

Real-Time Pulsed Characterization of a Device-Under-Test

PATHWAVE

In this programming example an M3202A AWG and an M3102A digitizer are used to perform a real-time pulsed characterization experiment on a Device-Under-Test (DUT). This example can be used for power amplifier characterization for 5G mobile communications and quantum bit characterization experiments for quantum applications, in which case the AWG generates the control and readout pulses necessary for characterization of quantum bits.

PATHWAVE

Test Sync Executive

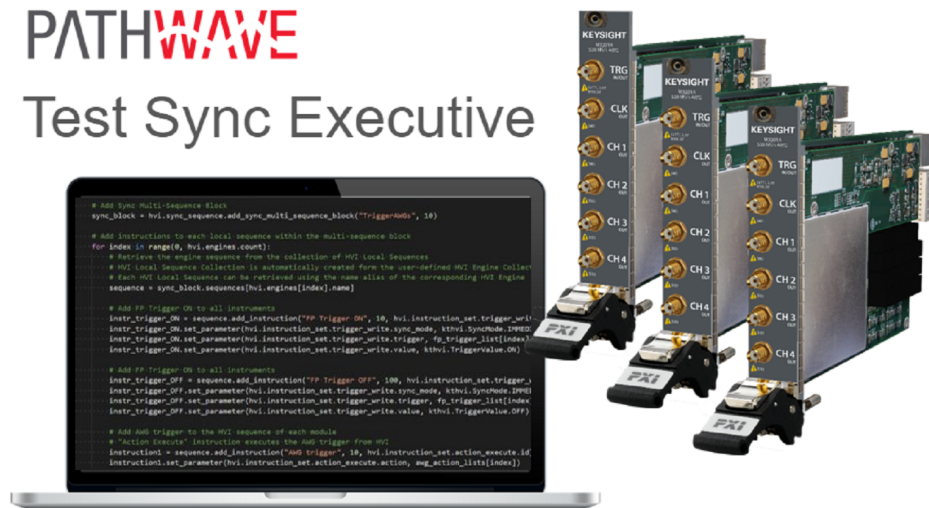


Table of Contents

KS2201A - Programming Example 4 - Real-Time Pulsed Characterization of a Device-Under-Test	4
Introduction	4
System Setup	4
System Requirements	4
How to Install Python 3.7.x 64-bit	5
How to Install Chassis Driver, SFP and Firmware	6
How to Install PathWave Test Sync Executive, SD1 SFP and M3xxxA FPGA Firmware	7
How to Install PathWave FPGA	8
Multi-Chassis Setup Implementation	8
10 MHz Clock Reference Source	10
Programming Example Overview	12
How to run this programming example	15
Measurement Results	17
HVI Application Programming Interface (API): Detailed Explanations	28
System Definition	33
Define Platform Resources: Chassis, PXI triggers, Synchronization	34
Define HVI Engines	34
Define HVI Actions, Events, Triggers	35
Program HVI Sequence	35
Define HVI Registers	36
Synchronized While	37
Synchronized Multi-Sequence Block	38
HVI Native Instruction: Register Assign	39
Sync Register Sharing	40
IF-ELSEIF-ELSE Statement	40
HVI Instrument-Specific Instruction: Queue AWG Waveform	41
Action Execute: AWG trigger, DAQ trigger	42
Wait Time	42
Register Increment	42
Delay Statement	43
Export the Programmed HVI Sequences to File	43
Compile, Load, Execute the HVI	43
Compile HVI	43
Load HVI to Hardware	44

Execute	44
Release Hardware	44
Further HVI API Explanations	45
Conclusions	46

KS2201A - Programming Example 4 - Real-Time Pulsed Characterization of a Device-Under-Test

In this programming example, an M3202A AWG and an M3102A digitizer are used to perform a real-time pulsed characterization experiment on a Device-Under-Test (DUT). A pool of different waveforms is loaded to the AWG RAM. The digitizer can use the register sharing functionality to select real-time the waveform to be played by the AWG at each iteration of the experiment steps. The selected waveform is used by AWG CH1 and CH2 to play I-Q modulated pulses and re-play them after a Variable delay. In the same iteration, AWG CH3 and CH4 play a **second burst of I-Q pulses** after another Variable delay. The **second burst** pulse length can be increased after each iteration. The experiment can be repeated for a user-defined number of loops, allowing the user to choose the delay between each loop, delay necessary for example to let the DUT return to its equilibrium state. Example use cases for this programming example include power amplifier characterization for 5G mobile communications and quantum bit characterization experiments for quantum applications, in which case the AWG generates the control and readout pulses necessary for characterization of quantum bits.

Introduction

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

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

System Setup

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

System Requirements

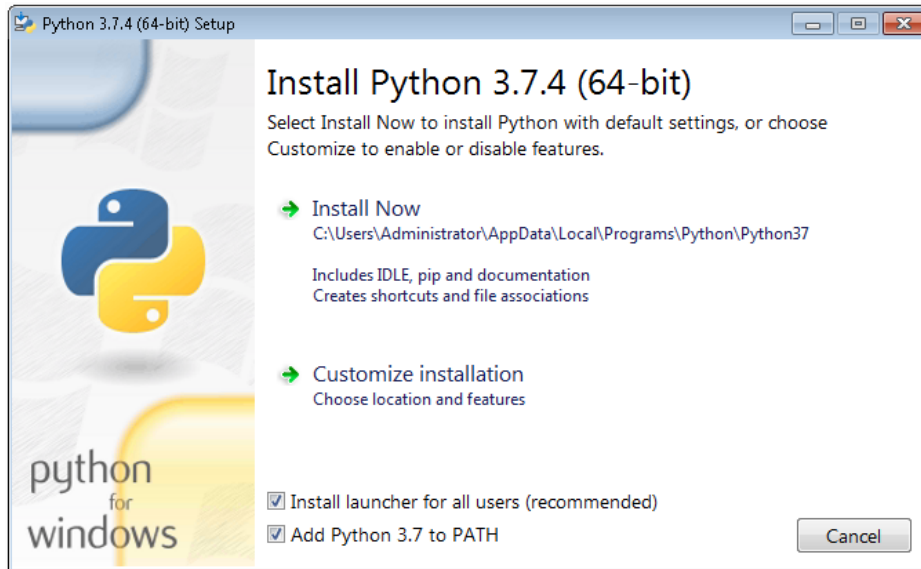
The versions of software, FPGA firmware, drivers, and other components that are required to run this programming example are listed below. All pieces of SW and firmware listed below need to be installed on the external PC or internal chassis controller that is used to control the PXI chassis. FPGA FW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

1. Software versions required:
 - Python 3.7.x 64-bit, including Python packages time, numpy, matplotlib
 - Keysight IO Libraries Suite 2020 (v18.1.25310.1 or later)
 - Keysight SD1 Drivers, Libraries and SFP (v3.1.9 or later)
 - Keysight PathWave Test Sync Executive Update 1 (v1.4.3 or later)
2. Chassis firmware and driver:
 - Keysight Chassis M9019A firmware (v2019EnhTrig or later)
 - Keysight PXIe Chassis Family Driver (v1.7.402.1 or later)
3. M3xxxA with -HVx HW option and following FPGA firmware versions (to be installed using Keysight SD1 SFP):
 - M3202A AWG FPGA firmware (v4.1.20 or later)
 - M3201A AWG FPGA firmware (v4.2.85 or later)
 - M3102A Digitizer FPGA firmware (v2.01.60 or later)

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

How to Install Python 3.7.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 official webpage <https://www.python.org>. 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.

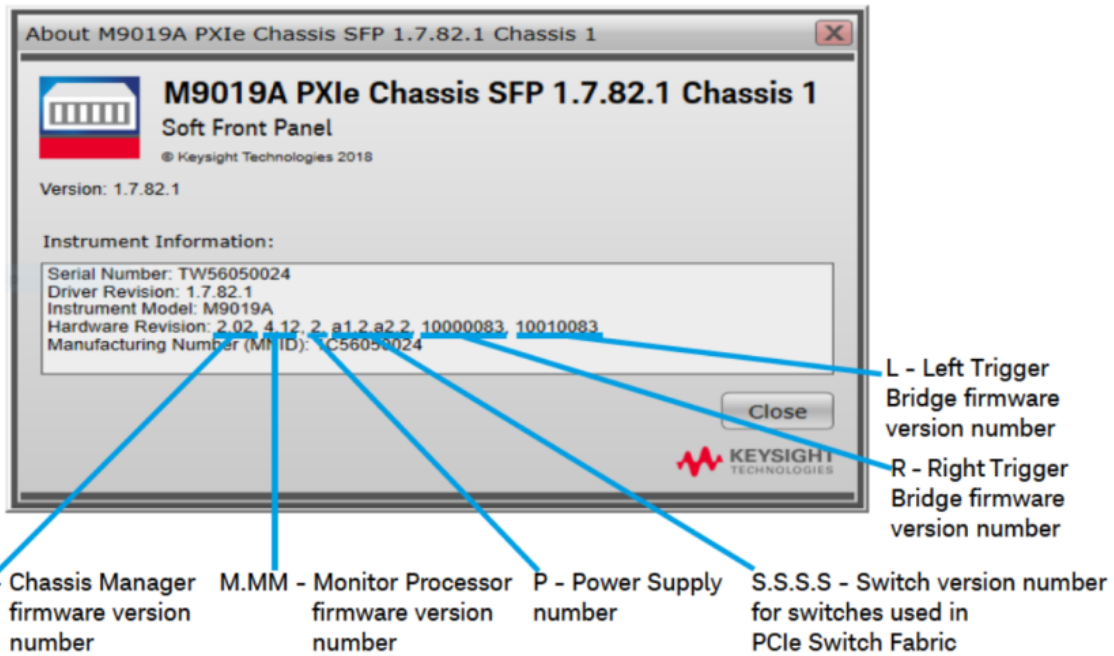


NOTE PathWave Test Sync Executive programming examples require the Python packages ***time***, ***numpy*** and ***matplotlib***. These packages can be installed using the Python package installer pip. For more information about pip and how to use it, please visit <https://pypi.org/project/pip/>.

NOTE Users installing Python through a distribution that is different than the one available from the Python official webpage <https://www.python.org> (e.g. Anaconda distribution) need to make sure that their PATH environment Variable includes the path to setup the HVI API Python library. This can be done by adding to the programming example Python code a line that include that path, for example: `sys.path.append(C:\Program Files\Keysight\PathWave Test Sync Executive 2020\api\python)`

How to Install Chassis Driver, SFP and Firmware

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



M9019A Firmware Version Components

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

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

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

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

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

How to Install PathWave FPGA

The project files of the 3rd programming example titled "PathWave Test Sync Executive Integration with PathWave FPGA" include PathWave FPGA projects designed using **Keysight PathWave FPGA 2020 Update 1.0**. To install and obtain a license for Keysight PathWave FPGA 2020 Update 1.0 (or a later version) please consult the product webpage on www.keysight.com. PathWave FPGA also require Xilinx Vivado software to run. For further information please consult the PathWave FPGA User Manual on www.keysight.com.

Multi-Chassis Setup Implementation

This section explains how to execute the reference examples provided with this document on a multiple-chassis setup. In a multi-chassis setup, it is necessary to interconnect the PXI triggers and clocking of the multiple chassis.

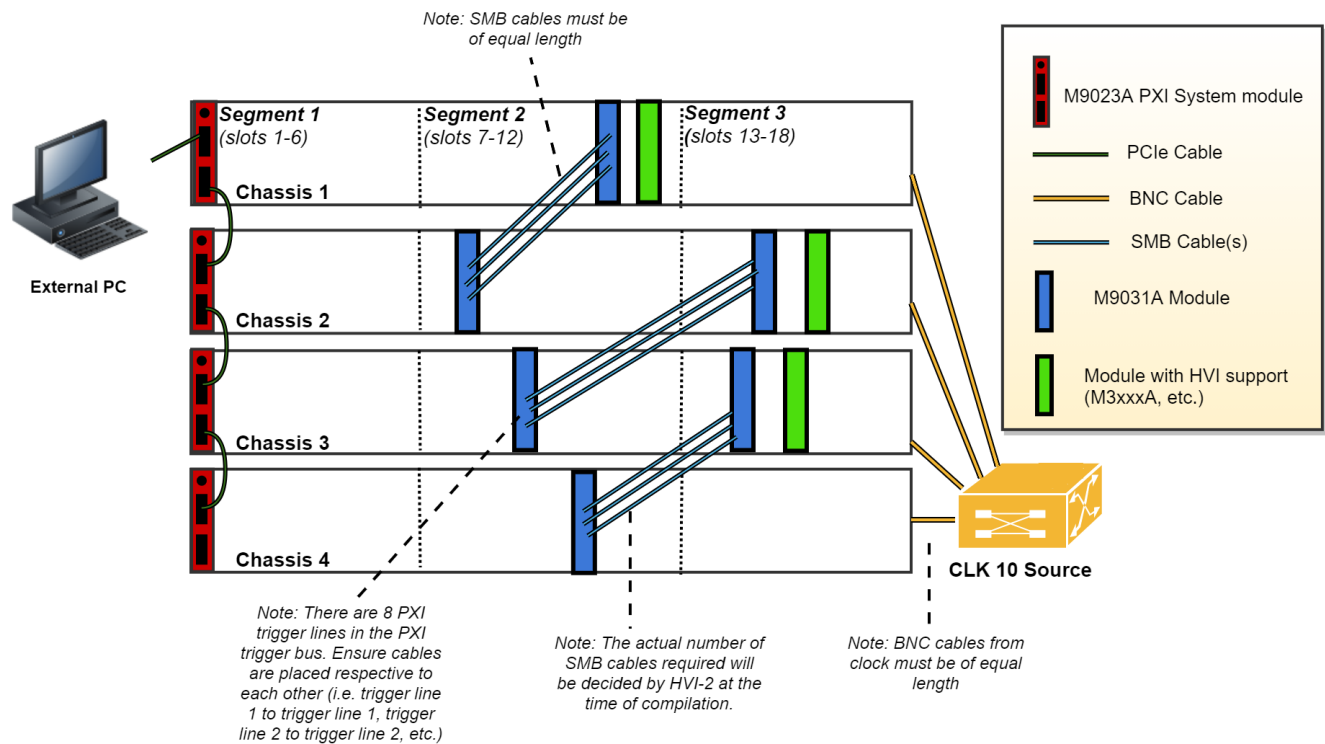
With the currently available infrastructure to interconnect PXIe backplane triggers a pair of M9031A boards must be placed in a specific segment in each chassis to be interconnected.

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:

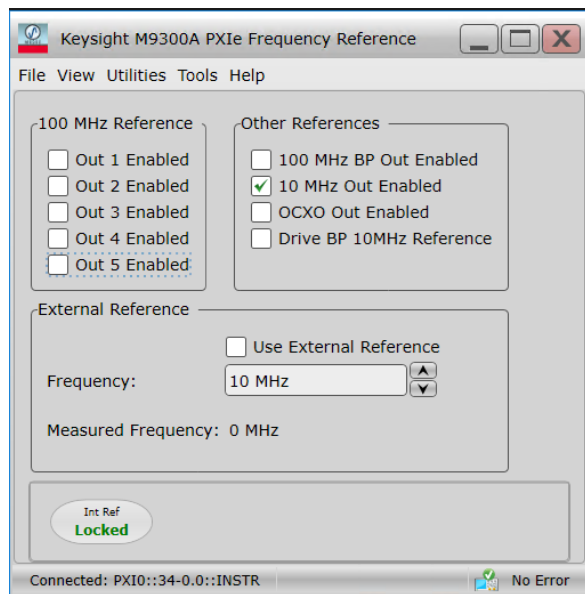
- The 1st chassis must include a M9031A together with a PXI module with HVI in segment 2
- All Middle chassis must have a M9031A in the segment 2, and a M9031A together with a PXI Module with HVI support in Segment 3
- The last chassis must include a M9031A in segment 2.

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 $2*(N-1)$.

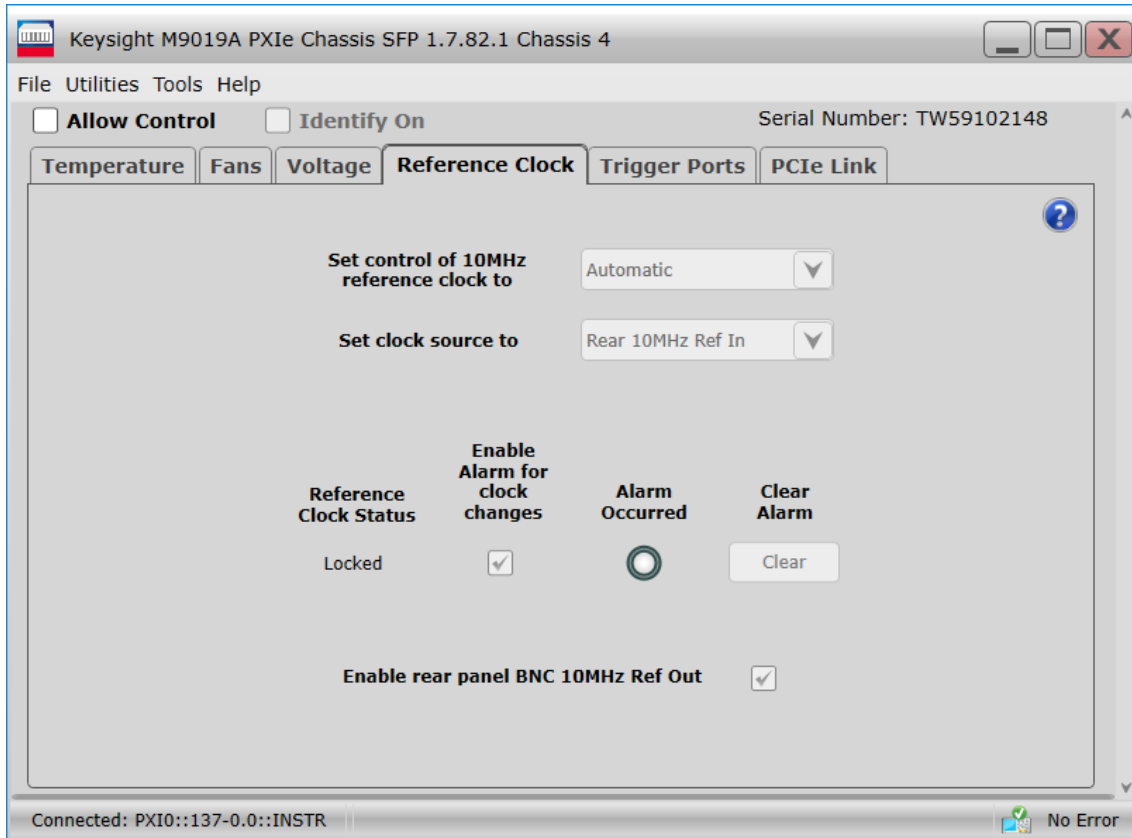
Additionally, a very clean 10 MHz source should be used to provide the same reference signal to all chassis. One option is to use a multi-output 10 MHz source, for best performance probably driven by an atomic clock, connecting each output to the 10 MHz reference input of each chassis using cables that 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.

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".



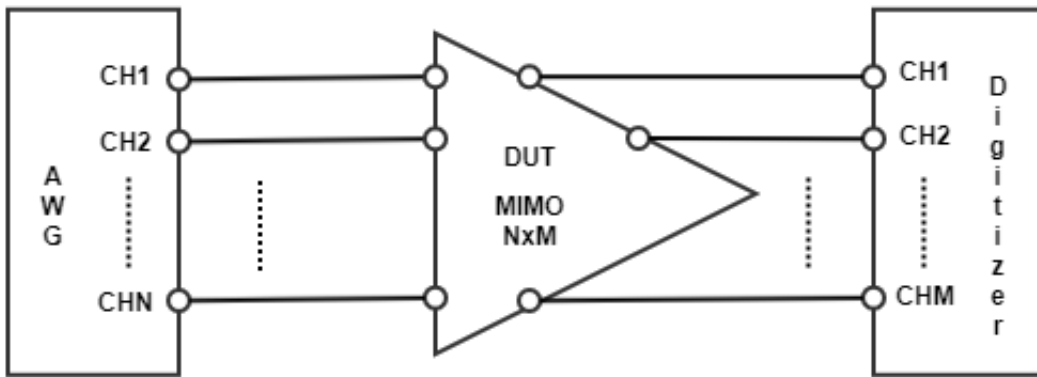
Additionally, in the case of using a remote controller card, like the M9023A PXI System Module used in this application, it is possible to see the backplane status LEDs that also indicate the correct clocking. On the chassis backplane REF and LOCK LED lights are lighted in green when the chassis is correctly 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.



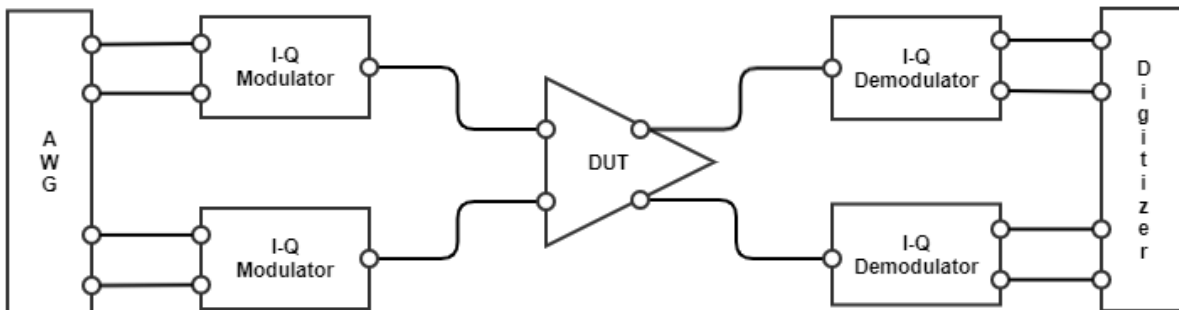
For more details on the Keysight PXIe Chassis Family please visit www.keysight.com.

Programming Example Overview

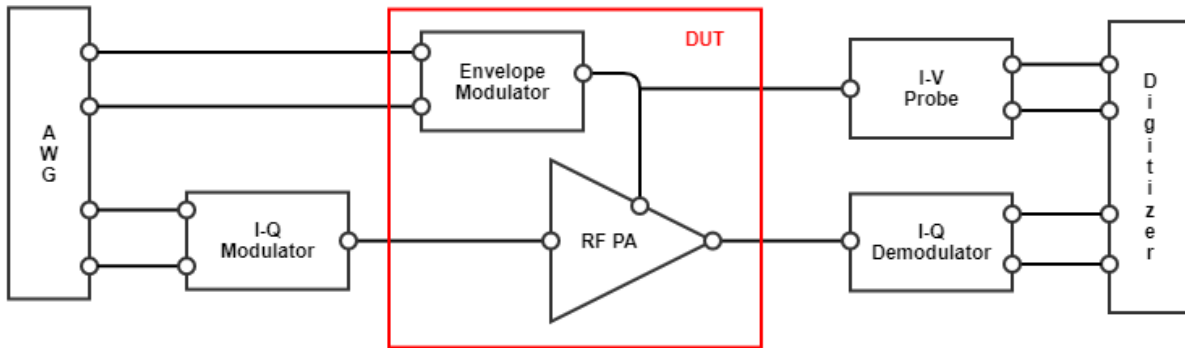
The DUT characterization experiment implemented in this programming example is represented in the setup diagram below.



In the general case, this programming example can be deployed on Multiple-Input Multiple-Output (MIMO) Device-Under-Tests (DUTs). The number of inputs and outputs depends on the DUT. To deploy this programming example on an $N \times M$ MIMO DUT, it is necessary to use an AWG with N channels and a digitizer with M channels. The example application and measurement results carried out in the rest of the document are obtained using an AWG M3202A and a digitizer M3102A having four channels each. Hence, the specific use case addressed by this document applies to DUTs up to MIMO 4×4 , or MIMO 2×2 in case the AWG and digitizer respectively generate and measure I-Q (In-phase and Quadrature) signals that need to pass through frequency converters (i.d. I-Q modulators/demodulators) before they can be applied to the DUT. This latter use case is depicted in the figure below.

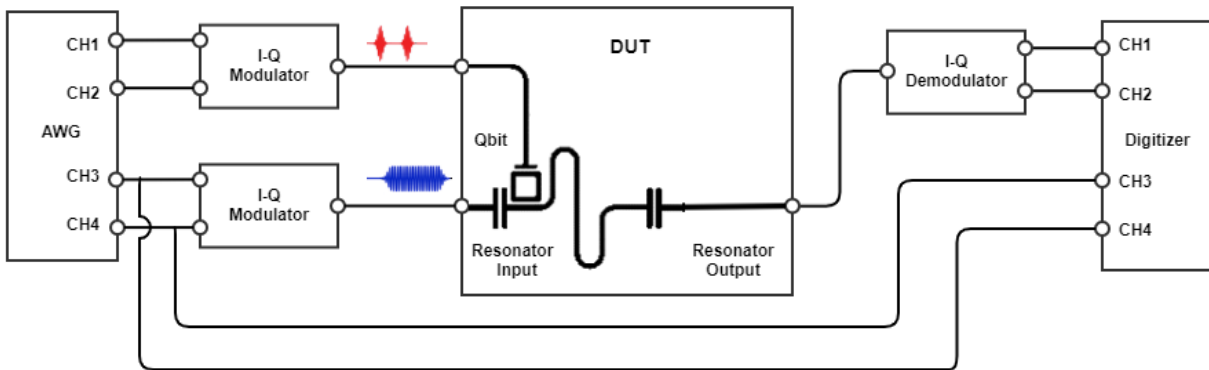


As an example, Radio Frequency (RF) Power Amplifiers (PAs) used in mobile communications are typically Single-Input Single-Output (SISO) systems, but the latest advanced transmitter configuration for the 5th Generation (5G) of mobile communications can include multiple amplifiers configured together to form an Active Phased Array (APA) containing multiple PAs. High-efficiency transmitter architectures including the Envelope Tracking (ET) configuration can also be addressed by this programming example, as represented in the following figure.



In particular, to address the ET PA characterization use case, users might prefer to substitute the example pulsed waveforms used in this programming example with real telecommunication waveform data samples. The usage of I-Q modulators/demodulators, I-V probe is not covered in this programming example. This programming example does not cover either the application of calibration techniques aiming at reconstructing the true waveforms at the DUT reference planes. This is left to the user as a possible add-on.

Another interesting use case is the characterization of quantum bits (Qbits) for quantum applications. Such applications can be covered by this programming example using a setup similar to the one represented in the figure below.



The arbitrary waveforms loaded to the AWG RAM in this programming example include the π and $\pi/2$ gaussian pulses typically used as Qbit excitation signals. The measurement results included in this document show I-Q pulses output from the AWG channels to produce the typical saturation and readout pulses to be sent to a superconductive Qbit and its resonator to perform the Qbit coherence time T_1 (also known as energy relaxation time) and the Qbit dephasing time T_2 .

How to run this programming example

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

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

Afterward, it is necessary to specify the actual chassis number and slot number where the real PXI instruments are located. The model number of the used PXI instruments 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 deploys one AWG and one digitizer, therefore two instances of the `module_descriptor` are used. Please update the properties in each `module-descriptor` object before running the programming example:

```
# Define module descriptors below with your instruments information
self.digitizer_descriptor = ModuleDescriptor('M3102A', 1, 9, self.options, self.dig_engine_
Name)
self.awg_descriptor = ModuleDescriptor('M3202A', 1, 8, self.options, self.awg_engine_Name)
```

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

The chassis to be used in the programming example must also be specified, and listed by chassis number. In the case of a multi-chassis setup, please specify the connection between each pair of M9031 modules using the `M9031_descriptor` class.

```
# Define list of chassis numbers included in the programming example
self.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.
self.M9031_descriptors = [M9031Descriptor(1, 11, 2, 11)]
```

```
class M9031Descriptor:
    # Describes the interconnection between each pair of M9031 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 HVI 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 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,
    kthvi.TriggerResourceId.PXI_TRIGGER4,
    kthvi.TriggerResourceId.PXI_TRIGGER5,
    kthvi.TriggerResourceId.PXI_TRIGGER6,
    kthvi.TriggerResourceId.PXI_TRIGGER7]

```

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 shall include at least the necessary number of PXI lines for the application to execute. Since this programming example uses the [Sync Register Sharing](#) functionality, the number of reserved PXI lines for HVI needs to be greater than the number of bits shared between the registers that are used for the [Sync Register Sharing](#).

Users can set the AWG and digitizer parameters using the classes defined in the following code snippets:

```

"""AWG parameters
"""self.all_ch_mask = 0xF # binary mask defining which channels to use
# 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_SINUSOIDAL
self.start_delay = 0 # x10 [ns]
self.awg_prescaler = 0
self.wfm_cycles = 2 # number of pulsed wfms for the T2 experiment
self.amplitude = 1 # [V]
self.offset = 0 # [V]
# Trigger settings
self.awg_trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG_CYCLE
# Latency values for M3202A AWGQueueWfm() [ns]
# Latencies depend on AWG FPGA FW. Please check the SD1 3.x User Guide for detailed info
self.queue_wfm_latency = 100 # [ns] Minimum start delay necessary to execute an AWGQueueWfm()
instruction
self.awg_trigger_latency = 2300 # [ns] Minimum latency necessary between an AWGQueueWfm()
instruction and an AWGtrigger action.
# Readout pulse parameters

```



```

self.rorise_id = 1000 # wfm ID for the rising edge of the readout pulse
self.rofall_id = 1001 # wfm ID for the falling edge of the readout pulse

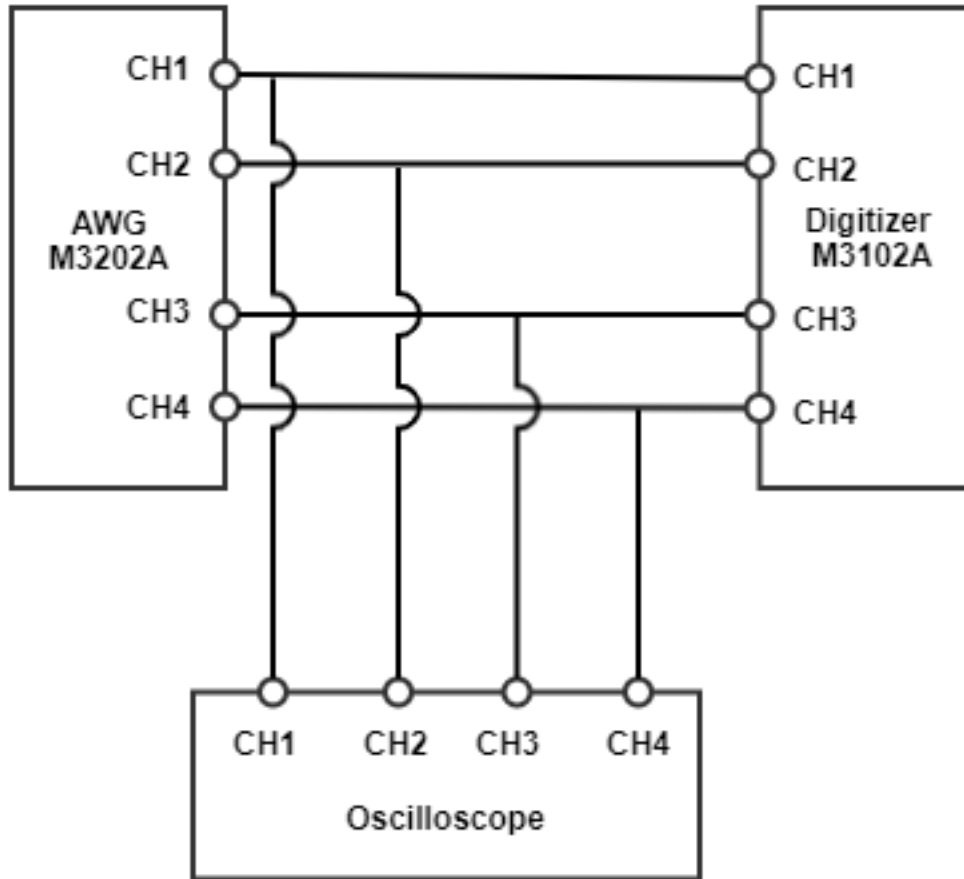
"""Digitizer parameters
"""self.sampling_time = 2 # [ns] 1/sample_rate, sample_rate = 500 MSa/s for Digitizer
M3102A
self.dig_prescaler = 0 # Prescaler values are explained in M3xxxA User Guide
self.fullscale = 2 # [V] enter x Volts to set the full scale to [-x, x] Volts
self.acquisition_points_per_cycle = int(self.acquisition_window / self.sampling_time) #
[Sa]
self.num_cycles = self.num_steps*self.num_loops # insert -1 for infinite cycles
self.acquisition_points = self.acquisition_points_per_cycle*self.num_cycles
self.acquisition_delay = 0 # x2[ns]
self.dig_trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG
self.dig_mask = self.all_ch_mask

```

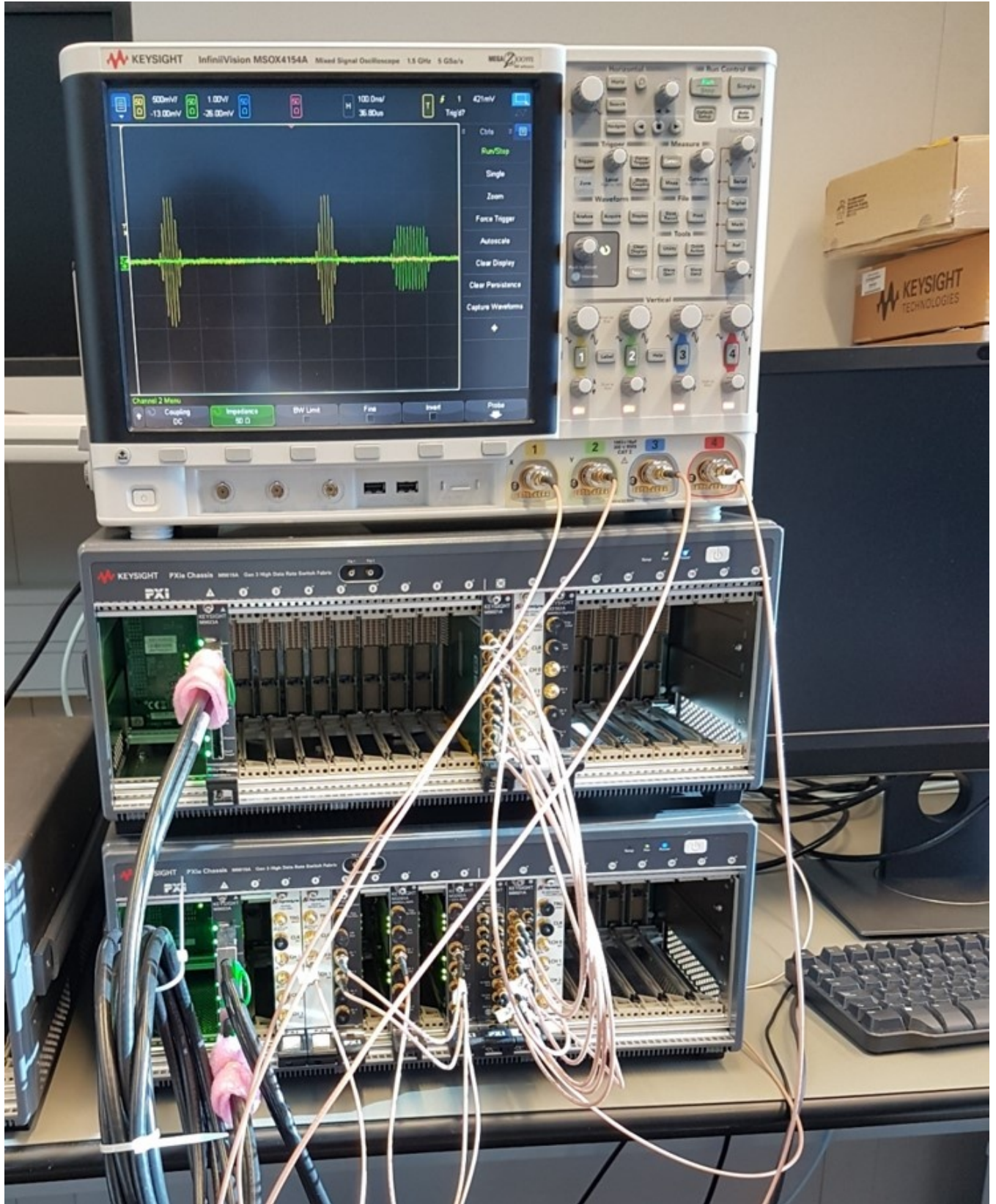
For details on the parameters defined for AWG and digitizer please refer to M3xxxA AWG and digitizer user guides available on www.keysight.com. Experiment parameters must also be set before running this programming example. Detailed information to set them are provided in the next section of this programming example.

Measurement Results

The programming example capabilities will be illustrated through some example measurement results obtained using the measurement setup depicted below where each of the four channels of the M3202A AWG is connected to the corresponding channel of the M3102A digitizer and to the corresponding channel of a Keysight oscilloscope, using a T-connector.



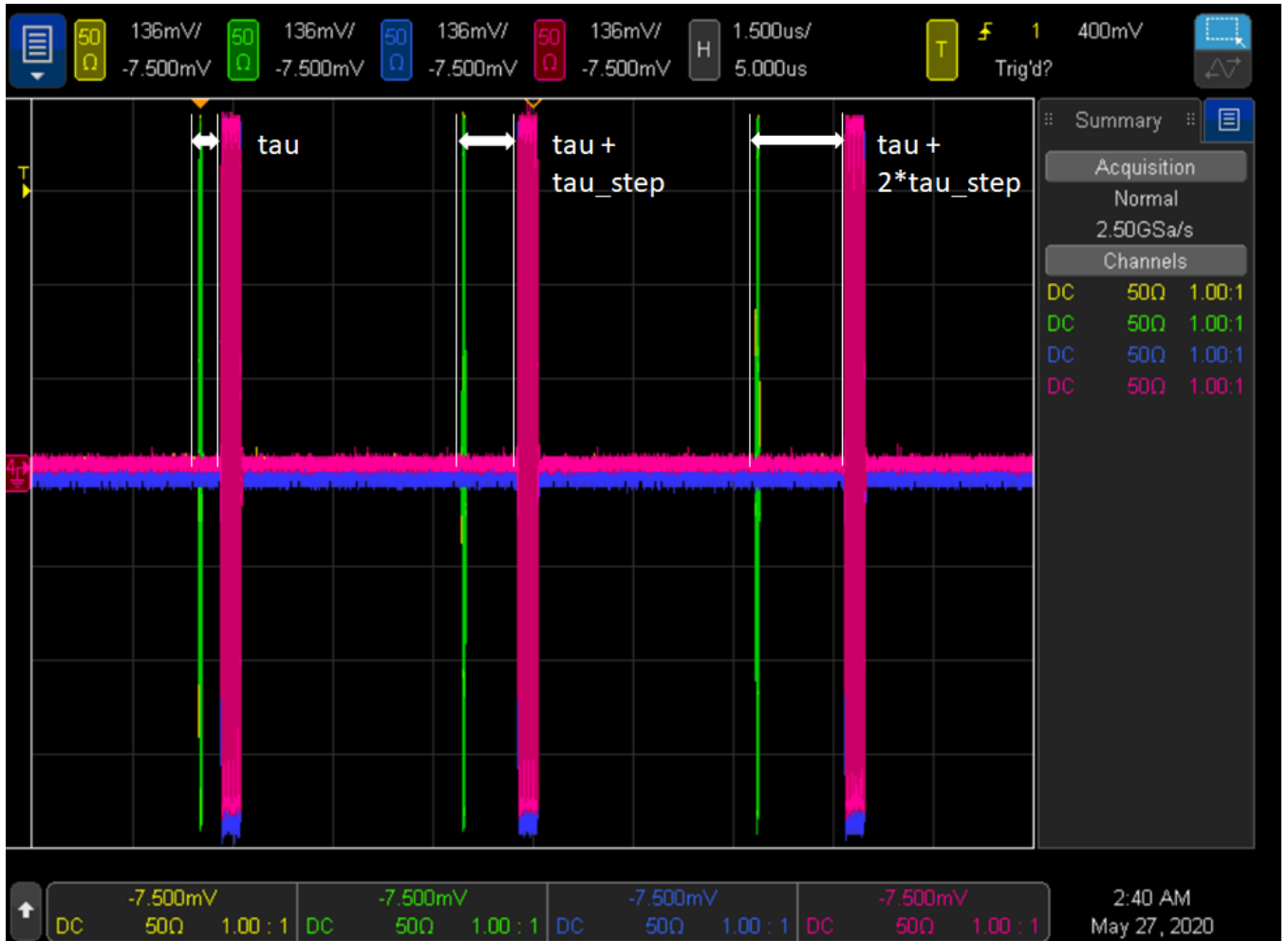
A photograph of the measurement setup used for the measurement results reported in this programming example is reported below:



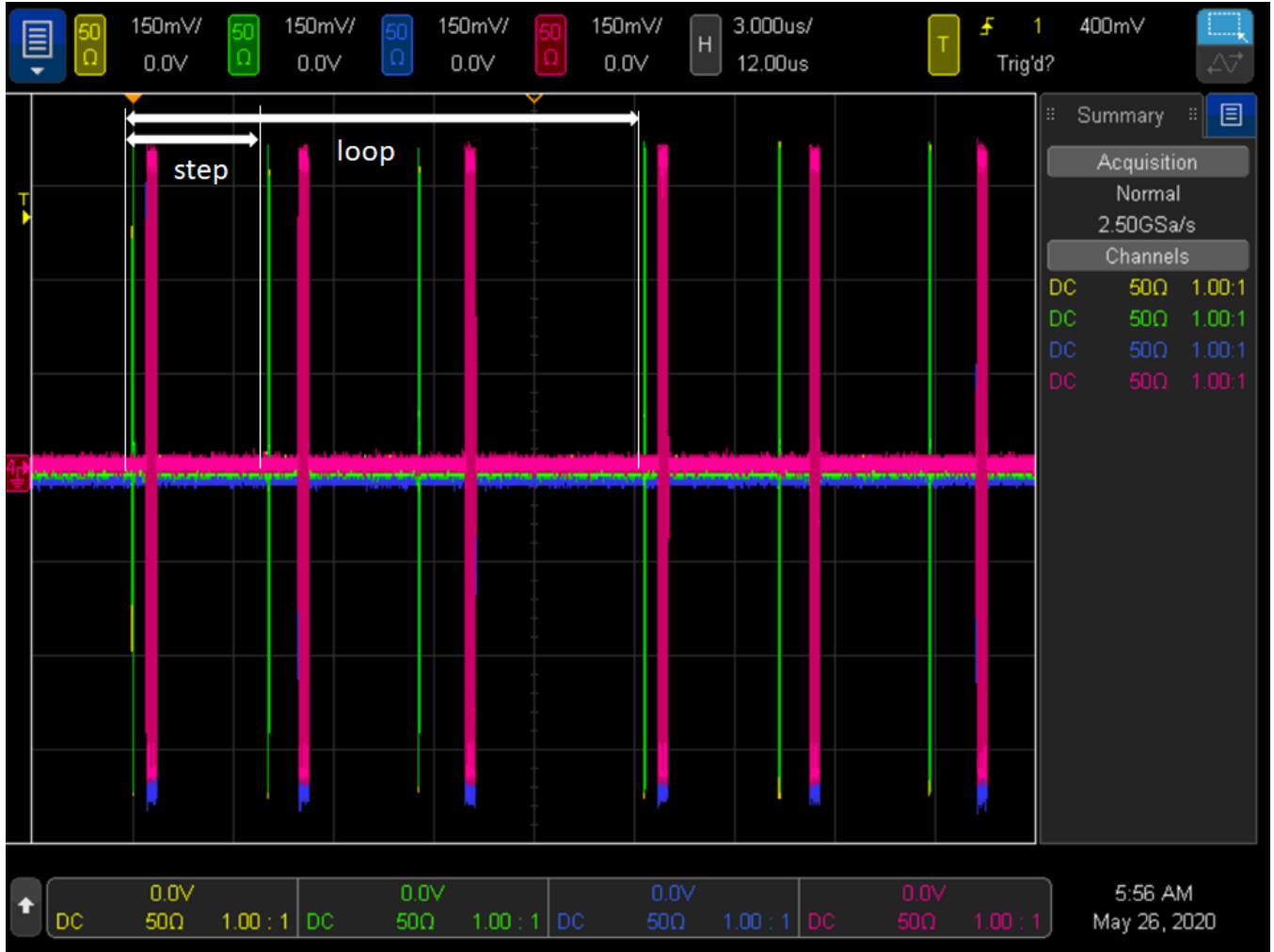
The first step to run this programming example is to define the experiment parameters. The example measurement results reported in the rest of this document are obtained with the experiment parameter values set as follows:

```
"""Defines the experiment parameters
"""
self.num_wfms = 1 # Number of waveforms to be loaded to the AWG RAM
self.T2_flag = 0 # User can choose to run a T1 or T2 experiment
self.initial_tau = 10 # x10[ns] # The initial time delay between the control and readout
pulse, in ns
self.tau_step = 20 # x10[ns] # Time that is incrementally added to delay between the
control and readout pulse, in ns
self.ro_delay = 150 # [ns] # Delay in ns that is applied after the last control pulse, but
before the readout pulse
self.step_delay = 0 # x10[ns] # Time to wait between each experiment step
self.loop_delay = 0 # x10[ns] # Time to wait between each experiment loop
self.initial_acq_delay = 250 # x10[ns] # Delay before starting to capture waveforms with
digitizer
self.acquisition_window = 2000 # [ns] time window to be acquired by DAQ channel each time a
DAQ trigger is sent out
self.carrier_frequency = 100e6 # [Hz] frequency of the IF carrier modulating the I-Q pulses
at the AWG output
self.initial_pulse_length = 30 # x10[ns] # Initial readout pulse length
self.delta_length = 20 # x10[ns] # Duration increment of the readout pulse length at each
step
self.num_steps = 5 # Number of iterations to increase tau by tau_step
self.num_loops = 2 # Number of experiments to execute
```

The experiment repeats for a number of iteration steps. At each step, parameters such as the delay tau between the saturation pulse and the readout pulse can be incremented by an incremental quantity defined as an experiment step (*tau_step* Python code Variable listed above). Each step iteration is repeated after a step delay that can be defined by the user to make sure the DUT responses at each experiment steps are uncorrelated. The oscilloscope measurement below displays how the tau delay increments over two experiment steps.



The experiment is then repeated for a number of experiment loops. Each experiment loop can start after a user-defined loop delay to allow the DUT to return to its equilibrium state before the next series of experiment steps can be performed. By increasing the number of experiment loops, the user can collect repeated DUT measurements that can enable you to calculate statistics on the experiment results. Experiment step and loop iterations are depicted in the oscilloscope measurement below representing an example experiment execution with three steps ($num_steps = 3$) and two loops ($num_loops = 2$).



An example of a successful execution is represented in the system console below:

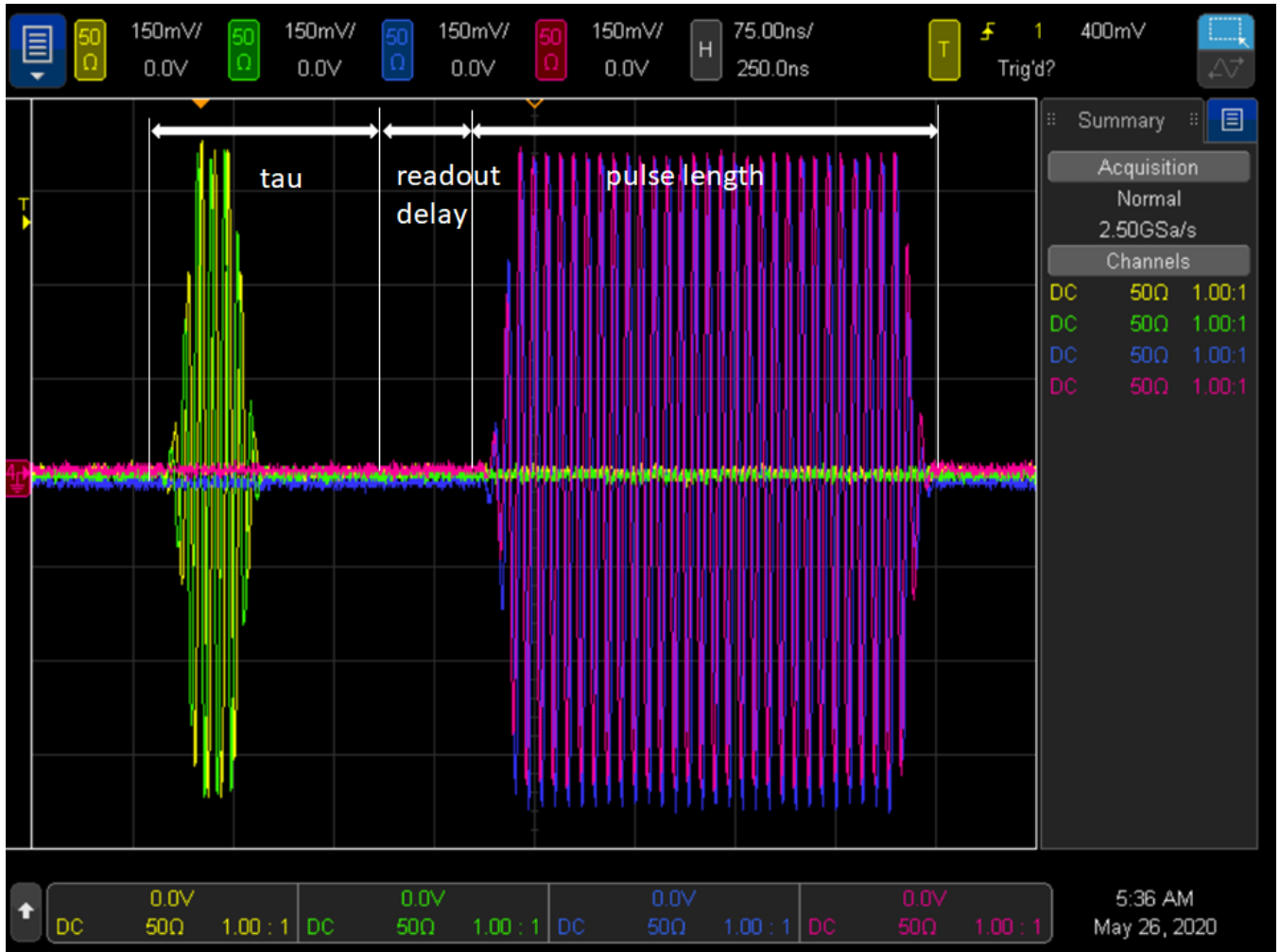
```
##### Experiment Parameters #####
Number of HVI loops: 2
Number of HVI steps: 5
Carrier frequency: 100 MHz
Number of different AWG waveforms: 1
Digitizer configured to acquire 1000 points in 10 cycles
Total dig. acquisition_points = points_per_cycle*num_cycles = 10000
Readout delay: 150 ns
Tau delay set to sweep from 100 ns to 900 ns in 4 steps of 200 ns
Readout pulse duration set to sweep from 300 ns to 1100 ns in 4 steps of 200 ns

Programming the HVI sequences...
Programmed HVI sequences exported to file
HVI Compiled
This HVI application needs to reserve 5 PXI trigger resources to execute
HVI Loaded to HW
HVI Running...

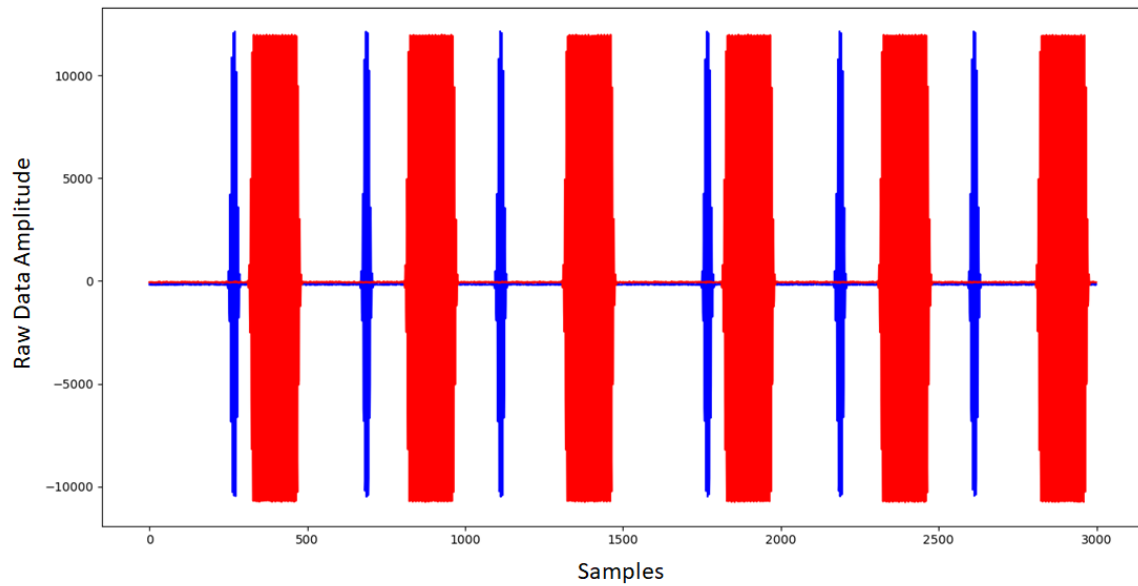
##### Experiment Results #####
Experiment steps executed: 5
Experiment loops executed: 2
Final awg_counter: 10
Final dig_counter: 10
Final wfm_id: 0
Final tau: 110
Final acq_delay: 350
Final pulse_length: 130

HVI Execution Completed Successfully!
Releasing HW...
PXI modules closed
```

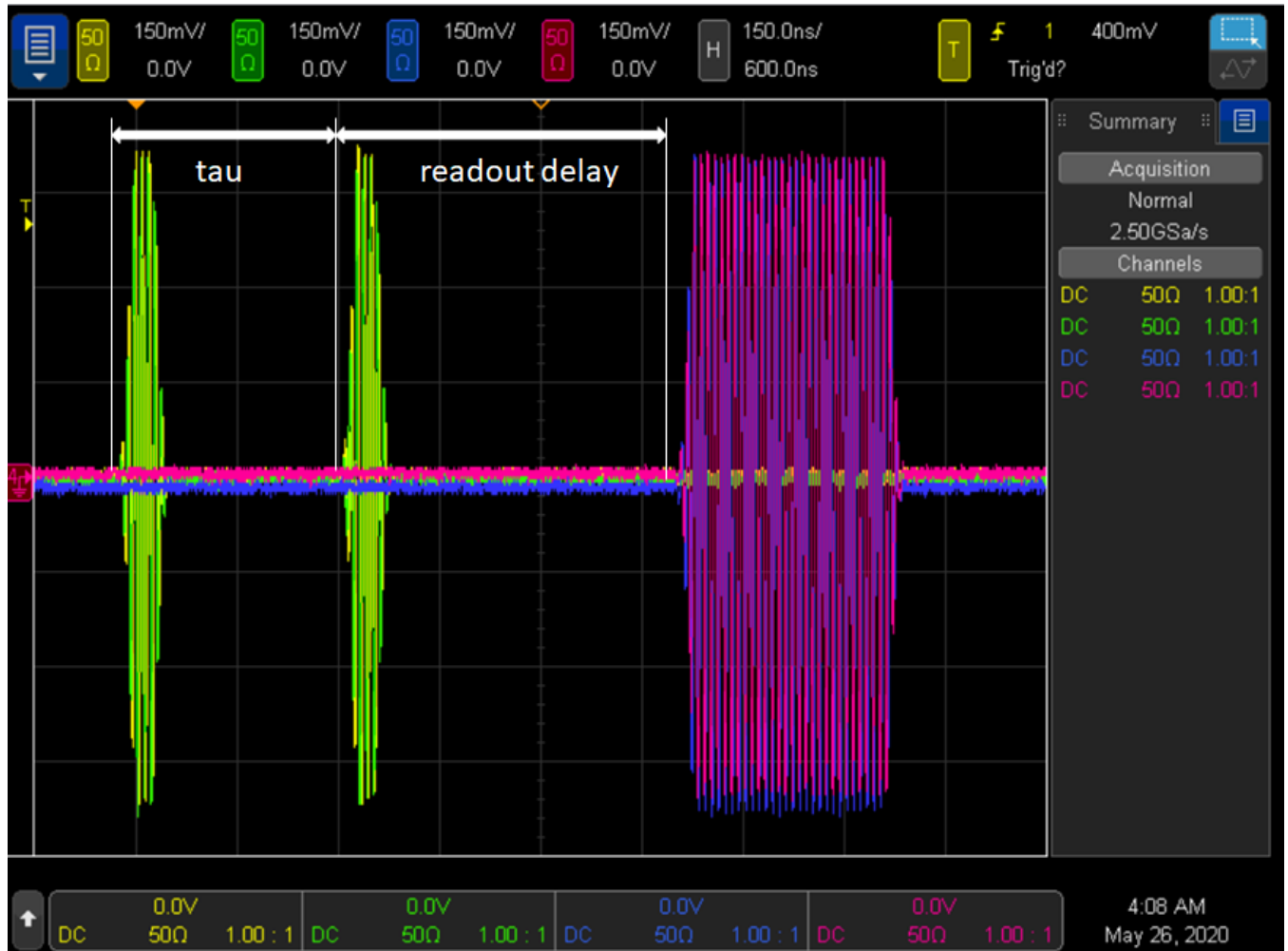
The oscilloscope measurement below represents the experiment parameters tau, readout delay and readout pulse length, all implemented using HVI registers. More details on the HVI resources and sequences programmed to implement the programming example functionalities are provided in the next section.



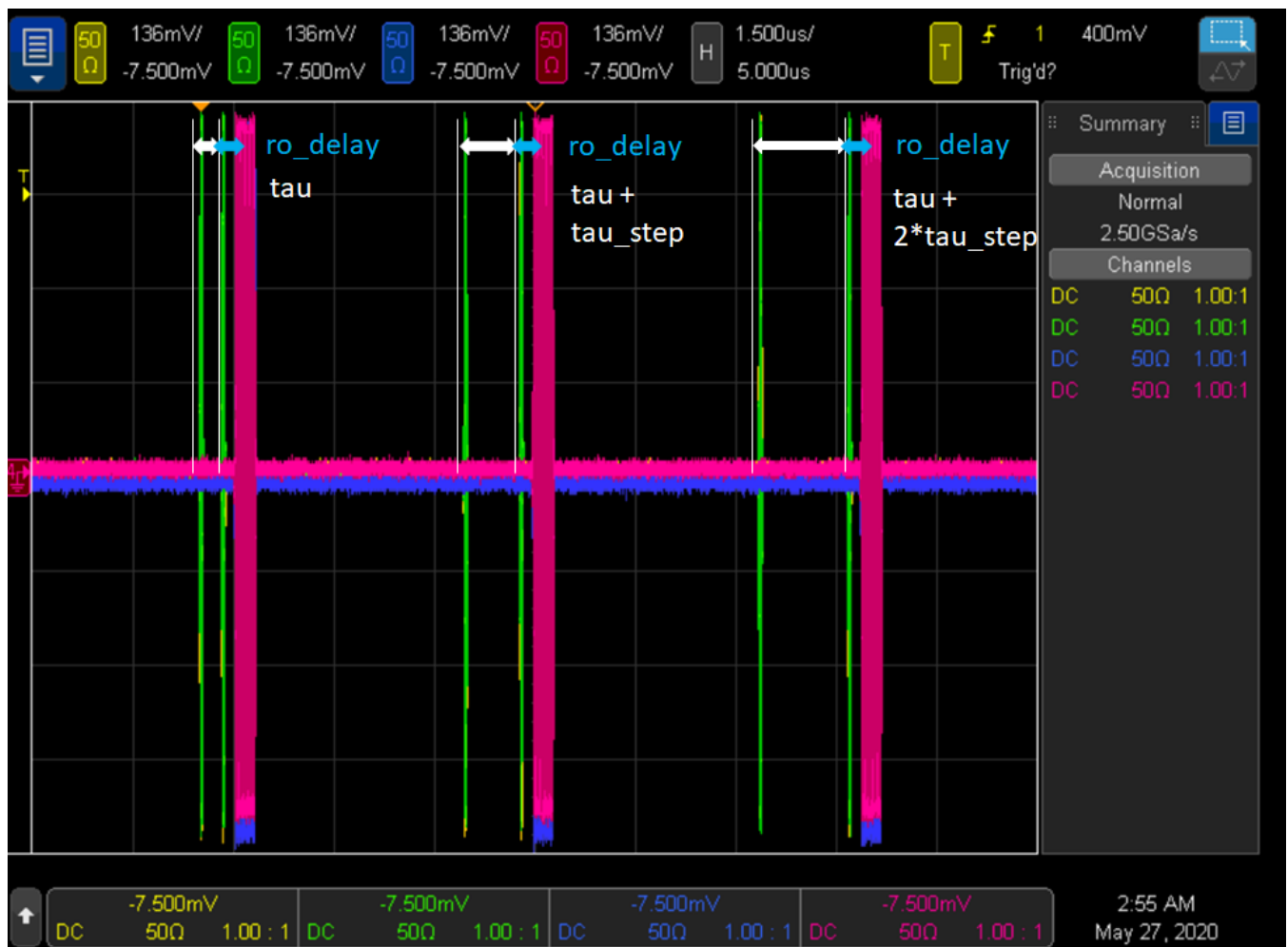
Thanks to the powerful synchronization capabilities of **PathWave Test Sync Executive** and HVI technology, each digitizer acquisition cycle can be precisely triggered synchronously with the time window of the waveform generated by the AWG. Users can adjust the starting point of the acquisition time window by setting the *initial_acq_delay* parameter. The figure below represents an example of a completed series of digitizer acquisition cycles corresponding to the same experiment steps and loops shown in the previous oscilloscope measurements. The red and blue waveform represented below correspond respectively to the raw measured data at DAQ channels CH1 and CH3, which are connected to the AWG channels generating the in-phase saturation pulse and readout pulse respectively.



Users can change the experiment parameters to achieve different types of DUT characterization. By setting the experiment parameter $T2_flag = 1$, the Python code execution generates at each experiment step two consecutive I-Q pulses output from AWG CH1 and CH2. The two pulses are separated by a delay τ that increments at each iterations step, whereas the readout delay with respect to the I-Q readout pulses output by the AWG CH3 and CH4 stays fixed.



The activation of the T2_flag parameters allows you to run this programming example to perform an experiment typically used for the characterization of the T2 time, i.e. dephasing time of quantum bits. This experiment is also known as Ramsey experiment. The oscilloscope measurements below represent three iteration steps of such Ramsey experiment.

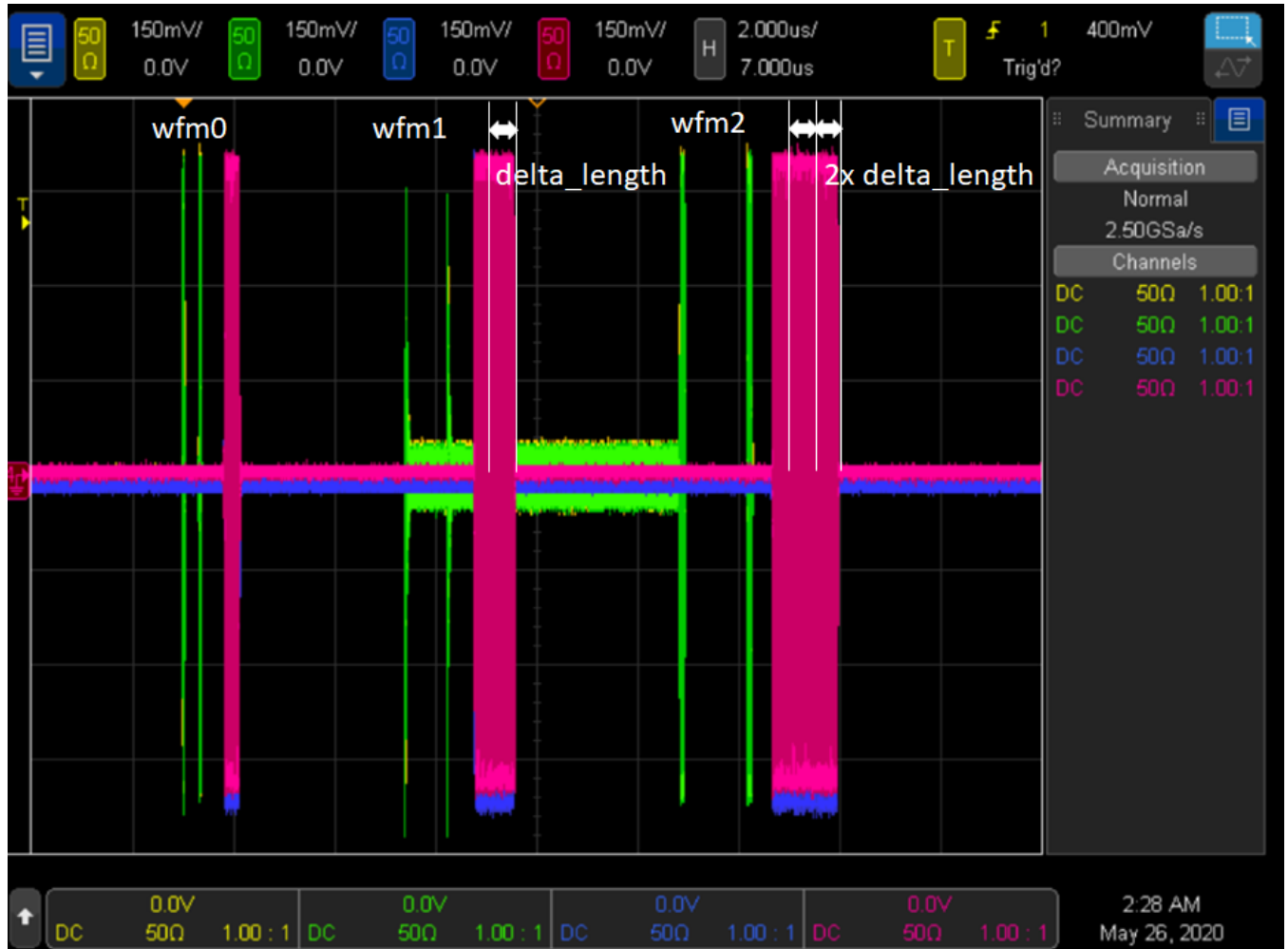


Finally, two additional features included in the experiment template of this programming example allow you to:

1. Change the waveform played by the AWG at each iteration of the experiment steps (real-time fast branching).
2. Increment the readout pulse length after each experiment step.

The capability of the AWG to be able to switch in real-time between a pool of different waveforms is also known as fast branching. Users can enable this capability by setting the `num_wfms` (number of waveforms) parameter represented by the Python code Variable `num_wfms`. The number of waveforms the AWG can quickly switch from depends on the waveforms previously loaded to the AWG RAM, within the Python code method `configure_awg()`. M3xxxA AWG RAM allows to load up to 2GB of waveform data and queue up to one million different waveforms. For more details please refer to the [M3xxxA AWG User Guide](http://www.keysight.com) on www.keysight.com. The oscilloscope measurement reported below depicts three experiment steps where the AWG can switch a different waveform at each iteration step and the readout pulse length is incremented at each iteration step by a

quantity defined by the Python code Variable `delta_length` listed among the experiment parameters reported above.



The functionality to increment both the tau delay and the readout pulse length at each experiment iteration step is implemented using the HVI statement `Wait Time`. The selection of a different waveform in real-time is achieved using the [Sync Register Sharing](#) functionality. In the general case, the digitizer instrument can communicate the decision on the next waveform to be played based on processing on the measurement data that contain information on the DUT state. Users can modify this programming example to add custom processing in the digitizer sandbox using Keysight PathWave FPGA. For more information please consult the [PathWave FPGA User Guide](#) on www.keysight.com

HVI Application Programming Interface (API): Detailed Explanations

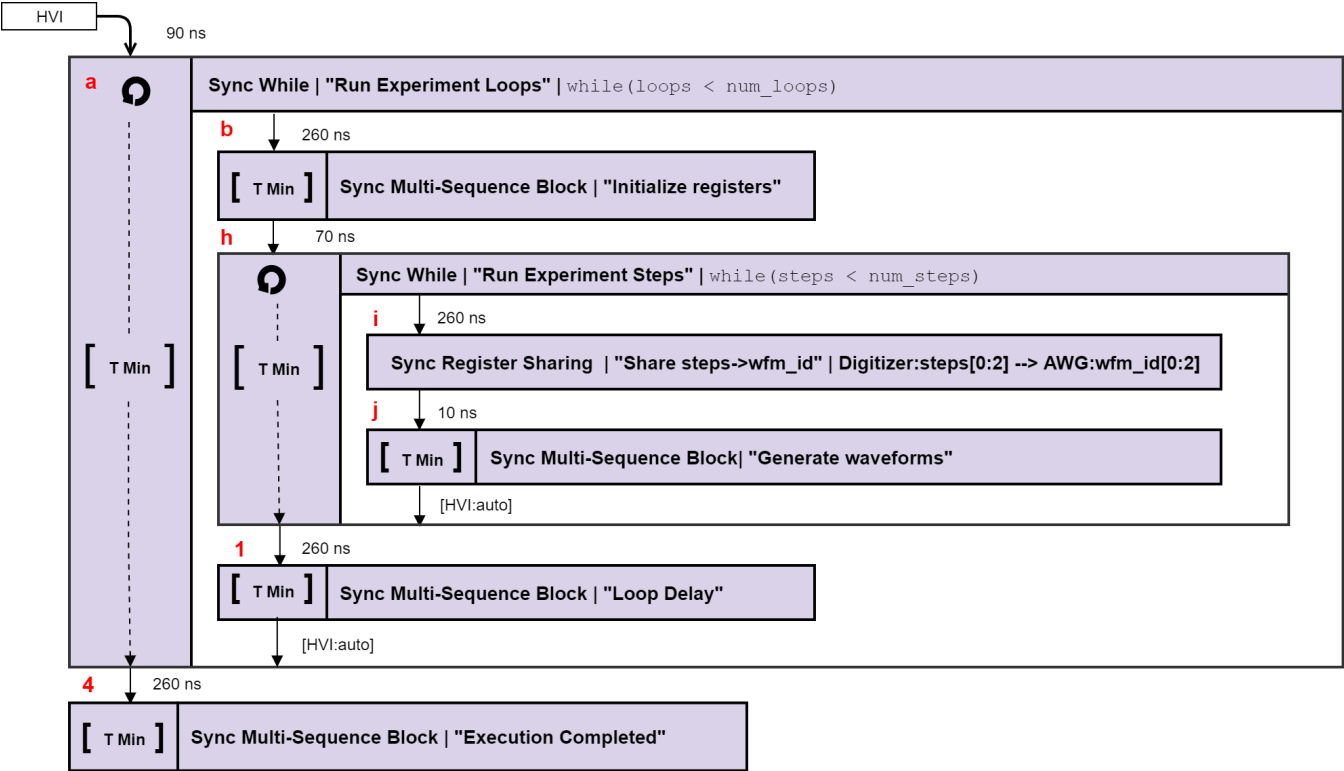
PathWave Test Sync Executive implements the next generation of HVI technology and delivers the HVI Application Programming Interface (API). This section explains how to implement the use case of this

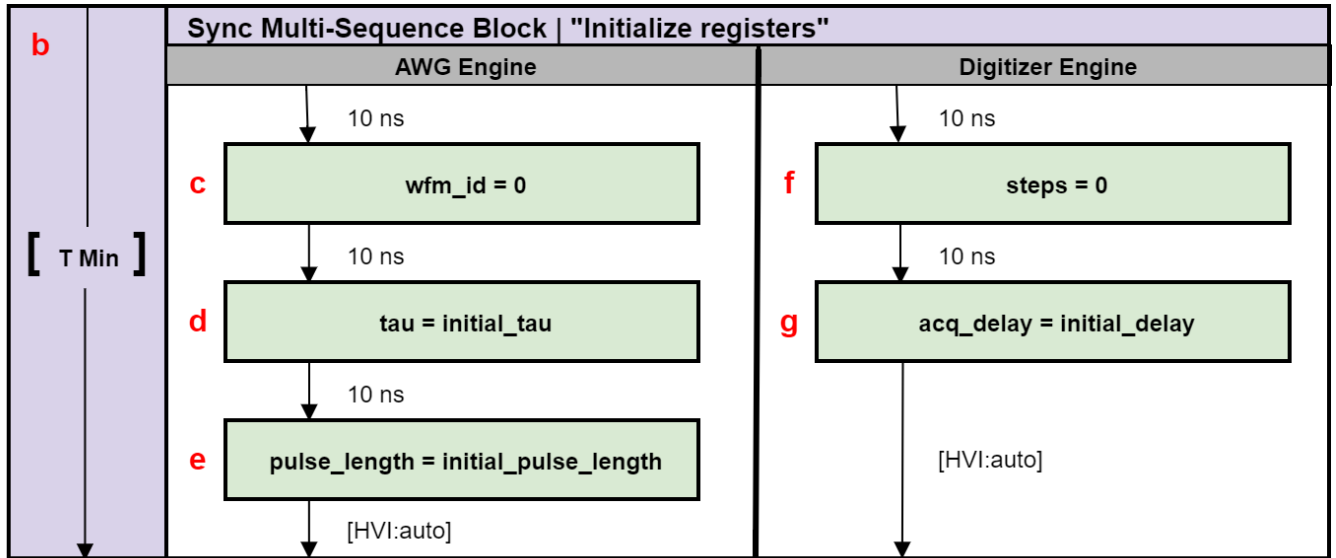
programming example using HVI API. The sequence of operations executed by each of the instruments using HVI technology is explained in the diagram below. The diagram depicts the HVI sequences executed within this programming example and the HVI statements used to program the sequences. Every HVI statement is described in detail later in this section, referencing with a letter the equivalent block in the HVI diagram and explaining in detail the corresponding HVI API code block and the HVI functionalities that it implements.

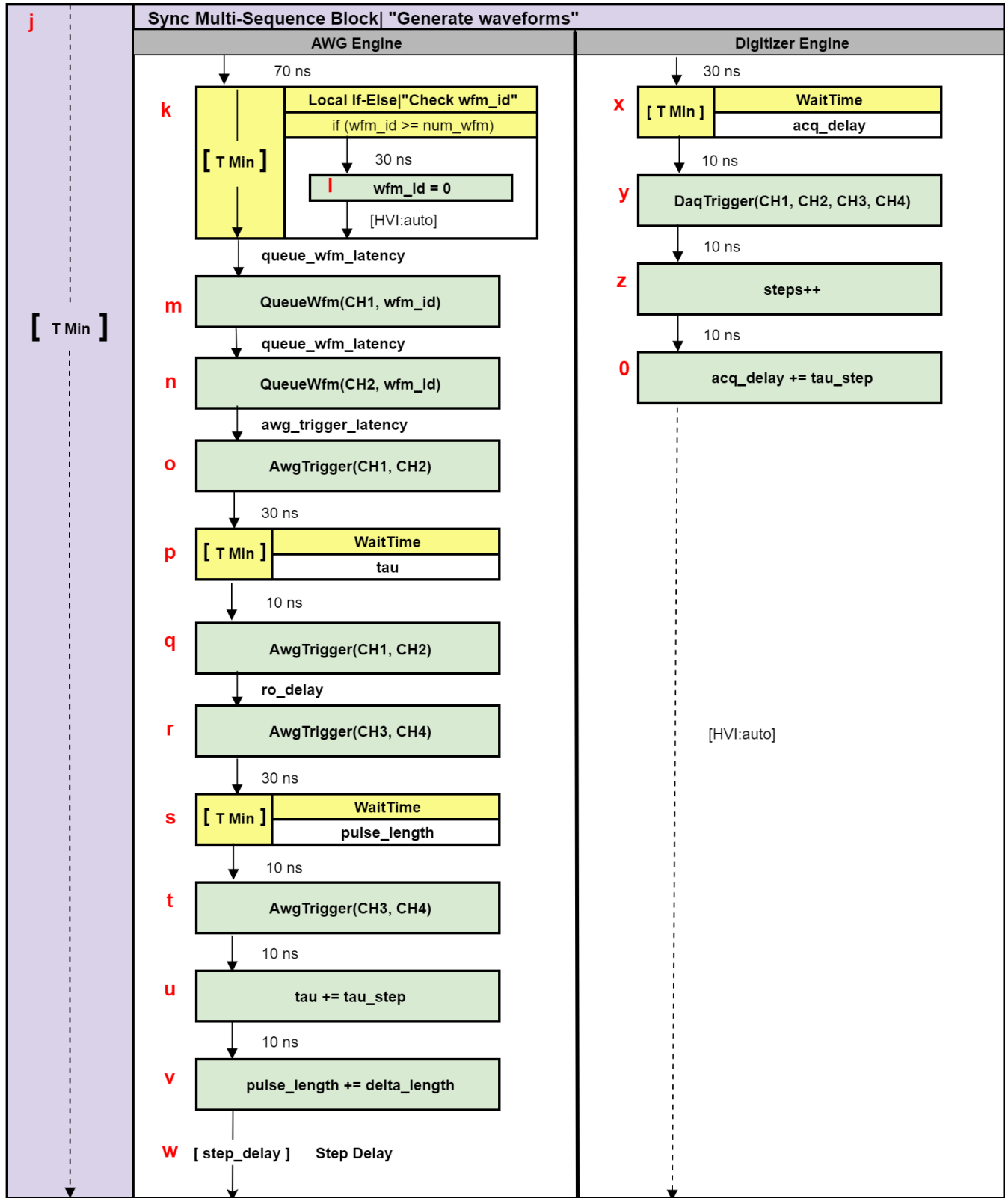
Please note that the start delays of HVI statement inserted in the following HVI diagram are set to very specific values. Unless differently specified, those values correspond to the minimum latencies that can be used for those start delays. Please consult Chapter 7 of the **PathWave Test Sync Executive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

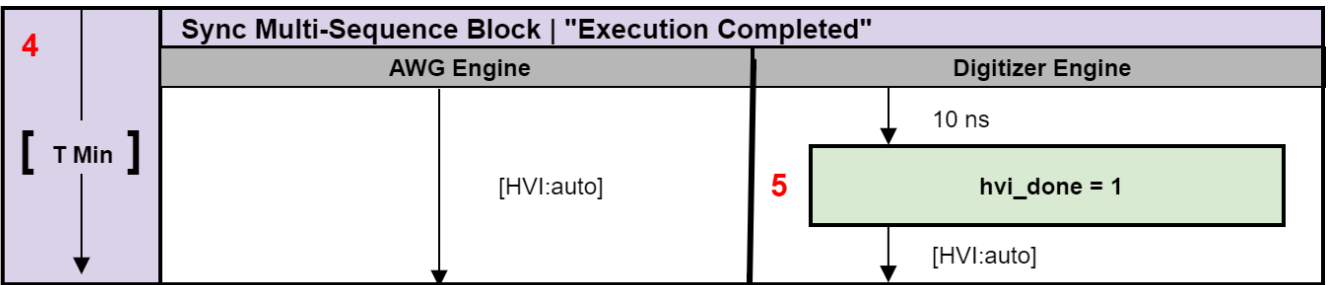
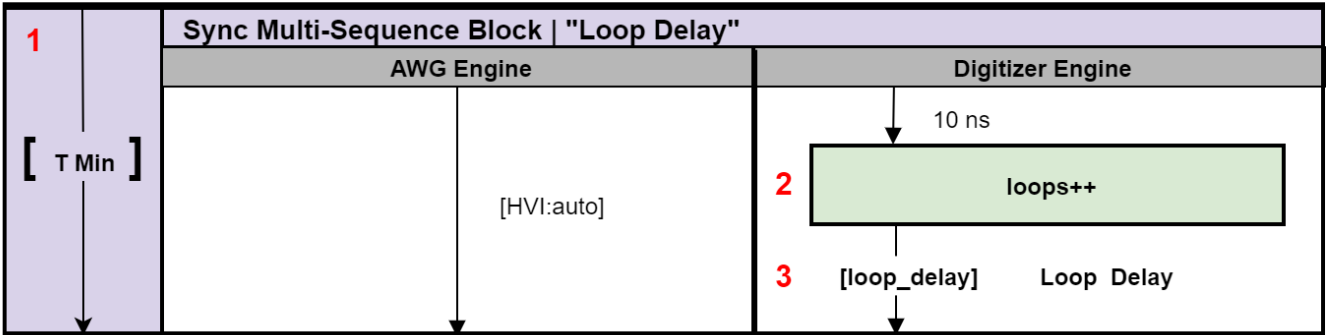
In the HVI diagram below two nested HVI Sync While loops are used to implement the experiment iteration steps and loops. The functionality to increment both the tau delay and the readout pulse length at each experiment iteration step is implemented using the HVI statement Wait Time. Delays between waveforms are implemented using Python code Variables like `ro_delay` when the delay is fixed and not expected to change during the HVI execution or using registers like `tau`, `acq_delay`, when the delay is updated at each iteration of the HVI execution.

HVI instrument-specific instructions are used to queue and play the waveforms from the M3202A AWG. These instructions are represented by the green boxes labeled 'QueueWfm(...)' and 'AwgTrigger(...)' in the HVI diagram depicted below. For additional information about the M3202A AWG functionalities and its HVI definitions please consult the *M3xxxA AWG User Guide* on www.keysight.com.









NOTE This example uses two types of parametrized delays: *fixed delays* and *Variable delays*.

Fixed delays can be parametrized in HVI sequences by using a Variable as the start delay for an HVI statement. In this example, this is done using *ro_delay*, *queue_wfm_latency* and *awg_trigger_latency* properties of the *ApplicationConfig* class. If the fixed delay needs to be placed after the last statement inside a Sync Multi-Sequence Block, the Delay statement can be used. See for example "Step Delay" and "Loop Delay" statements in this example.

Variable delays, i.e., delays expected to change during HVI execution, can be implemented using the WaitTime statement. In this example, this feature is used to change the pulse delay tau, digitizer acquisition delay and readout pulse length at each iteration of the experiment.

NOTE The duration of each iteration of the Sync While loops used in this example is unknown due to the Variable delays implemented using WaitTime statements inside the loops. This is represented by the dotted arrows in the HVI diagram. Due to its unknown duration, it is not possible to use the Sync While duration property to specify how long each experiment step or loop should last.

NOTE

AWG queue waveform and AWG trigger operations require a minimum latency to correctly execute which is specified using Python Variables *queue_wfm_latency* and *awg_trigger_latency*. These Variables can be also updated using the *ApplicationConfigclass*. Using lower values than what specified in this example may cause misbehaviors during HVI execution. This may happen for example because the AWG FPGA FW is not being allowed enough time to real-time queue the waveforms before the command to reproduce them (AWG trigger) is issued. AWG latency information is documented in the M3xxxA AWG documentation and in the SD1 3.x documentation.

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

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

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

System Definition

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

Python

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

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

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, config):
    """ Define HW platform: chassis, interconnections, PXI trigger resources,
    synchronization, HVI clocks
    """
    # Add chassis resources
    # For multi-chassis setup details see programming example documentation
    for chassis_number in config.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 config.M9031_descriptors:
        interconnects = sys_def.interconnects
        for descriptor in config.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 = config.pxi_sync_trigger_resources
    # Assign clock frequencies that are outside the set of the clock frequencies of each
HVI engine
    # Use the code line below if you want the application to be in sync with the 10 MHz
clock
    sys_def.non_hvi_core_clocks = [10e6]
```

Define HVI Engines

All HVI Engines to be included in the HVI instance need to be registered into the EngineCollection class instance. Each HVI Engine object added to the engine collection contains collections of its own that allow you to access the actions, events and triggers that each specific engine will control and use within the HVI. In this programming example, in particular, two HVI engines are used, one for the AWG, the other for the digitizer.

Python

```
# HVI engine Names to be used in this application
self.awg_engine_Name = "AWG Engine"self.dig_engine_Name = "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, module in zip(module_dict.keys(), module_dict.values()):
    sys_def.engines.add(module.instrument.hvi.engines.main_engine, engine_Name)

```

Define HVI Actions, Events, Triggers

In this programming example, both the AWG and the digitizer need to trigger waveforms or acquisition very precisely. To do that the AWG trigger and DAQ trigger actions are issued from within the HVI execution. In the HVI use model, actions need to be added to the action collection of each HVI engine before they can be executed. This is done in this programming example as explained in the code snippets below.

Python

```

# HVI action Names to be used by each HVI engine
self.awg_trigger_Name = "AWG_Trigger"self.daq_trigger_Name = "DAQ_Trigger"

def define_hvi_actions(sys_def, module_dict, config):
    """ This function defines a list of DAQ/AWG trigger actions for each module,
        to be executed by the "action-execute" instructions within the HVI sequence.
        The number of actions in each engine's list depends on the instrument's number of
        channels.
    """ # For each engine, add each HVI Actions to be executed to its own HVI Action
    Collection
        for engine_Name, module in zip(module_dict.keys(), module_dict.values()):
            for ch_index in range(1, module.num_channels + 1):
                # Actions need to be added to the engine's action list so that they can be
                executed
                # Example: hvi.engines[i].actions.add(module_dict[i].hvi.actions.awg1_trigger,
                'AWG1_trigger')
                if engine_Name == config.dig_engine_Name:
                    action_Name = config.daq_trigger_Name+ 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_Name+ str(ch_index) # arbitrary user-
                    defined Name
                    instrument_action = "awg{}_trigger".format(ch_index) # Name decided by
                    instrument API
                action_id = getattr(module.instrument.hvi.actions, instrument_action)
                sys_def.engines[engine_Name].actions.add(action_id, action_Name)

```

Program HVI Sequence

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 core of the HVI diagram consists of two nested Sync while statements that allow you to implement a cycle of experiment steps nested within a number of experiment loops.

Python

```

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

def program_dut_experiment(sequencer, module_dict, config):
    """    This method programs the HVI sequence of this application.
    Different HVI statements are encapsulated as much as possible in separated SW methods
    to help users visualize
    the programmed HVI sequences.
    The programming example documentation on www.keysight.com contains an HVI diagram that
    graphically represents the programmed HVI sequence.
    """    # Define registers within the scope of the outmost sync sequence
    define_registers(sequencer, config)
    # Define sync while condition
    loops = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers[config.loops_
Name]
    sync_while_condition = kthvi.Condition.register_comparison(loops,
kthvi.ComparisonOperator.LESS_THAN, config.num_loops)
    # Add Sync While Statement
    sync_while = sequencer.sync_sequence.add_sync_while("Run Experiment Loops", 90, sync_
while_condition)
    # Program experiment loops
    program_experiment_loops(sync_while.sync_sequence, module_dict, config)
    # Add SMSB statement
    sync_block = sequencer.sync_sequence.add_sync_multi_sequence_block("Execution
Completed", 260)
    # Program the SMSB to Complete the Execution
    program_execution_completed(sync_block, config)

```

Define HVI Registers

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

Python

```

# HVI register Names to be used within the scope of each HVI engine
self.steps_Name = "Steps"self.loops_Name = "Loops"self.wfm_id_Name = "Waveform ID"self.tau_
Name = "Tau"self.pulse_length_Name = "Pulse Length"self.acq_delay_Name = "Acquisition

```

```

Delay"self.awg_counter_Name = "AWG Counter"self.dig_counter_Name = "Digitizer
Counter"self.hvi_done_Name = "HVI Done"

def define_registers(sequencer, config):
    """    Defines all registers for each HVI engine in the scope of the global sync
sequence
    """    # Digitizer registers
        loops = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.loops_Name, kthvi.RegisterSize.SHORT)
        loops.initial_value = 0
        steps = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.steps_Name, kthvi.RegisterSize.SHORT)
        steps.initial_value = 0
        acq_delay = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.acq_delay_Name, kthvi.RegisterSize.SHORT)
        acq_delay.initial_value = 0
        loop_delay = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.loop_delay_Name, kthvi.RegisterSize.SHORT)
        loop_delay.initial_value = config.loop_delay
        hvi_done = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.hvi_done_Name, kthvi.RegisterSize.SHORT)
        hvi_done.initial_value = 0
        dig_counter = sequencer.sync_sequence.scopes[config.dig_engine_Name].registers.add
(config.dig_counter_Name, kthvi.RegisterSize.SHORT)
        dig_counter.initial_value = 0
        # AWG registers
        awg_counter = sequencer.sync_sequence.scopes[config.awg_engine_Name].registers.add
(config.awg_counter_Name, kthvi.RegisterSize.SHORT)
        awg_counter.initial_value = 0
        tau = sequencer.sync_sequence.scopes[config.awg_engine_Name].registers.add(config.tau_
Name, kthvi.RegisterSize.SHORT)
        tau.initial_value = 0
        wfm_id = sequencer.sync_sequence.scopes[config.awg_engine_Name].registers.add
(config.wfm_id_Name, kthvi.RegisterSize.SHORT)
        wfm_id.initial_value = 0
        pulse_length = sequencer.sync_sequence.scopes[config.awg_engine_Name].registers.add
(config.pulse_length_Name, kthvi.RegisterSize.SHORT)
        pulse_length.initial_value = 0
        step_delay = sequencer.sync_sequence.scopes[config.awg_engine_Name].registers.add
(config.step_delay_Name, kthvi.RegisterSize.SHORT)
        step_delay.initial_value = config.step_delay

```

Synchronized While

Synchronized While appears in statements (a, h). Synchronized While (Sync While) statements belong to the set of HVI Sync Statements and are defined by the API class *SyncWhile*. A Sync While allows you to synchronously execute multiple local HVI sequences until a user-defined condition is met, that is, the sync while condition. For local sequences to be defined within the Sync While, it is necessary to use synchronized multi-sequence blocks.

Python

```

# Define sync while condition
sync_while_condition = kthvi.Condition.register_comparison(loops,

```

```
kthvi.ComparisonOperator.LESS_THAN, exp_params.num_loops)
# Add Sync While Statement
sync_while = sync_sequence.add_sync_while('Run Experiment Loops', 90, sync_while_condition)
```

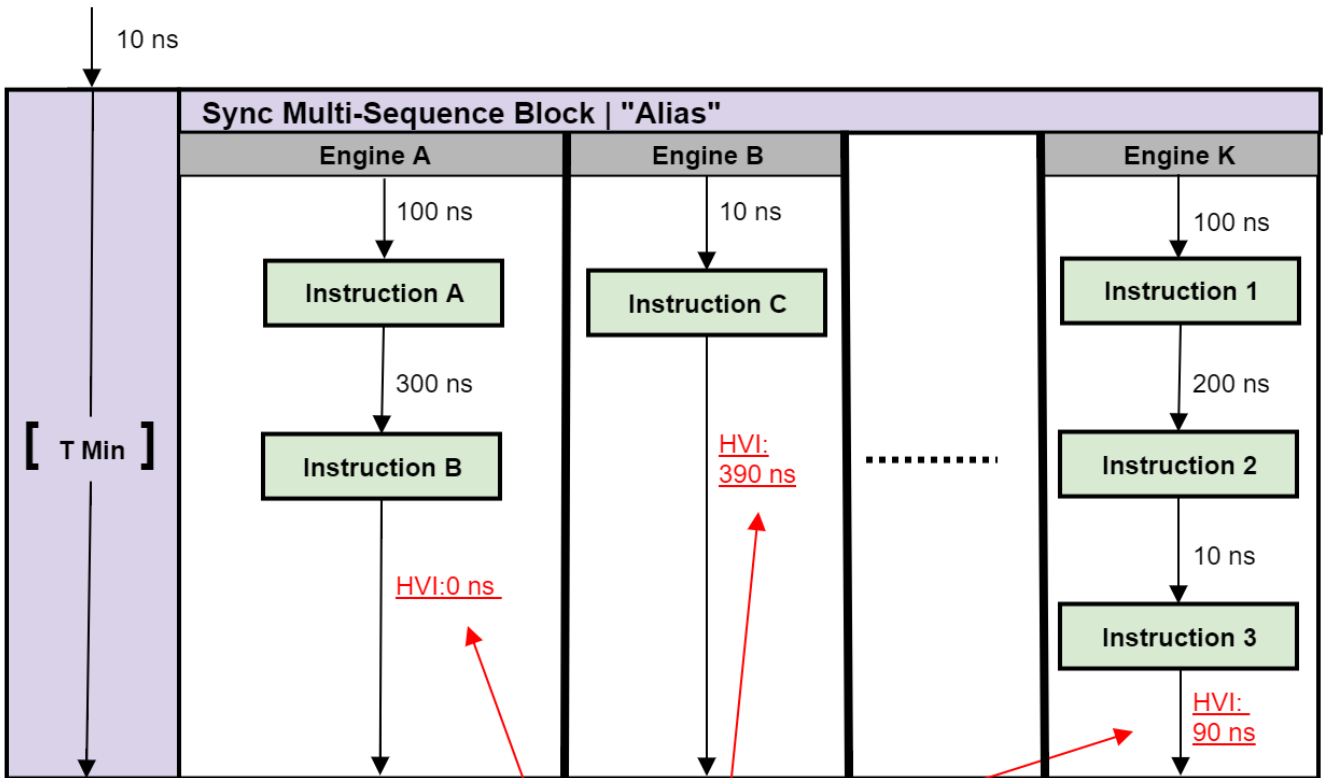
Synchronized Multi-Sequence Block

It can be found in statements (b, j, 1, 4) of 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('Initialize registers', 160)
```

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.



Automatically caclulated by HVI

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

Please note that the Sync Multi-Sequence Block has an execution duration time labeled as "T Min" in the figure above. The "T Min" default value for any sync statement corresponds to the minimum time necessary to complete the operations included inside. KS2201A Update 1.0 release provides the *Duration* property in Sync Statement objects that allows users to set an arbitrary duration value larger than "T Min". The timing at the end of each local sequence is automatically adjusted by HVI according to the duration specified by the user for the SMSB. In the case of duration "T min", HVI will automatically add no time to the local sequence with the longest duration and adjust the other sequences accordingly, as in the example depicted in the figure above. The resolution for HVI-defined time adjustment at the end of a sync multi-sequence block corresponds to the 10 ns FPGA clock period for an application including instruments that are all within the Keysight M3xxxA family. For further explanations about the timing of HVI sequence execution please refer to the www.keysight.com **KS2201A PathWave Test Sync Executive User Manual** available on www.keysight.com

HVI Native Instruction: Register Assign

Statements (c, d, e, f, g, l, 5) are Register Assign instructions. 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 before the HVI execution, by using the property *initial_value*.

Python

```
# Load previously defined parameters and resources
awg_sequence = sync_block.sequences[config.awg_engine_Name]
tau = awg_sequence.scope.registers[config.tau_Name]

# Initialize tau = initial_tau
instruction = awg_sequence.add_instruction("tau = initial_tau", 10, awg_
sequence.instruction_set.assign.id)
instruction.set_parameter(awg_sequence.instruction_set.assign.destination.id, awg_
sequence.scope.registers[config.tau_Name])
instruction.set_parameter(awg_sequence.instruction_set.assign.source.id, config.initial_
tau)
```

Sync Register Sharing

This corresponds to statement (i) in the HVI diagram. Register sharing is a functionality defined and programmed using the *RegisterSharing* class. Register sharing allows you to share the content of N adjacent bits of a source register and write the information to a destination register in any of the other HVI engines included in the HVI execution. In this programming example, this functionality is used to share the content of the digitizer register *steps* and write into the AWG register *wfm_id* to use it to select real-time the waveform to be played at each experiment step. In this programming example, the register step is incremented at each iteration of the experiment inner loop. In a more generic case, the feedback loop from the digitizer to the AWG can include more complex processing on the acquired measured data so that the AWG can fast branch among the different possible waveforms in response to the feedback from the digitizer. Keysight offers PathWave FPGA software as a design environment to implement complex data processing into the instrument FPGA to be used for example for such feedback loop. For more information please consult the **PathWave FPGA User Manual** on www.keysight.com

Python

```
# Previously defined registers
steps = sync_sequence.scopes[config.dig_engine_Name].registers[config.steps_Name]
wfm_id = sync_sequence.scopes[config.awg_engine_Name].registers[config.wfm_id_Name]

# Add sync register sharing
bits_to_share = 2
sync_sequence.add_sync_register_sharing("Share steps->wfm_id", 260, steps, wfm_id, bits_to_
share)
```

IF-ELSEIF-ELSE Statement

This corresponds to the statement (k) in the HVI diagram. IfStatement class allows you to add an IF-ELSEIF-ELSE statement within the main HVI sequence of any instrument engine. The IF-ELSEIF-ELSE statement 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

```
# Previously defined resources
wfm_id = sync_sequence.scopes[config.awg_engine_Name].registers[config.wfm_id_Name]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]

# Define If condition and parameters
if_condition = kthvi.Condition.register_comparison(wfm_id,
kthvi.ComparisonOperator.GREATER_THAN_OR_EQUAL_TO, config.num_wfms)
enable_ifbranches_time_matching = True
# Add If statement
if_statement = awg_sequence.add_if("Check wfm_id", 70, if_condition, enable_ifbranches_
time_matching)
if_branch_seq = if_statement.if_branch.sequence
# Reset wfm_id = 0 within the IF sequence
instruction = if_branch_seq.add_instruction("wfm_id = 0", 30, awg_sequence.instruction_
set.assign.id)
instruction.set_parameter(awg_sequence.instruction_set.assign.destination.id, wfm_id)
instruction.set_parameter(awg_sequence.instruction_set.assign.source.id, 0)
```

HVI Instrument-Specific Instruction: Queue AWG Waveform

This corresponds to statements (m, n) in the HVI diagram. This statement executes a product-specific HVI instruction. The API method `add_instruction()` allows you to add the required 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

```
# Previously defined resources
wfm_id = sync_sequence.scopes[config.awg_engine_Name].registers[config.wfm_id_Name]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]

# Queue Readout rising edge waveform to CH3, CH4
for awg_ch in range(3, 5):
    instrLabel = "Queue RWaveRise CH" + str(awg_ch)
    instruction0 = awg_sequence.add_instruction(instrLabel, config.queue_wfm_latency, awg_
module.hvi.instruction_set.queue_waveform.id)
    #Set every parameter of AWGQueueWaveform(awg_ch, waveformNumber, triggerMode,
startDelay, cycles, prescaler);
    instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.waveform_
number.id, config.rorise_id)
    instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.channel.id,
awg_ch)
```

```

instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.trigger_
mode.id, config.trigger_mode)
instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.start_
delay.id, config.start_delay)
instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.cycles.id, 1)
instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.prescaler.id,
config.prescaler)

```

Action Execute: AWG trigger, DAQ trigger

This type of instruction can be found in statements (o, q, r, t, y). 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

```

## Previously defined resources
wfm_id = sync_sequence.scopes[config.awg_engine_Name].registers[config.wfm_id_Name]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]
awg_trigger_12 = [
    awg_sequence.engine.actions[config.awg_trigger_Name + str(1)],
    awg_sequence.engine.actions[config.awg_trigger_Name + str(2)]]

# AWG trigger CH1, CH2 - Generates first pulse
inst_awg_trigger = awg_sequence.add_instruction("AwgTrigger(CH1, CH2)", config.awg_trigger_
latency, awg_sequence.instruction_set.action_execute.id)
inst_awg_trigger.set_parameter(awg_sequence.instruction_set.action_execute.action.id, awg_
trigger_12)

```

Wait Time

This type of statement can be found in statements (p, s, w). Inserting an instance of WaitTime instruction class causes an HVI sequence to wait for an amount of time specified by a register previously added to the same HVI sequence. The register used needs to be initialized before its usage. The time unit is expressed as an integer multiple of the instrument clock cycle duration. For example, in M3xxxA PXI modules a clock cycle lasts 10 ns.

Python

```

# Previously defined resources
tau = sync_sequence.scopes[config.awg_engine_Name].registers[config.tau_Name]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]

# WaitTime: tau
awg_sequence.add_wait_time('WaitTime: tau', 30, tau)

```

Register Increment

This type of instruction can be found in statements (u, v, z, 0, 2). 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 have been added previously to the scope of the corresponding HVI engine.

Python

```
# Previously defined resources
tau = sync_sequence.scopes[config.awg_engine_Name].registers[config.tau_Name]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]

# tau += tau_step
instruction = awg_sequence.add_instruction('tau += tau_step', 10, awg_sequence.instruction_set.add.id)
instruction.set_parameter(awg_sequence.instruction_set.add.destination.id, tau)
instruction.set_parameter(awg_sequence.instruction_set.add.left_operand.id, tau)
instruction.set_parameter(awg_sequence.instruction_set.add.right_operand.id, config.tau_step)
```

Delay Statement

This type of statement can be found in statements (w, 3). Inserting an instance of *DelayStatement* class causes an HVI sequence to wait for a fixed amount of time that is known at compilation time and it is not expected to change during HVI execution. The amount of time is specified in nanoseconds. The Delay Statment functions like the start delay parameter used in each method that programs a statement into an HVI sequence. The main difference is that a start delay allows specifying a delay before a statement, whereas the delay statement allows to specify it afterward, for example at the end of a Sync Multi-Sequence Block, as it is used in this programming example. To specify a Variable delay that can change during HVI execution, one shall use the WaitTime statement instead.

Python

```
# Step delay
awg_sequence.add_delay("Step Delay", config.step_delay)
```

Export the Programmed HVI Sequences to File

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

```
# Generate HVI sequence description text
output = sequencer.sync_sequence.to_string(kthvi.OutputFormat.DEBUG)
print("Programmed HVI sequences exported to file")
```

Compile, Load, Execute the HVI

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

Compile HVI

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

Python

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

Load HVI to Hardware

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

Python

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

Execute

HVI 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>.

Conclusions

This Programming Example showed how to use an M320xA AWG and an M3102A digitizer to perform a real-time pulsed characterization experiment on a Device-Under-Test (DUT). Register sharing functionality was used to establish a feedback loop between the digitizer and the AWG. This way the digitizer can select real-time the waveform to be played by the AWG at each experiment iteration step. Wait Time functionality of PathWave Test Sync Executive was used to change real-time the delay between subsequent characterization pulses sent to the DUT within each experiment step. It was also shown how pulse duration can be increased real-time using the same functionality. It was shown how users can choose to repeat the experiment for a user-defined number of loops. Users can also customize the pulse characterization experiment by setting the experiment parameters as explained in the application note. Example measurement results showed how the application code can produce the I-Q pulses necessary to perform T1 and T2 characterization experiments on quantum bits for quantum applications. The same application code can also be used for power amplifier characterization for 5G mobile communications or other type of DUT characterization.