PROGRAMMING EXAMPLE 2

Synchronous Mixed-Signal Measurements using M3xxxA PXI Instruments

PATHWAVE

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.

KEYSIGHT TECHNOLOGIES

Table of Contents

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

In this programming example a 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.

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).

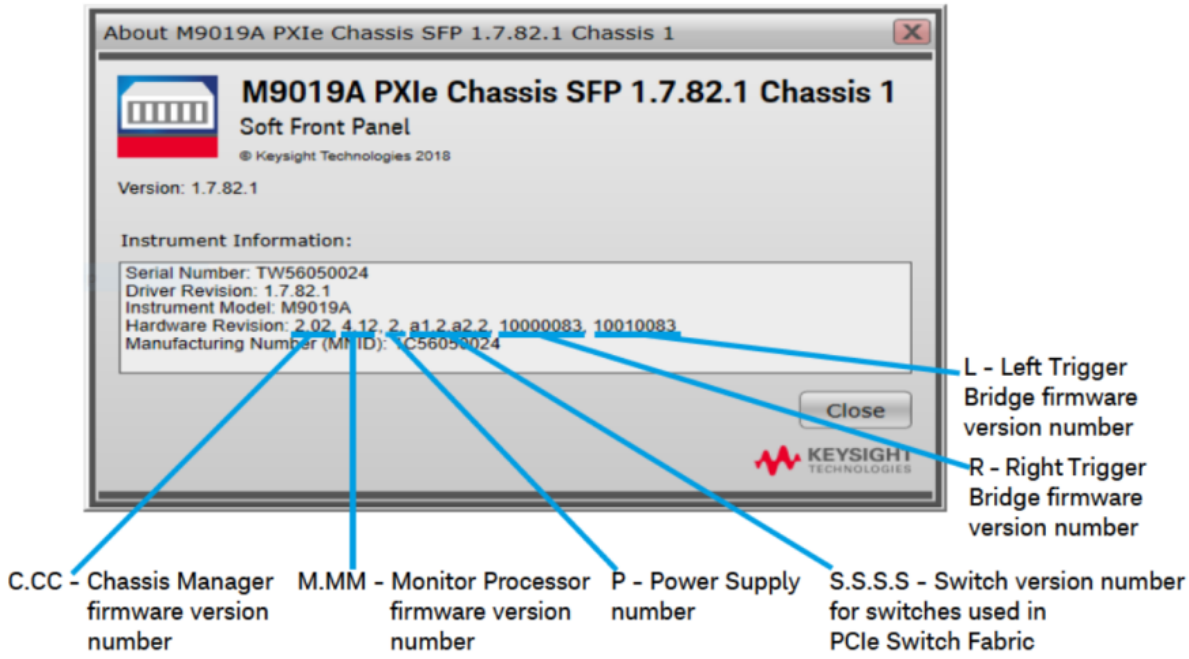
1. Software versions required:
   - Keysight IO Libraries Suite 2020 (v18.1.25310.1 or later)
   - Keysight SD1 Drivers, Libraries and SFP (v3.00.95 or later)
   - Keysight PathWave Test Sync Executive Update 0.2 (v1.00.18 or later)
2. Chassis firmware and driver:
   - Keysight Chassis M9019A firmware (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)
   - M3102A Digitizer FPGA firmware (v2.01.40 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. Please update the properties in each *module-descriptor* object before running the programming example:

```
# Define instruments, chassis, interconnects
hvi_eng_Names = HVI_engine_Names()
# Update module descriptors below with your instruments information
digitizer_descriptor = module_descriptor('M3102A', 1, 9, options, hvi_eng_Names.dig_engine)
rf_gen_descriptor = module_descriptor('M3202A', 1, 8, options, hvi_eng_Names.rf_gen_engine)
# AWG1 to be used as an RF Pulse Gen.
awg_descriptors = [module_descriptor('M3202A', 1, 7, options, hvi_eng_Names.awg_engine)]
# Assign AWG engine Names to AWG1-AWGN in case more than 2 AWGs are used
for index in range(len(awg_descriptors)):
    awg_descriptors[index].engine_Name = hvi_eng_Names.awg_engine + str(index)
```

```
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.

Application-specific parameters necessary to configure the digitizer and the AWGs are listed in dedicated classes. Before running the programming example, users shall update, if necessary, the AWG and digitizer parameters contained in the code classes listed below. Measurment results reported in this documents were obtained using the parameters value reported in the following code snippets.

```
class AWG_parameters:
    # Configures AWG for waveform generation
    def __init__(self):
        # 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 value of AWGqueueWfm() [ns]
        self.awgtrigger_latency = 2000 # [ns]
        self.wfm_length = 100 # [ns]


class RF_pulse_parameters:
    # Configures RF pulse generator parameters
    def __init__(self):
        self.all_ch_mask = 0xF # binary mask definig 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 # [V]
        self.OFF_value = 0 # [V]
        self.n_AWG = 1 # channel number to be used as RF Gen
        self.pulse_ontime = 200 # [ns]
        self.pulse_offtime = 150 # [ns]
        self.pulse_delay = 100 # [ns]


class DIG_parameters:
    # Configures Digitizer parameters
    def __init__(self):
        rfgen_params = RF_pulse_parameters()
        all_ch_mask = 0xF
        sampling_time = 2 # [ns] 1/sample_rate, sample_rate = 500 MSa/s for Digitizer
M3102A
        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_points_per_cycle = acquisition_points_per_cycle
        self.acquisition_time_per_cycle = acquisition_points_per_cycle*sampling_time
        self.num_cycles = rfgen_params.num_loops #insert -1 for infinite cycles
        self.acquisition_points = int(acquisition_points_per_cycle*rfgen_params.num_loops)
        self.acquisition_delay = 150 # x2 [ns]
        self.trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG
        self.mask = all_ch_mask
```

Synchronous Signal Generation & Acquisition using M3xxxA PXI Instruments

This programming example illustrates the following functionalities:

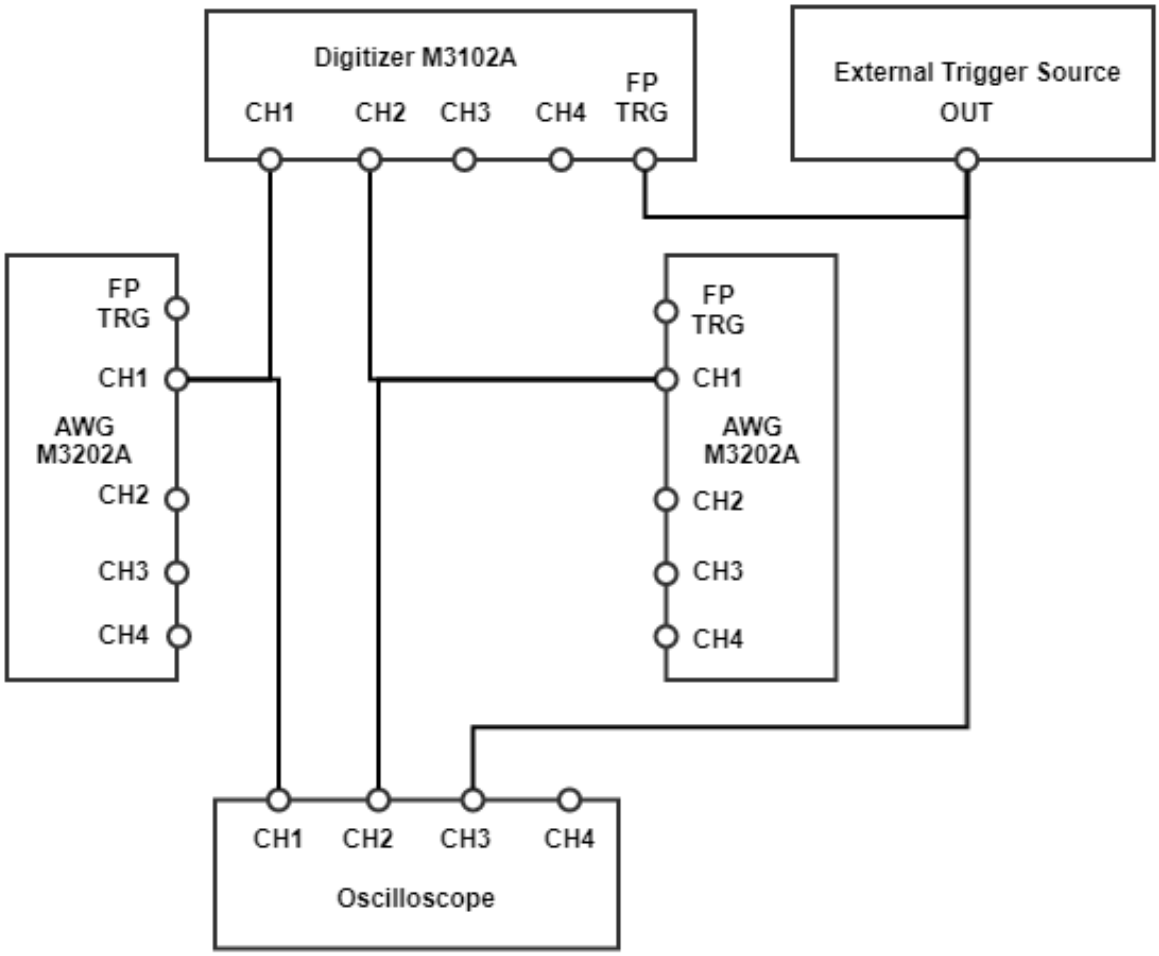
1. Synchronized While Global Statement
2. Wait-for-event Statement
3. Use of registers and scopes

4. Local Flow Control Statements: WHILE loop, IF loop
5. HVI Product-specific Instructions

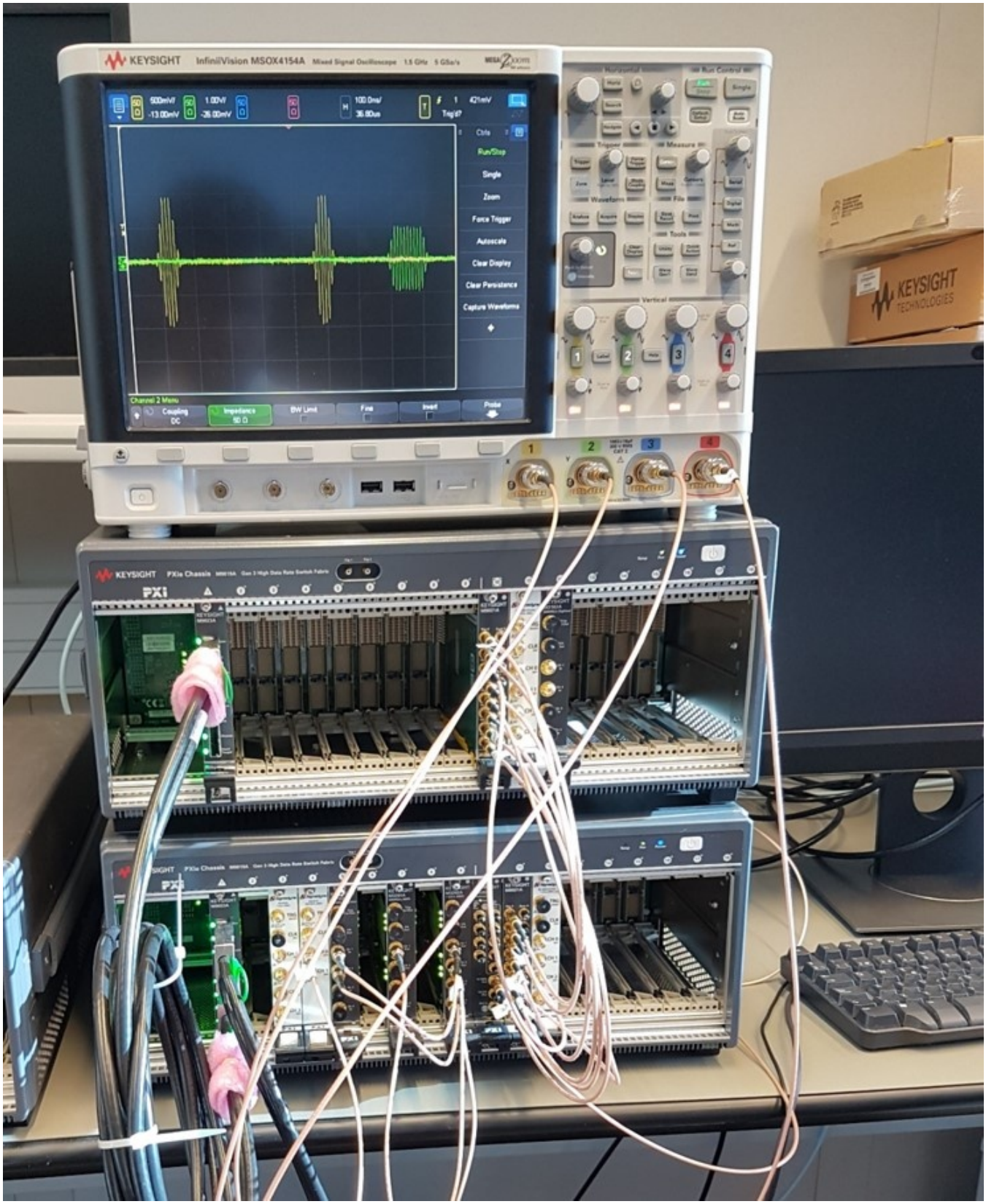
AWG signal generation is controlled using local flow control loops. This way, a train of RF pulses can be generated and previously loaded arbitrary waveforms can be queued and played. An HVI Synchronized While Statement controls the digitizer acquisitions and enables synchronization of each acquisition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

Measurement Results

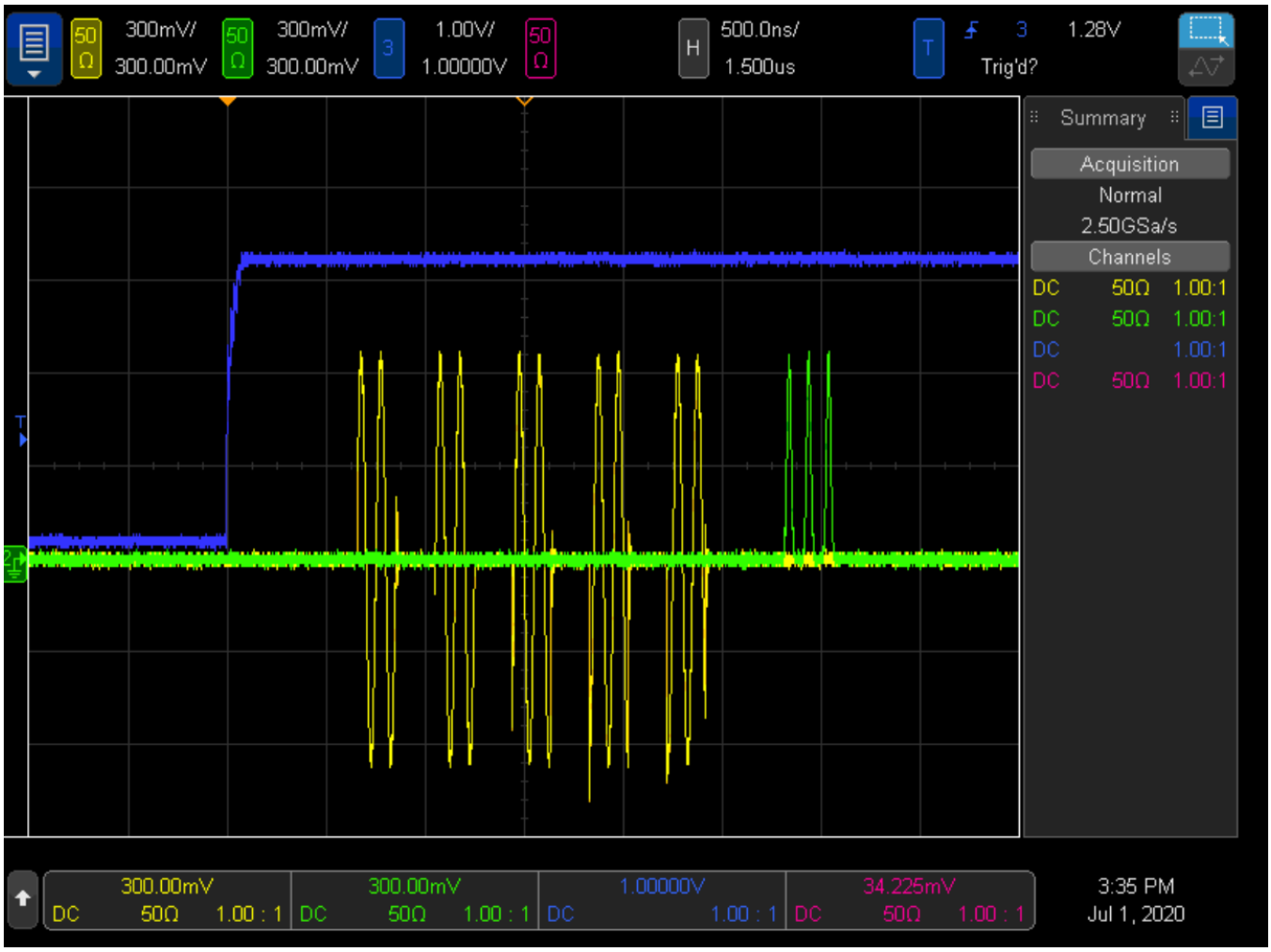
This section describes the measurement results obtained by deploying this programming example on a setup including two M9019A PXI chassis, an M3102A digitizer and two M3202A AWGs. A block diagram of the measurement setup used in this documented is reported below.



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

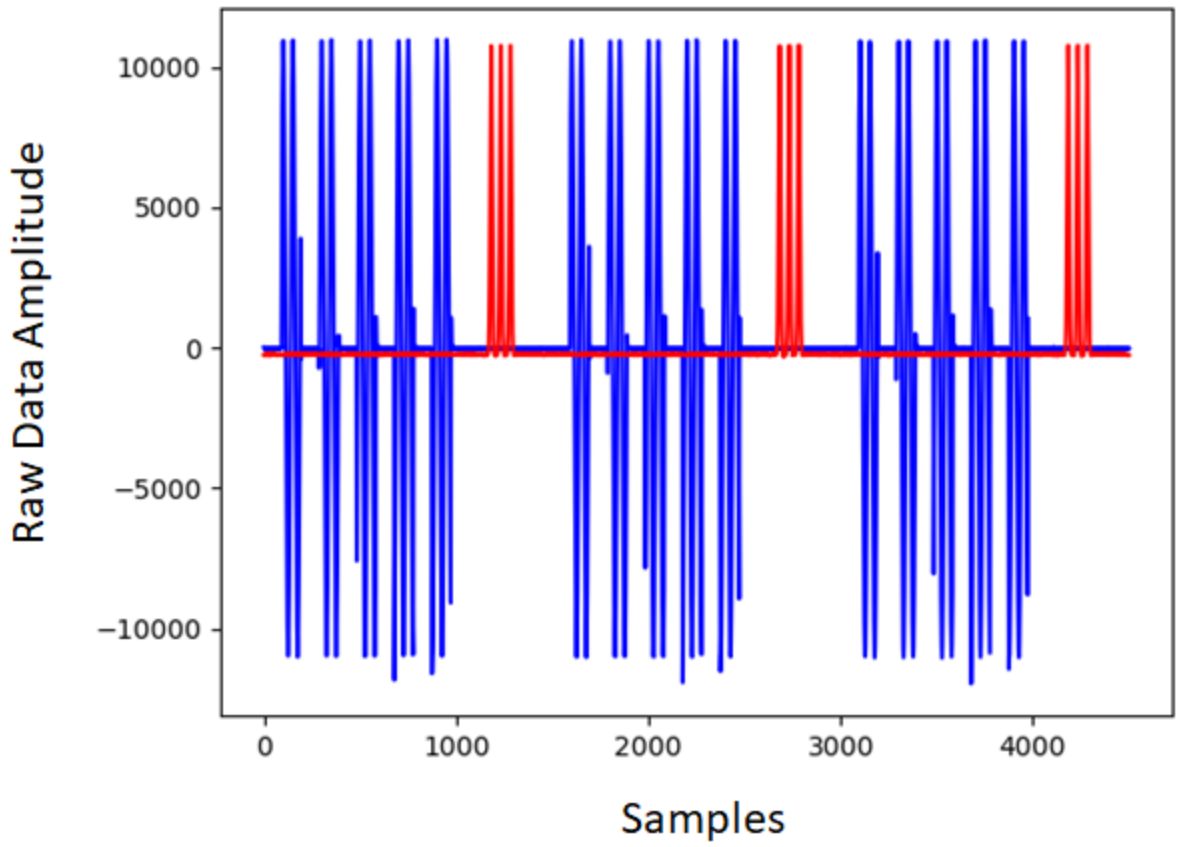

KEYSIGHT
InfiniiVision MSOX4154A
Mixed Signal Oscilloscope
Run/Stop
Single
Zoom
Force Trigger
Autoscale
Clear Display
Clear Persistence
Capture Waveforms
KEYSIGHT
PXIe Chassis
PXI
KEYSIGHT
TECHNOLOGIES

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

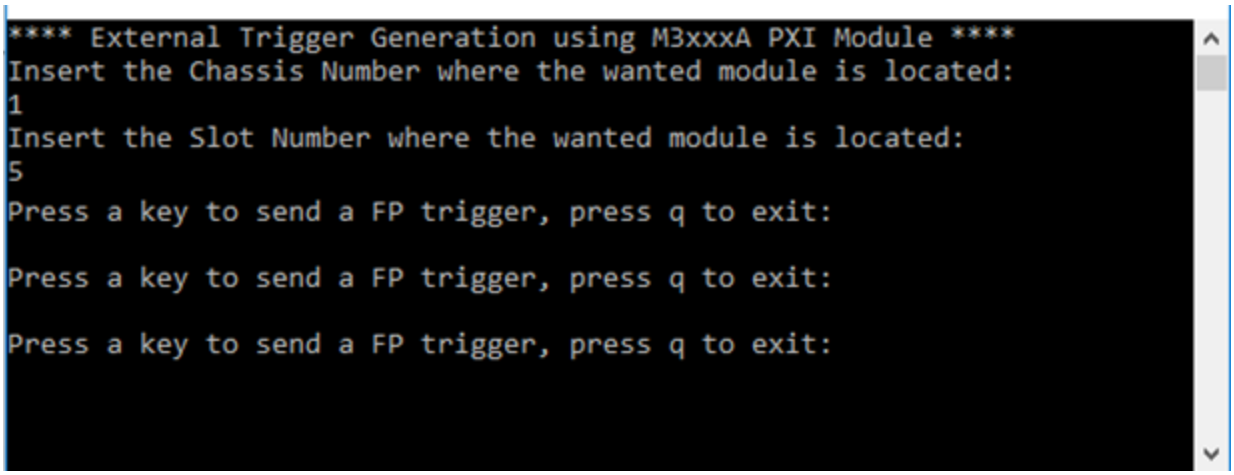
DAQ2 channel measurements obtained from the AWG output that generates arbitrary waveforms selected by the user.



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

```
OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS 35

Press enter to program the HVI sequence and execute it

HVI Compiled
This HVI application needs to reserve 2 PXI trigger resources to execute
HVI Loaded to HW
HVI Running...

####### Application Parameters #######
Number of HVI loops: 3
Number of RF Pulses: 3
RF Pulse ON Time: 200 [ns]
RF pulses frequency: 10 Mhz
AWG waveforms loaded to RAM: 2
Digitizer configured to acquire 1000 points in 3 cycles
Total dig. acquisition_points = points_per_cycle*num_cycles = 3000
########################################
Please connect your External Trigger source to the FP TRIG_OUT connector of DIG module in chassis 1, slot 11
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!
Releasing HW...
Modules closed
```

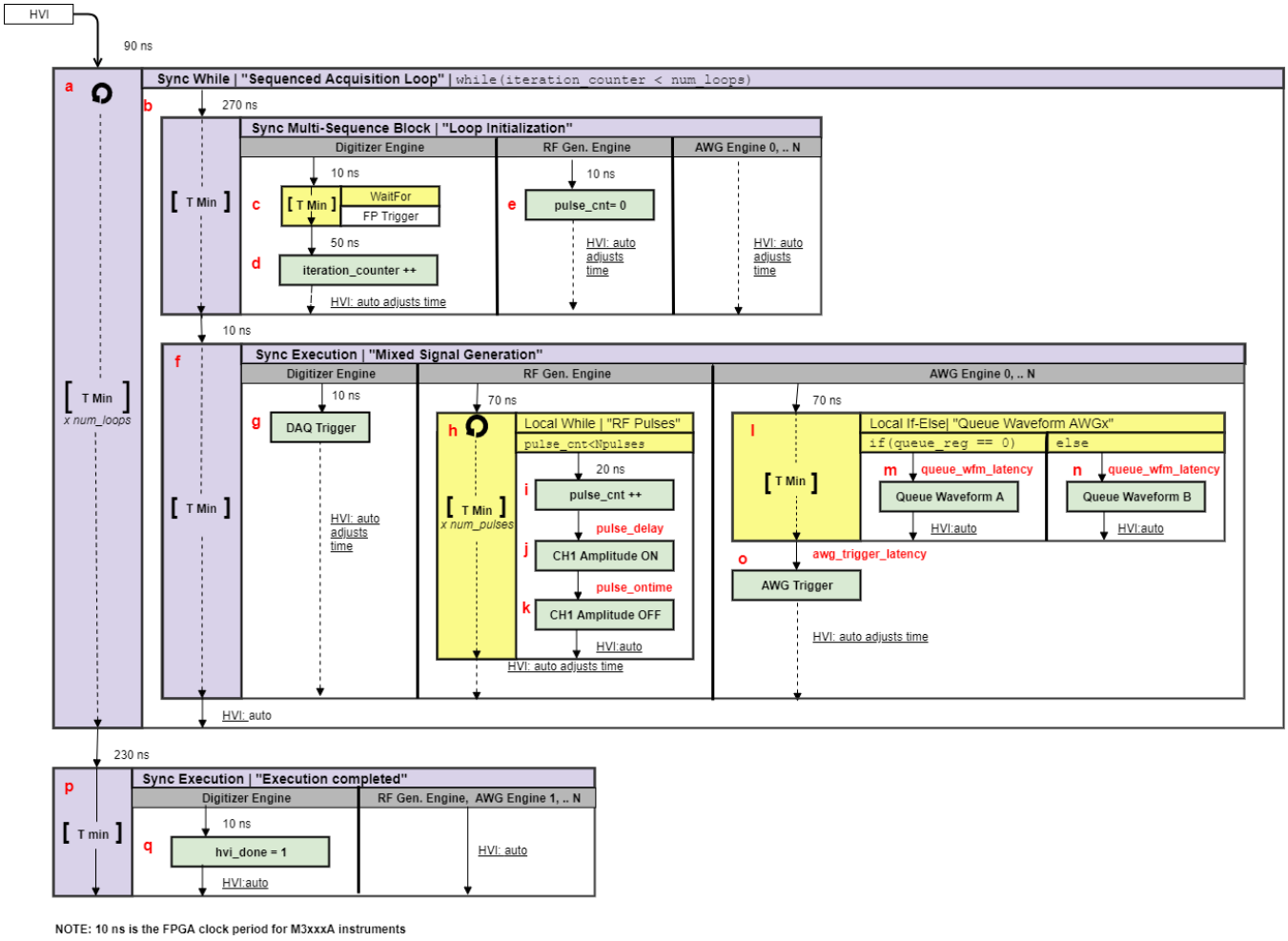
With this programming example, a provided executable GenExtTrigger_M3xxxA.exe can be used to generate the FP triggers from any M3xxxA AWG module that is external to the HVI application. It is used here to emulate the external trigger. An example execution on the console terminal of this independent executable for FP trigger generation is displayed in the screenshot below.



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 example using HVI API. The sequence of operations executed by each of the instruments using

HVI technology are 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. The programming example is parametrized to run on an arbitrary number of AWGs. Any additional AWGs after the second one will execute the same HVI sequence as the ones executed by the "AWG Engine 2" depicted below. For further explanations about the elements in the diagram below, please refer to the **PathWave Test Sync Executive User Manual**.



NOTE: 10 ns is the FPGA clock period for M3xxxA instruments

NOTE The Python Variables *pulse_delay* and *pulse_ontime* are used to parametrize the RF pulse generation. Users can update them before execution using the *RF_pulse_parameters* class. 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 updated using the *AWG_parameters class*. AWG latency information are documented in the M3xxxA AWG documentation and in the SD1 documentation.

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 API class *SystemDefintion* allows to define all necessary HVI resources. The definition of HVI resources is the first step of an application using HVI. 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 application 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
my_system = 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
    """    # Load configuration
    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 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 digitiizer.

Python

```
class HVI_engine_Names:
    # Defines the Names of HVI engine used in this programming example
    def __init__(self):
        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

```
class HVI_resource_Names:
    # Defines the HVI action Names to be used by each HVI engine
    def __init__(self):
        # 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):
    """ 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 """    # Load configuration
    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
            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):
    " Defines the FP trigger to be used as a wait condition by the digitizer "    # Load
configuration
    config = ApplicationConfig()
    # 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, users can 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

```
def program_mixed_sig_meas_sequence(sequencer, module_dict):
    """    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.
    """    # Load configuration
    config = ApplicationConfig()
    # Define registers within the scope of the outmost sync sequence
    define_registers(sequencer, module_dict)
       #
    # 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)
    # Add 3rd Sync Multi-Sequence Block
    sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("Execution
Completed", 230)
    program_execution_completed(sync_block)
```

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. Register 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

```
class HVI_register_Names:
    # Defines the HVI registers (and their Names) to be used within the scope of each HVI
engine
    def __init__(self):
        self.iteration_counter = 'Iteration Counter'
        self.pulse_counter = 'Pulse Counter'
        self.awg1_counter = 'AWG1 Counter'
        self.queue_reg = 'Queue Reg'
        self.reg_wfm_A = 'Wfm A'
        self.reg_wfm_B = 'Wfm B'
        self.counter_reg = 'Counter Reg'
        self.hvi_done = 'HVI Done'



def define_registers(sequencer, module_dict):
    # Defines all registers for each HVI engine in the scope af the global sync sequence
    # Load previously defined register Names
    eng_Names = HVI_engine_Names()
    register_Names = HVI_register_Names()
    awg_params = AWG_parameters()
    # Digitizer registers
    iteration_counter = sequencer.sync_sequence.scopes[eng_Names.dig_engine].registers.add
(register_Names.iteration_counter, kthvi.RegisterSize.SHORT)
    iteration_counter.initial_value = 0
    hvi_done = sequencer.sync_sequence.scopes[eng_Names.dig_engine].registers.add(register_
Names.hvi_done, kthvi.RegisterSize.SHORT)
    hvi_done.initial_value = 0
    # RF Gen registers
    pulse_counter = sequencer.sync_sequence.scopes[eng_Names.rf_gen_engine].registers.add
(register_Names.pulse_counter, kthvi.RegisterSize.SHORT)
    pulse_counter.initial_value = 0
    # AWG 1:N Registers
    for engine_Name in module_dict.keys():
        if engine_Name!=eng_Names.rf_gen_engine and engine_Name!=eng_Names.dig_engine:
            queue_reg = sequencer.sync_sequence.scopes[engine_Name].registers.add(register_
Names.queue_reg, kthvi.RegisterSize.SHORT)
            queue_reg.initial_value = 0
            reg_wfm_A = sequencer.sync_sequence.scopes[engine_Name].registers.add(register_
Names.reg_wfm_A, kthvi.RegisterSize.SHORT)
            reg_wfm_A.initial_value = awg_params.wfm_A
```

```
            reg_wfm_B = sequencer.sync_sequence.scopes[engine_Name].registers.add(register_
Names.reg_wfm_B, kthvi.RegisterSize.SHORT)
            reg_wfm_B.initial_value = awg_params.wfm_B
```

Synchronized While

It corresponds to statement (a) in the HVI diagram. Synchronized While (Sync While) statements belongs 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)
```

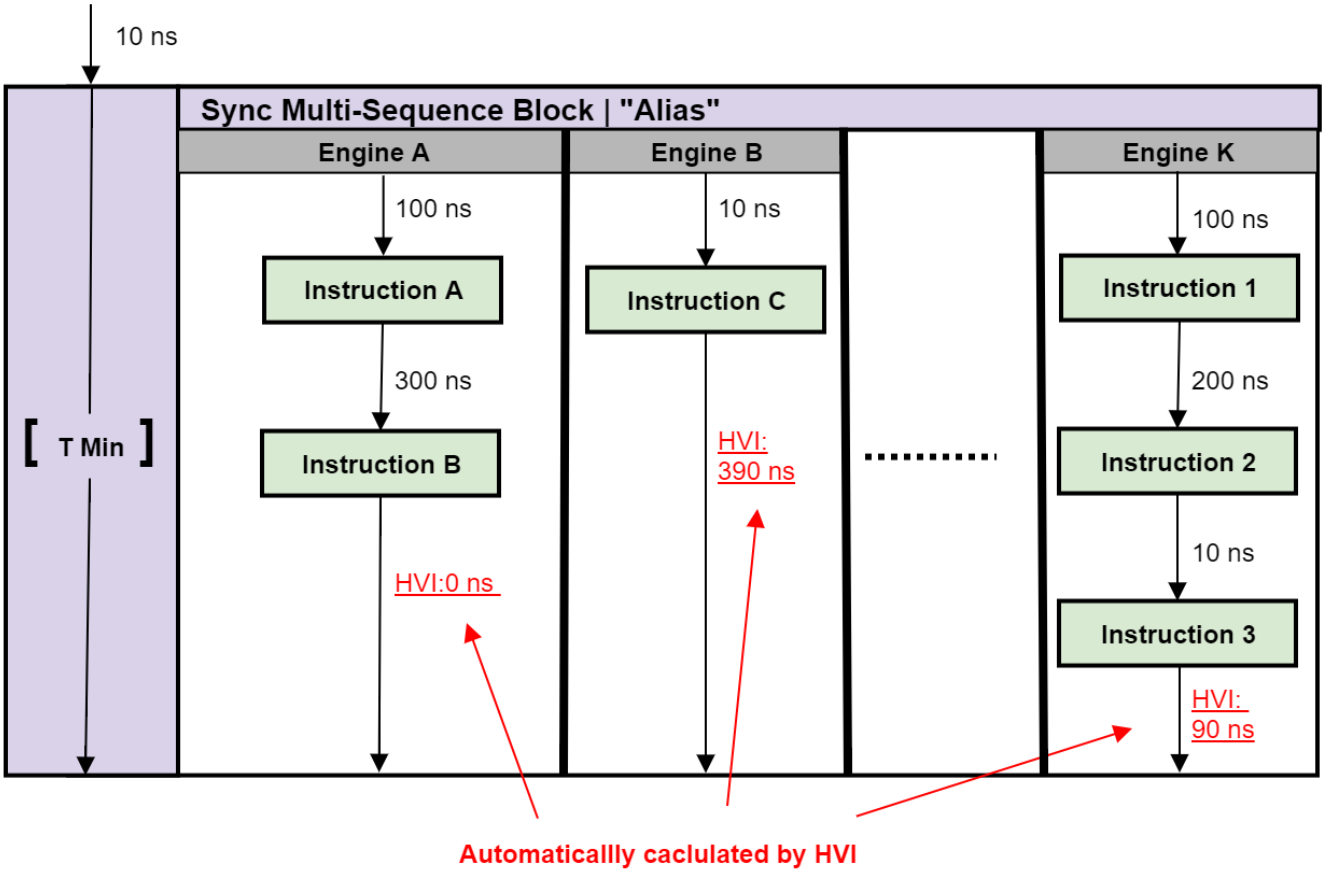
Synchronized Multi-Sequence Block

It 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. 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 KS2201APathWave Test Sync Executive User Manual available on www.keysight.com

Wait Statement

It 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 ca 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

It 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

It 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

It 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

```
# List of previously defined DAQ trigger actions
daq_trigger_all = []
for ch_index in range(1, module_dict[hvi_eng_Names.dig_engine].num_channels+1):
    daq_trigger_all.append(sys_def.engines[hvi_eng_Names.dig_engine].actions[ch_index-1])
```

```
# Digitizer sequence: DAQ trigger
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

It 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

```
# Sequence of RF Gen.
# Local WHILE: amplitude ON, amplitude OFF
local_while_condition = kthvi.Condition.register_comparison(pulse_counter,
kthvi.ComparisonOperator.LESS_THAN, rf_pulse_params.num_pulses)
while_loop = rfgen_sequence.add_while('Generate RF pulses', 60, local_while_condition)
while_sequence = while_loop.sequence
#Increment pulse_counter
instruction = while_sequence.add_instruction('Increment Pulse Counter', 10, while_
sequence.instruction_set.add.id)
instruction.set_parameter(while_sequence.instruction_set.add.destination.id, pulse_counter)
instruction.set_parameter(while_sequence.instruction_set.add.left_operand.id, pulse_
counter)
instruction.set_parameter(while_sequence.instruction_set.add.right_operand.id, 1)
```

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 M3xxxA instruments can be found in the M3xxxA 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

It 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(...)
...
```

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 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 2020\api\python\Help\index.htm.

Multi-Chassis Setup Implementation

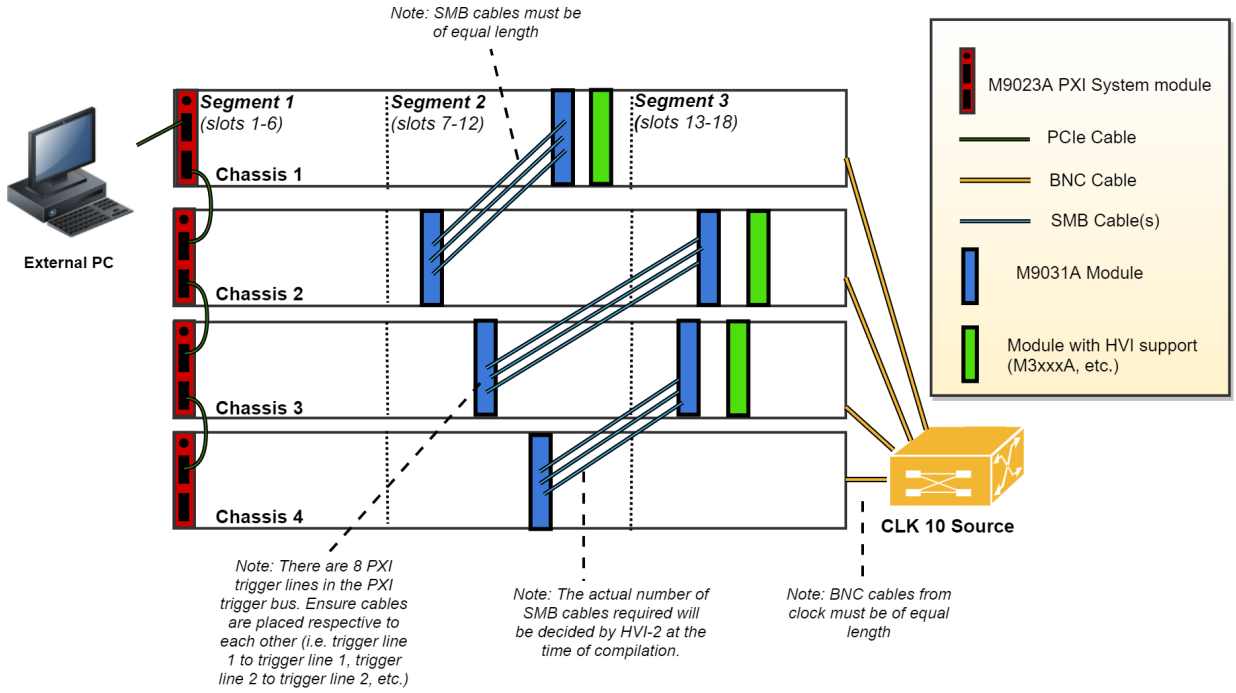
The reference examples provided with this document can be executed on a multiple-chassis setup with only the few modifications explained below. In a multi-chassis setup, it is necessary to interconnect the PXI triggers and clocking of the multiple chassis.

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:

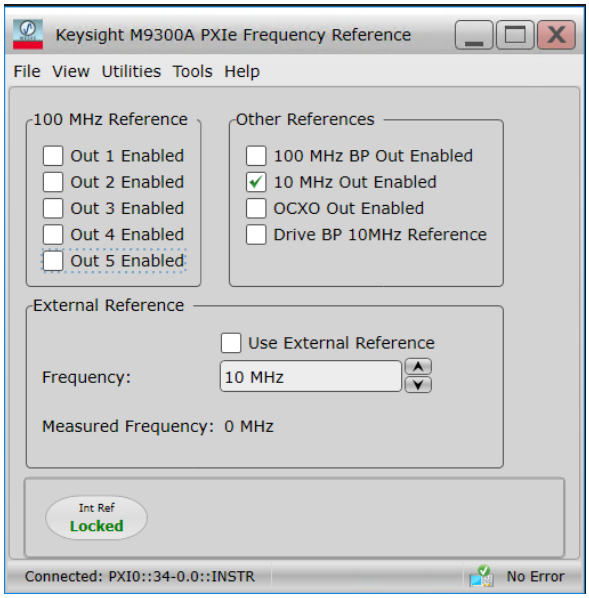
Python

```
# 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)
```

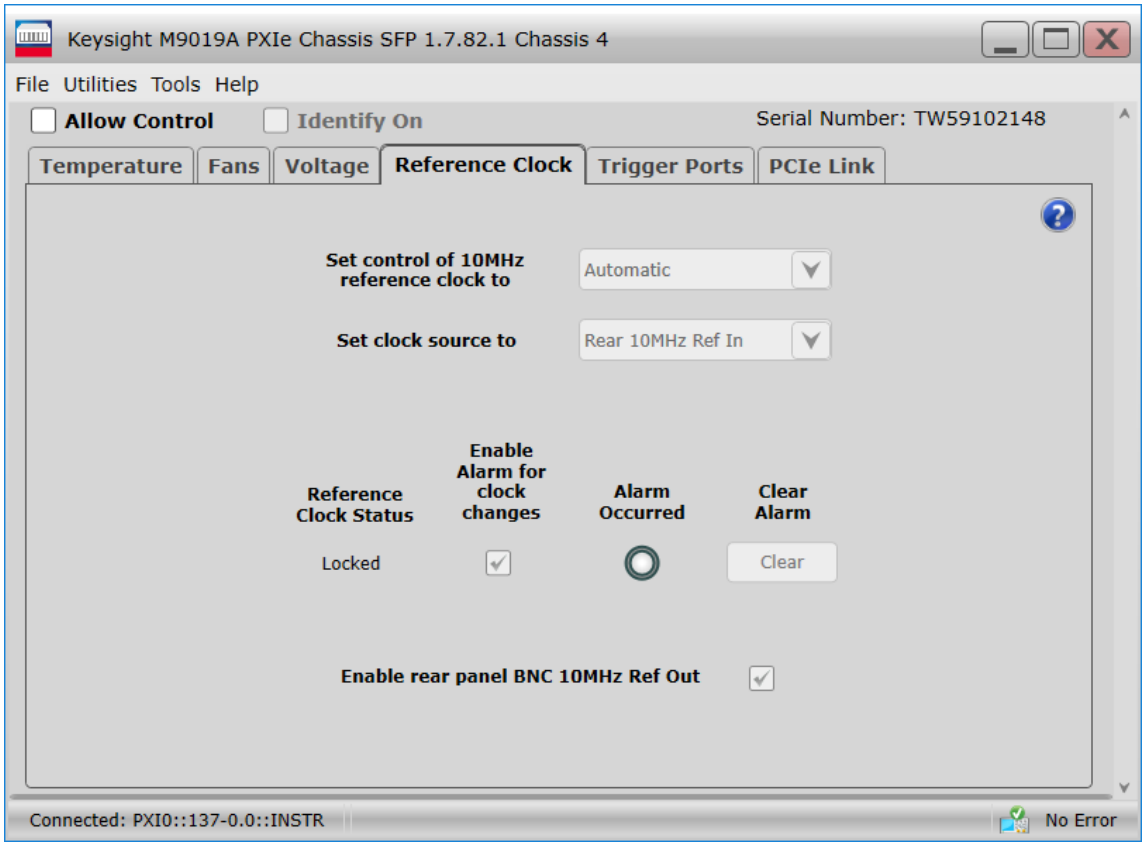
The above-mentioned code lines are part of this application code example and they can be used to adapt the code example to run on a multi-chassis setup.

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 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.