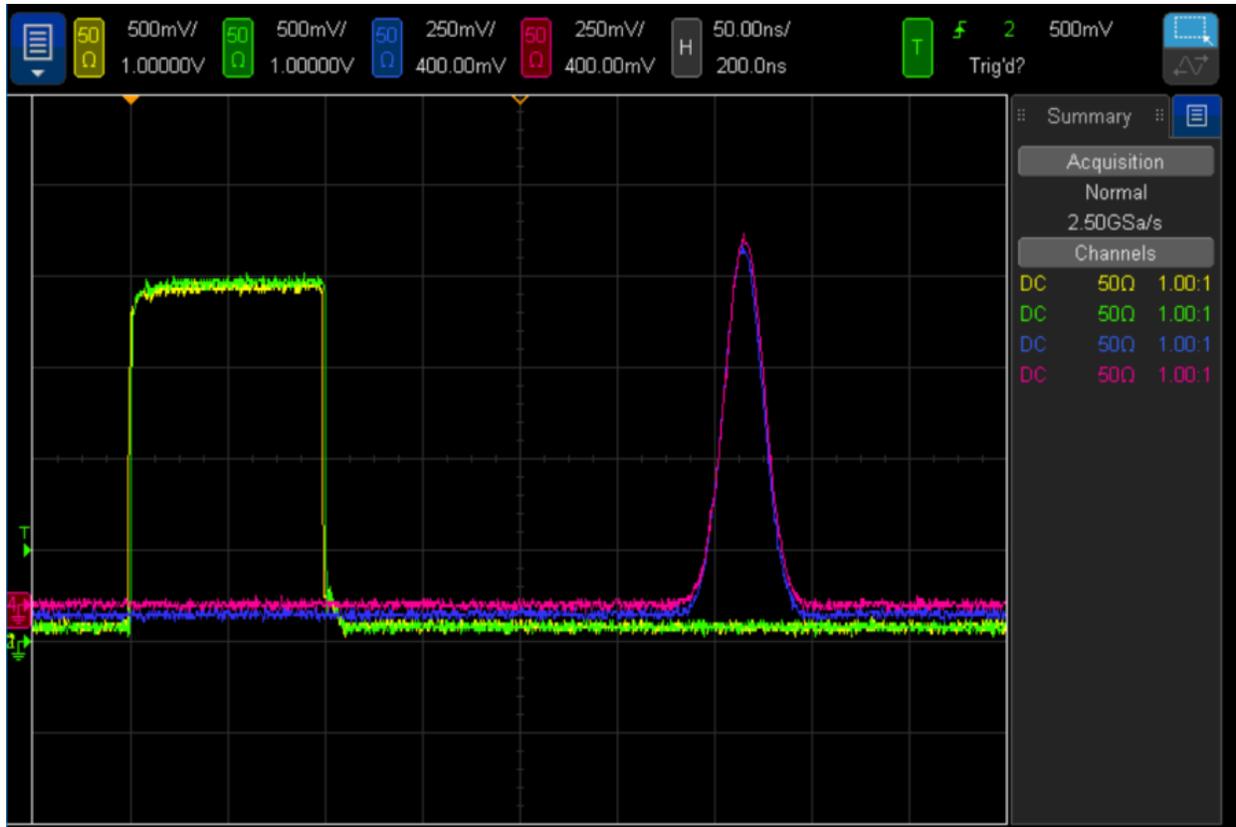
Measurement Results

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



```
C:\Windows\System32\cmd.exe
Press enter to begin creation of the HVI sequence
HVI Compiled
This HVI application needs to reserve 1 PXI trigger resources to execute
HVI Loaded to HW
HVI Running...
Press enter to run HVI again, q to exit...
Press enter to run HVI again, q to exit...
Press enter to run HVI again, q to exit...
q
Releasing HW...
Modules closed
P:\4Francesco\HVI2_AppNotes_SHP1\AppNote1>
```

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 AWG 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.

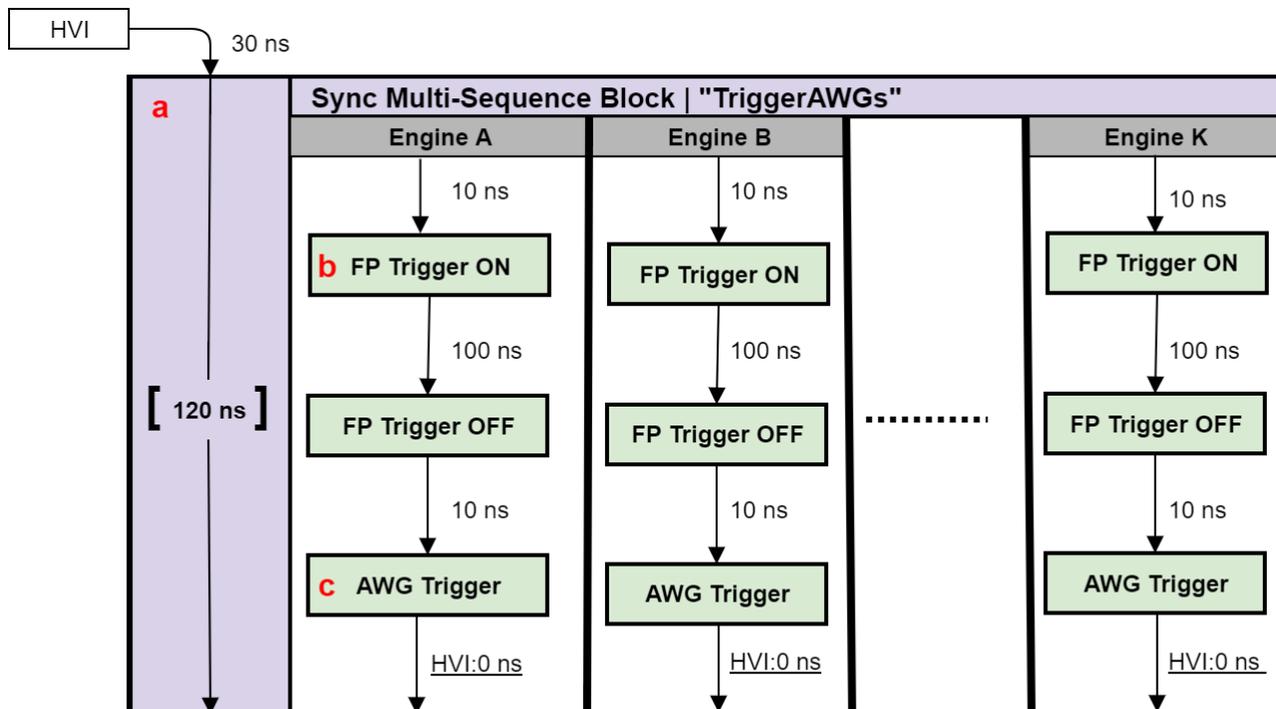


AWG Trigger Delay

Please note that the HVI sequences represented in the HVI diagram contained in the next section specify the "AWG Trigger" instruction 10 ns after the the "FP Trigger OFF" instruction. However, users must take into account that the AWG instrument requires time to process the AWG trigger action and propagate the command through its digital HW before the first waveform sample can appear at the AWG output. This processing time can be called AWG Trigger Delay and it explains why in the previously presented scope measurements there is a delay of about 150-170 ns between the FP trigger falling edge and the first sample of the Gaussian waveform generated by the AWG. For exact values of AWG Trigger Delay and other AWG specs please consult the documentation of Keysight M3xxxA AWGs

Getting Started with HVI Application Programming Interface (API)

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 using HVI API. The sequence of operations executed by each of the instruments using HVI technology are explained in the flow charts below: Every HVI statement is presented below with a letter referencing to the equivalent block in the HVI flowchart, to facilitate users to understand the new model. The programming is parametrized to run on an arbitrary number of AWGs. All AWGs opened in the programming are automatically added to the HVI sequence.



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

To deploy HVI into an application, three fundamental steps shall be followed:

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

The following sub-sections describe in details 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* allows to define all necessary HVI resources. HVI resources include all the platform resources, engines, triggers, registers, actions, events, etc. that the HVI sequences are going to use and execute. Users need to declare them upfront and add them to the corresponding collections. All HVI Engines included in the programming need to be registered into the *EngineCollection* class instance. HVI resources are described in details in the [PathWave Test Sync Executive User Manual](#). The HVI resource definitions are summarized in the code snippets below.

Python

```

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

def define_hvi_resources(sys_def, module_dict, chassis_list, M9031_descriptors, pxi_sync_
trigger_resources):
    """    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, chassis_list, M9031_descriptors, pxi_sync_trigger_
resources)
    # 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)
    # Defines the trigger resources
    define_hvi_triggers(sys_def, module_dict)

```

Define Platform Resources: Chassis, PXI triggers, Synchronization

All HVI instances need to define the chassis and eventual chassis interconnections using the *SystemDefinition* class. PXI trigger lines to be reserved by HVI for its execution can be assigned using the *sync_resources* interface of the *SystemDefinition* class. *SystemDefinition* class also allows you to add additional clock frequencies that the HVI execution can synchronize with. For further information please consult the section "HVI Core API" of the **PathWave Test Sync Executive User Manual**.

Python

```

def define_hw_platform(sys_def, chassis_list, M9031_descriptors, pxi_sync_trigger_
resources):
    """    Define HW platform: chassis, interconnections, PXI trigger resources,
synchronization, HVI clocks
    """    config = ApplicationConfig()
    # Add chassis resources
    for chassis_number in chassis_list:
        if config.hardware_simulated:
            sys_def.chassis.add_with_options(chassis_number,
'Simulate=True,DriverSetup=model=M9018B,NoDriver=True')
        else:
            sys_def.chassis.add(chassis_number)
    # Add M9031 modules for multi-chassis setups
    if M9031_descriptors:
        interconnects = sys_def.interconnects
        for descriptor in M9031_descriptors:
            interconnects.add_M9031_modules(descriptor.chassis_1, descriptor.slot_1,
descriptor.chassis_2, descriptor.slot_2)
    # Assign the defined PXI trigger resources
    sys_def.sync_resources = 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

```
class HVI_resource_Names:
    # Defines the Names of HVI engines, actions, triggers used in this programming
    def __init__(self):
        self.engine_Name = 'AwgEngine'
        self.awg_trigger_Name = 'AwgTrigger'
        self.fp_trigger_Name = 'FpTrigger'

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

Python

```
class HVI_resource_Names:
    # Defines the Names of HVI engines, actions, triggers used in this programming example
    def __init__(self):
        self.engine_Name = 'AwgEngine'
        self.awg_trigger_Name = 'AwgTrigger'
        self.fp_trigger_Name = 'FpTrigger'

def define_hvi_actions(sys_def, module_dict):
    # 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
    # Previously defined resource Names
    config = ApplicationConfig()
    # 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):
    " Defines and configure the FP trigger output of each AWG "      # Previously defined
    # resources
    config = ApplicationConfig()
    # Add to the HVI Trigger Collection of each HVI Engine the FP Trigger object of that
    # same instrument
    for engine_Name in module_dict.keys():
        fp_trigger_id = module_dict[engine_Name].instrument.hvi.triggers.front_panel_1
        fp_trigger = sys_def.engines[engine_Name].triggers.add(fp_trigger_id, config.fp_
        trigger_Name)
        # Configure FP trigger in each hvi.engines[index]
        fp_trigger.config.direction = kthvi.Direction.OUTPUT
        fp_trigger.config.polarity = kthvi.Polarity.ACTIVE_HIGH
        fp_trigger.config.sync_mode = kthvi.SyncMode.IMMEDIATE
        fp_trigger.config.hw_routing_delay = 0
        fp_trigger.config.trigger_mode = kthvi.TriggerMode.LEVEL
        #NOTE: FP trigger pulse length is defined by the HVI Statements that control FP
        # Trigger ON/OFF

```

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_trigger_sequence(sequencer, module_dict):
    " Programs the MIMO Trigger HVI sync sequence"      # Add a Sync Multi-Sequence Block
    # (SMSB)
    sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs", 30)
    # Program the SMSB to trigger AWGs
    program_trigger_awgs(sync_block, module_dict)

```

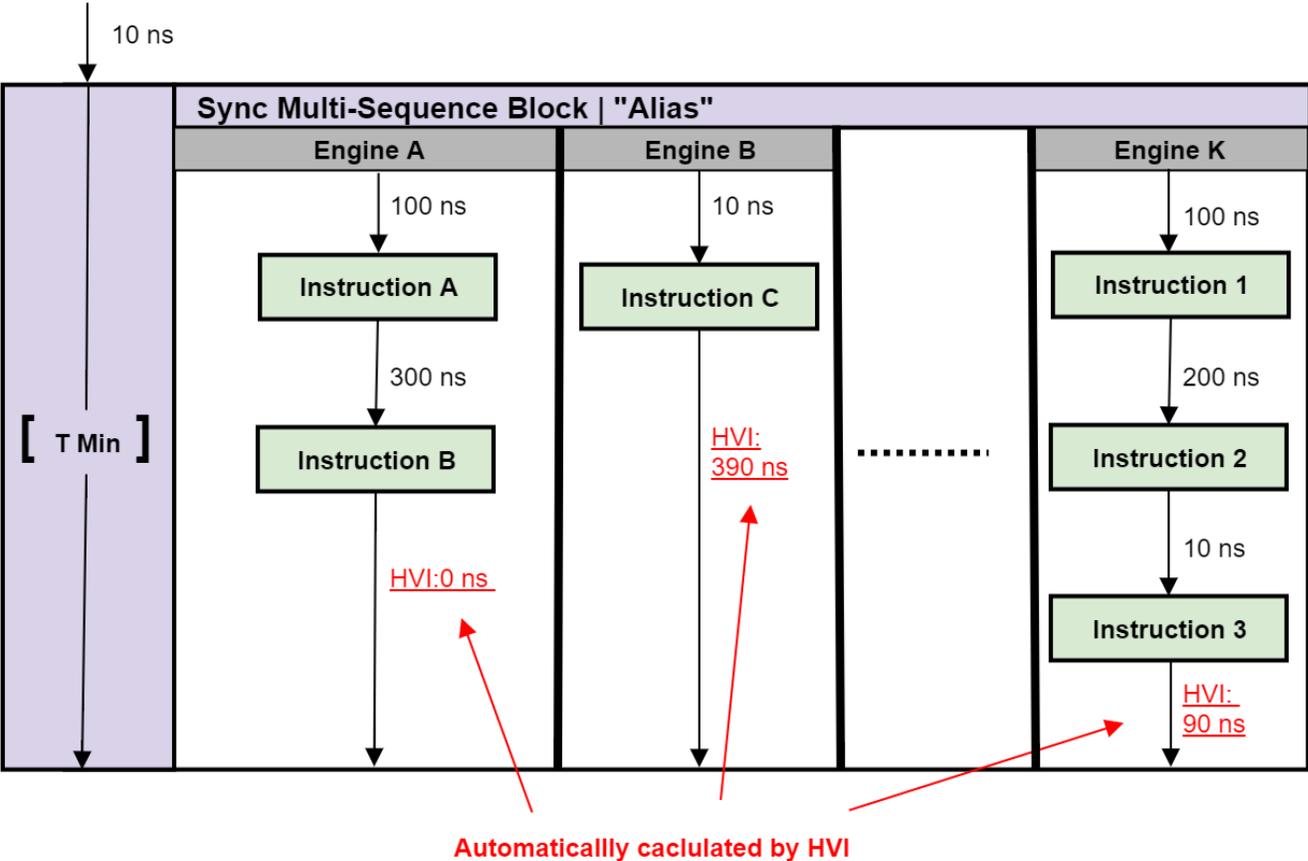
Synchronized Multi-Sequence Block (a)

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.

Python

```
# Program Sync Multi-Sequence Block
sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("TriggerAWGs", 30)
```

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. In future releases, the user will be able to

specify specific execution time values or allowed ranges. 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 case of duration "T min" HVI will automatically add no time to the local sequence having 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 M3xxxA family. For further explanations about the timing of HVI sequence execution please refer to "HVI Timing" section of the [KS2201A PathWave Test Sync Executive User Manual](#) available on www.keysight.com

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

This block executes a native HVI instruction. Native HVI instructions are common to every Keysight product. API method `add_instruction()` allows 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

```
# Write FP Trigger ON
fp_trigger = sequence.engine.triggers[config.fp_trigger_Name]
trigger_write = sequence.instruction_set.trigger_write
instr_trigger_ON = sequence.add_instruction("FP Trigger ON", 10, trigger_write.id)
instr_trigger_ON.set_parameter(trigger_write.trigger.id, fp_trigger)
instr_trigger_ON.set_parameter(trigger_write.sync_mode.id, trigger_write.sync_mode.IMMEDIATE)
instr_trigger_ON.set_parameter(trigger_write.value.id, trigger_write.value.ON)
```

Action Execute: AWG Trigger (c)

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)
```

Compile, Load, Execute the HVI

Once the HVI sequences are programmed by defining all the necessary HVI statements, users 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 to 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 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 sequence 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 2020\api\python\Help\index.htm>.

Multi-Chassis Setup Implementation

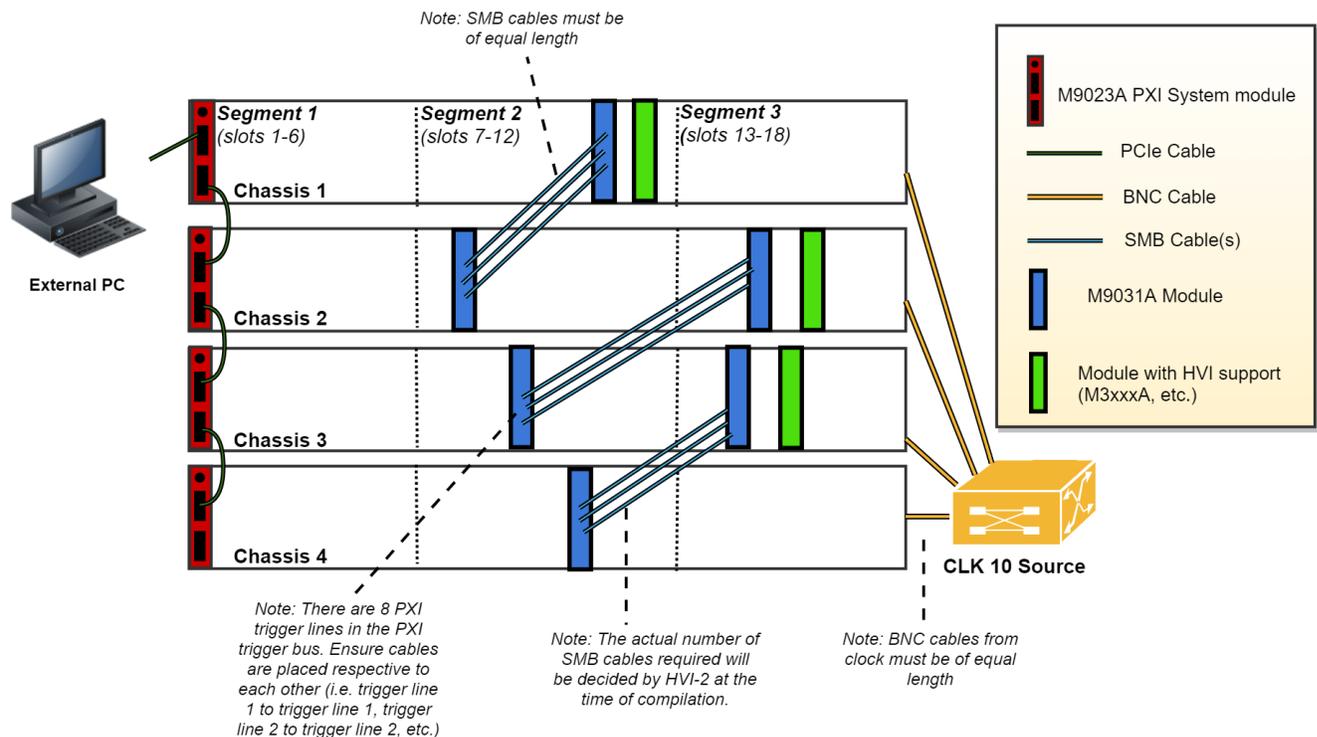
The reference examples provided with this document can be executed on a multiple-chassis setup with only the few modifications explained below. In a multi-chassis setup, it is necessary to interconnect the PXI triggers and clocking of the multiple chassis.

With the currently available infrastructure to interconnect PXIe backplane triggers a pair of M9031A boards must be placed in a specific segment in each chassis to be interconnected.

NOTE The SMB cables used to connect the M9031A modules need to be as short as possible. The chassis need to be stack in the same rack, on top of each other, as close as possible to each other to allow the SMB cables that connect them to be as short as possible.

On the two M9031A boards, the connectors corresponding to the same PXI line(s) are connected between each other. There are mainly three rules to consider when choosing the chassis slot where to place a M9031A board:

1. Only one M9031A board can be placed in a chassis segment. M9031A boards are connected in pairs. Each pair of M9031A connects two chassis together and shares info through their PXI lines.
2. If no other M9031A board is already placed in the central segment, then the M9031A board should be placed there as a preferred choice, to minimize the signal path length.
3. A PXI module included in the HVI application needs to be placed in the same chassis segment where the first M9031A board of each pair is placed, in order to control the exchange of PXI line values through the pair of boards.



The picture above illustrates in green the PXI modules that must be placed in the same segment as the M9031A modules in blue. Basically:

- The 1st chassis must include a M9031A together with a PXI module with HVI in segment 2
- All Middle chassis must have a M9031A in the segment 2, and a M9031A together with a PXI Module with HVI support in Segment 3
- The last chassis must include a M9031A in segment 2.

All the chassis that are part of the multi-chassis setup should be connected in a daisy chain. Chassis connections with M9031A are made to share the PXI lines that are used as sync resources. PXI trigger lines are shared using M9031A boards, connecting the ports corresponding to the same PXI line on both M9031A boards. The first and last chassis of the daisy chain each require one M9031A board; all the middle chassis in the daisy chain require two M9031A boards. A multi-chassis including N chassis requires a number of M9031A boards equal to $2*(N-1)$.

Additionally, a very clean 10 MHz source should be used to provide the same reference signal to all chassis. One option is to use a multi-output 10 MHz source, for best performance probably driven by an atomic clock, connecting each output to the 10 MHz reference input of each chassis using cables that have the same length. It is extremely important for the correct operation of HVI and in particular for synchronization that all chassis are running with their CLK10 and CLK100 fully locked and aligned, the skew between these clocks in the different chassis will result in skew in the instrument operation.

Add Chassis

Each chassis included in the multi-chassis setup can be added using any of the HVI API methods below. The AddAutoDetect() method shall be called only once to automatically detect and add all the chassis connected to the system.

Python

```
# To add chassis resources use:
hvi.platform.chassis.add_with_options(1,
'Simulate=True,DriverSetup=model=M9018B,NoDriver=True')
hvi.platform.chassis.add(chassis_number)
hvi.platform.chassis.add_auto_detect()
```

Add M9031A Boards

In the HVI API each M9031A board pair needs to be declared using the following software method:

Python

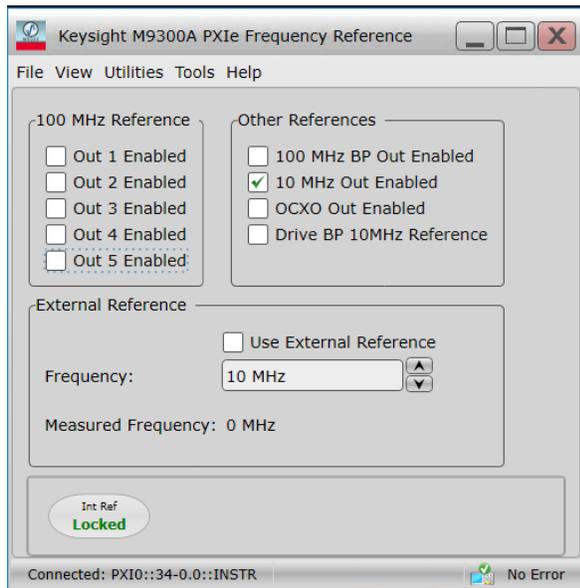
```
# To add each interconnected pair of M9031 modules use:
interconnects.add_M9031_modules(1st_M9031_chassis_number, 1st_M9031_chassis_slot, 2nd_
M9031_chassis_number, 2nd_M9031_chassis_slot)
```

The above-mentioned code lines are part of this application code example and they can be used to adapt the code example to run on a multi-chassis setup.

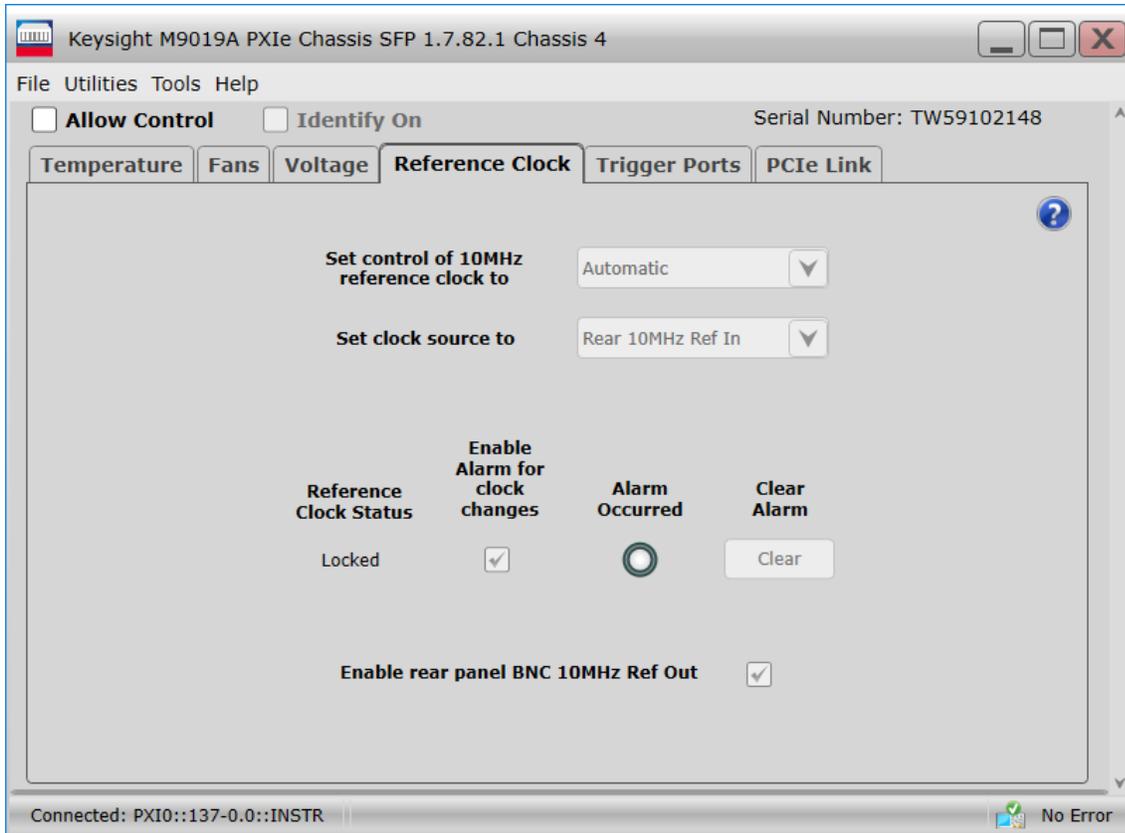
10 MHz Clock Reference Source

One option is to use as a 10 MHz Reference source the PXI module **Keysight M9300A PXIe Frequency Reference**. Please place this module in one of the chassis and use splitters to divide the 10 MHz clock output into N cables to be connected to the 10 MHz REF IN connector on the back panel of each of the chassis, including the chassis where the M9300A module is placed. Each time the system is restarted please open the M9300A SFP software to check the box "10 MHz Out Enabled and uncheck the box ""Drive BP 10 MHz

Reference". Please see screenshot below for clarifications. For more details on the Keysight M9300A PXIe Frequency Reference please visit www.keysight.com.



Once the common 10 MHz reference source is setup, the Chassis SFP can be used to verify that each chassis is correctly receiving the common external reference signal. This can be done from the "Reference Clock" window shown in the screenshot below. Once you open the window please clear any "Alarm" that possibly occurred during the 10 MHz reference setup. After clearing "Alarm occurred" icon should stay idle (white color). Clock source shall st to "Rear 10 MHz Ref In".



Additionally, in the case of using a remote controller card, like the M9023A PXI System Module used in this application, it is possible to see the backplane status LEDs that also indicate the correct clocking. On the chassis backplane REF and LOCK LED lights are lighted in green when the chassis is correctly locked to the external reference signal. By checking the LED lights on the backplane of each chassis users can ensure the 10 MHz reference is correctly shared among the different chassis. Please see picture below showing the LED lights on the chassis backplane, visible from the front panel by removing the panel in the chassis slot that is preceding chassis slot 1.



For more details on the Keysight PXIe Chassis Family please visit www.keysight.com .

Conclusions

This Programming Example explained how to use Pathwave Test Sync Executive and HVI (Hard Virtual Instrument) technology to synchronize multiple AWG to concurrently issue first a marker pulse from the Front Panel (FP) trigger port and then play a previously loaded waveform from all their AWG channels. The application use case illustrated here can be tested on any AWG of the Keysight M3xxxA PXI family. HVI technology was deployed using the HVI API (Application Programming Interface). Example measurement results demonstrated synchronized FP trigger marker output and waveform output from multiple AWGs.