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.

KEYSIGHT TECHNOLOGIES

Table of Contents

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.

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 2020 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)

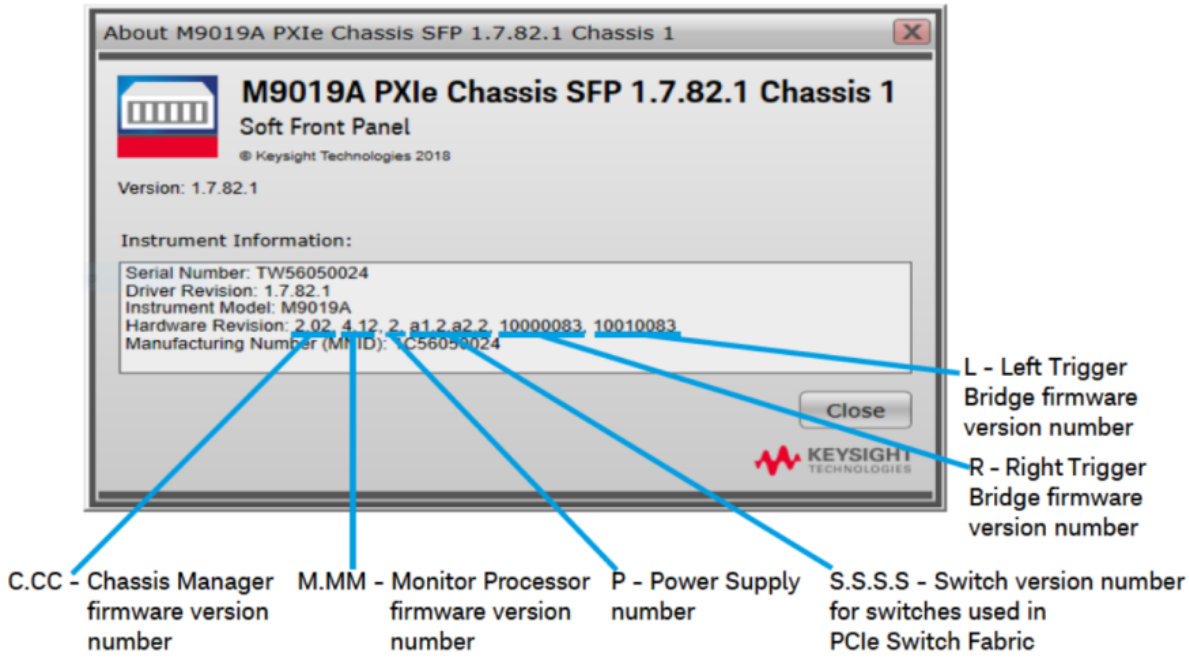
3. M3xxxA with -HVx HW option and following FPGA firmware versions (to be installed using Keysight SD1 SFP):
   - M3202A AWG FPGA firmware (v4.00.95 or later)
   - M3102A Digitizer FPGA firmware (v2.01.40 or later)

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

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 the *module_descriptor* are used. Please update the properties in each *module-descriptor* object before running the programming example:

```
# Update module descriptors below with your instruments information
digitizer_descriptor = module_descriptor('M3102A', 1, 9, options, hvi_eng_Names.dig_engine)
awg_descriptor = module_descriptor('M3202A', 1, 8, options, hvi_eng_Names.awg_engine)



class module_descriptor:
    # 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 = [M9031_descriptor(1, 11, 2, 11)]

class M9031_descriptor:
    # 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 forsynchronization, 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 line 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:

```
class AWG_parameters:
    # Configures AWG for waveform generation
    def __init__(self):
        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.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.trigger_mode = keysightSD1.SD_TriggerModes.SWHVITRIG_CYCLE
        # Latency values for M3202A AWGqueueWfm() [ns]
        # Latencies depend on AWG FPGA FW. Check M3xxxA User Guide for further info
        # Minimum start delay necessary to execute an AWGqueueWfm() instruction
               self.queuewfm_latency = 50 # [ns]
               # Minimum latency necessary between an AWGqueueWfm() instruction and an AWGtrigger
action.
        self.awgtrigger_latency = 2300 # [ns]
        # 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



class DIG_parameters:
    # Configures Digitizer parameters
    def __init__(self):
        exp_params = Experiment_parameters()
        awg_params = AWG_parameters()
```

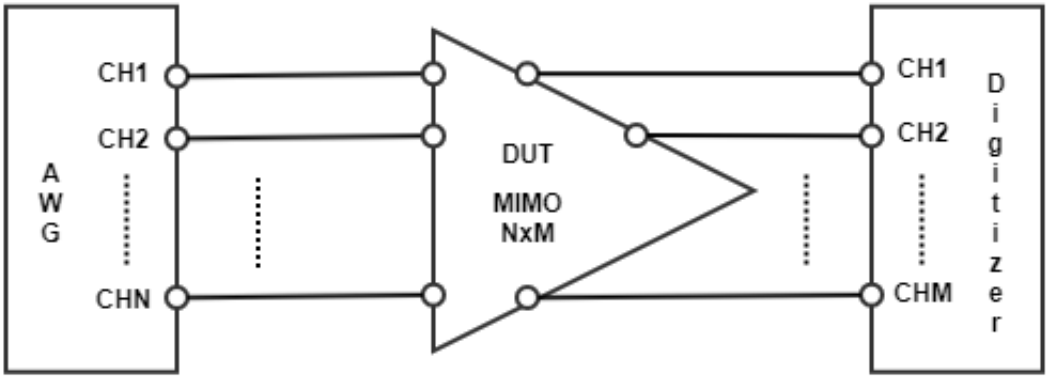
```
        all_ch_mask = 0xF
        sampling_time = 2 # [ns] 1/sample_rate, sample_rate = 500 MSa/s for Digitizer
M3102A
        acquisition_points_per_cycle = int(exp_params.acquisition_window / sampling_time) #
[Sa]
        self.prescaler = 0 # Prescaler values are explained in M3xxxA User Guide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

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

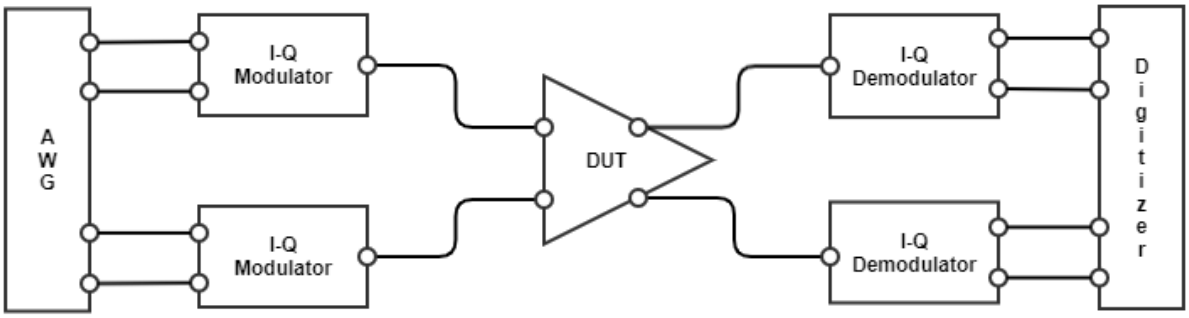
Overview

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

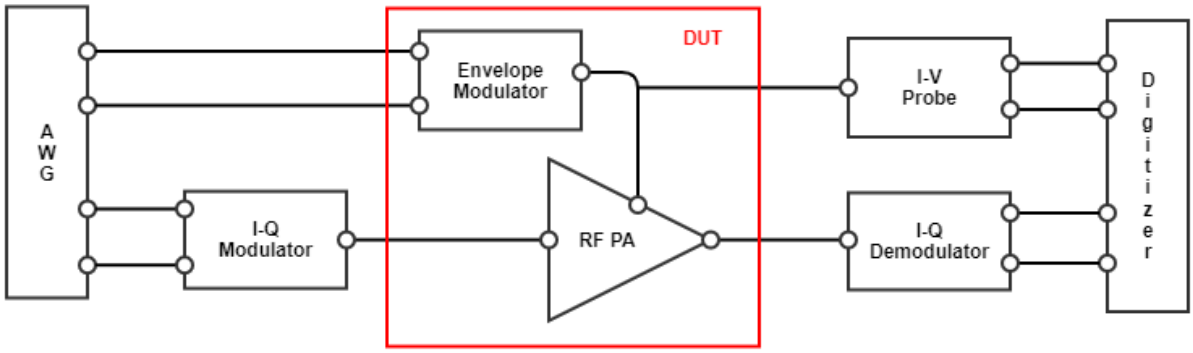


In the general case, this programming example can be deployed on Multiple-Input Multiple-Output (MIMO) Device-Under-Tests (DUTs). The number of inputs and outputs depends on the DUT. To deploy this programming example on an NxM MIMO DUT, it is necessary to use an AWG with N channels and a digitizer with 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 4x4, or MIMO 2x2 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.

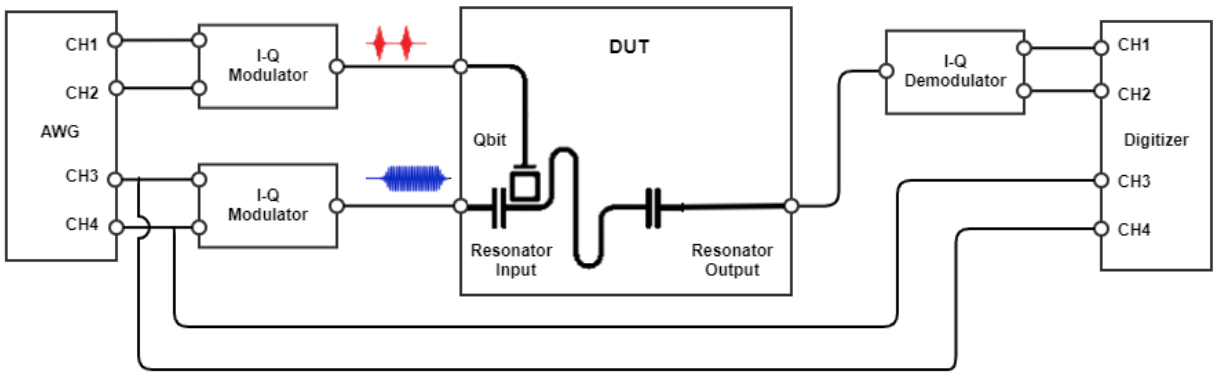


As an example, Radio Frequency (RF) Power Amplifiers (PAs) used in mobile communications are typically Single-Input Single-Output (SISO) systems, but the latest advanced transmitter configuration for the 5th Generation (5G) of mobile communications can include multiple amplifiers configured together 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.

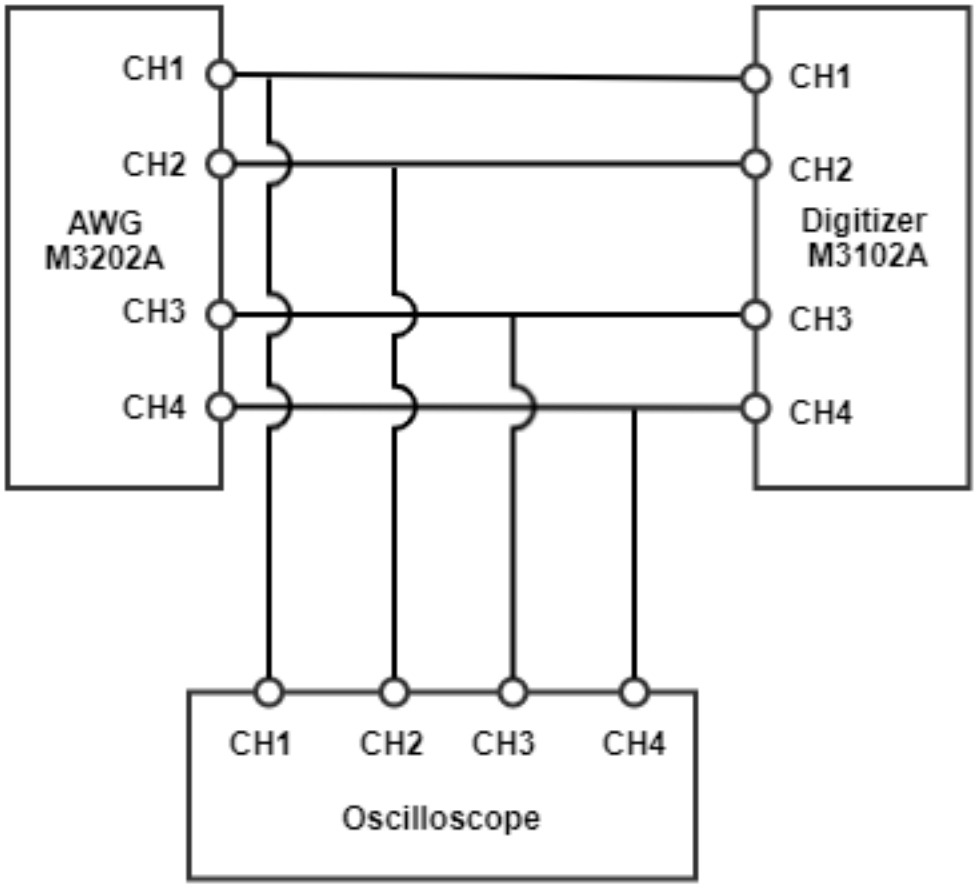


In particular, to address the ET PA characterization use case, users might prefer to substitute the example pulsed waveforms used in this programming example with real telecommunication waveform data samples. The usage of I-Q modulators/demodulators, I-V probe is not covered in this programming example. 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.

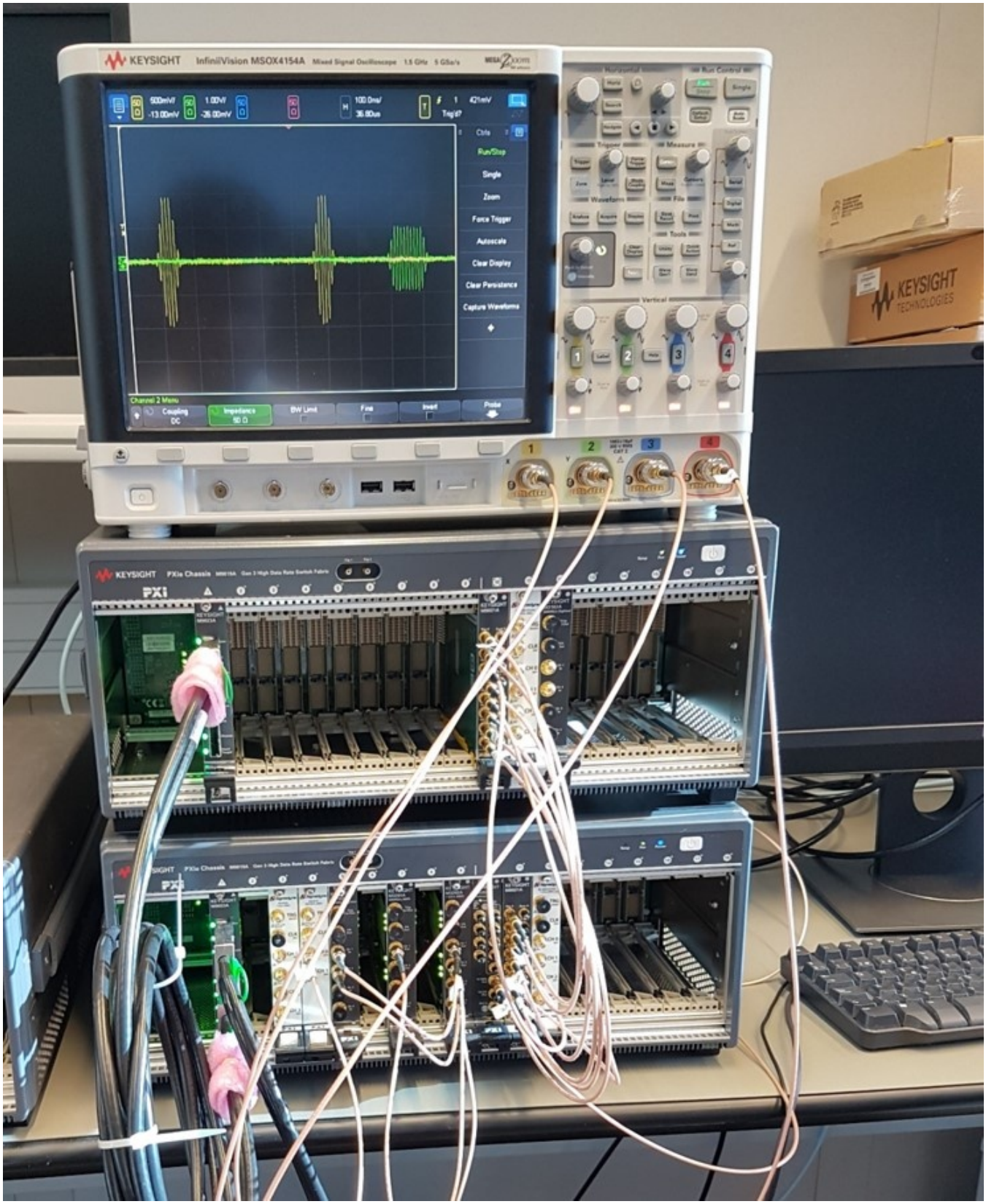
Another interesting use case is the characterization of quantum bits (Qbits) for quantum applications. Such applications can be covered by this programming example using a setup similar to the one represented in the figure below.



The arbitrary waveforms loaded to the AWG RAM in this programming example include the pi and pi/2 gaussian pulses typically used as Qbit excitation signals. The measurement results shown in the rest of this section 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 T1 (also known as energy relaxation time) and the Qbit dephasing time T2. 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 are connected to the corresponding channel of the M3102A digitizer and to the corresponding channel of a Keysight oscilloscope, using an T-connector.



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

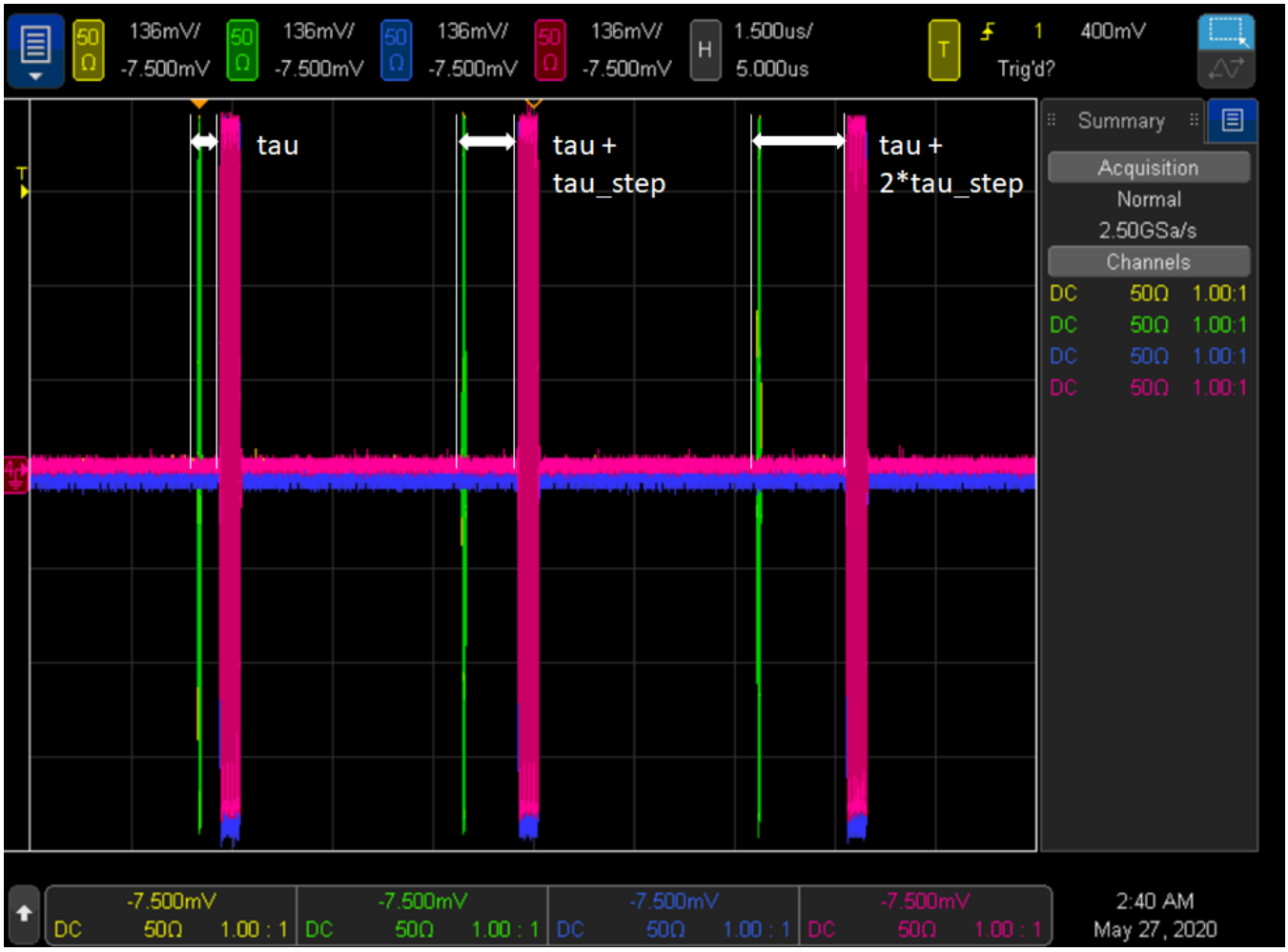

KEYSIGHT
InfiniiVision MSOX4154A
Mixed Signal Oscilloscope
Run/Stop
Single
Zoom
Force Trigger
Autoscale
Clear Display
Clear Persistence
Capture Waveforms
KEYSIGHT
PXIe Chassis
PXI
KEYSIGHT
TECHNOLOGIES

Measurement Results

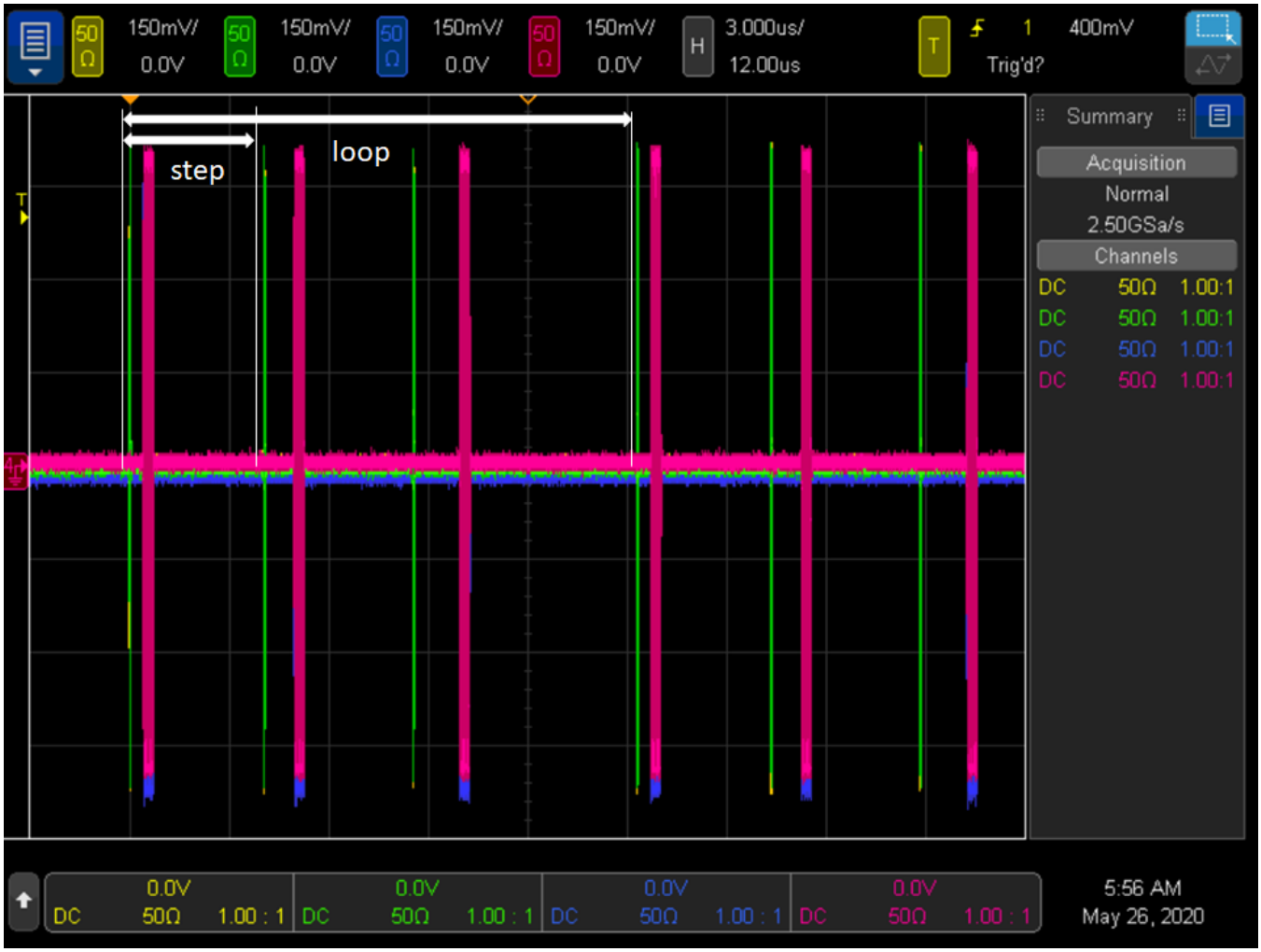
The first step to run this programming example is to define the experiment parameters. The example measurement results reported in the rest of this document are obtained with the experiment parameter values set as follows:

```
class Experiment_parameters:
    # Configures the experiment parameters
    def __init__(self):
        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 = 50 # x10[ns] # Time that is incrementally added to delay between
the control and readout pulse, in ns
        self.ro_delay = 10 # [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 = 100 # x10[ns] # Time to wait between each experiment loop
        self.initial_acq_delay = 230 # x10[ns] # Delay before starting to capture waveforms
with digitizer
        self.acquisition_window = 1000 # [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 = 0 # x10[ns] # Duration increment of the readout pulse length at
each step
        self.num_steps = 3 # Number of iterations to increase tau by tau_step
        self.num_loops = 2 # Number of experiments to execute
```

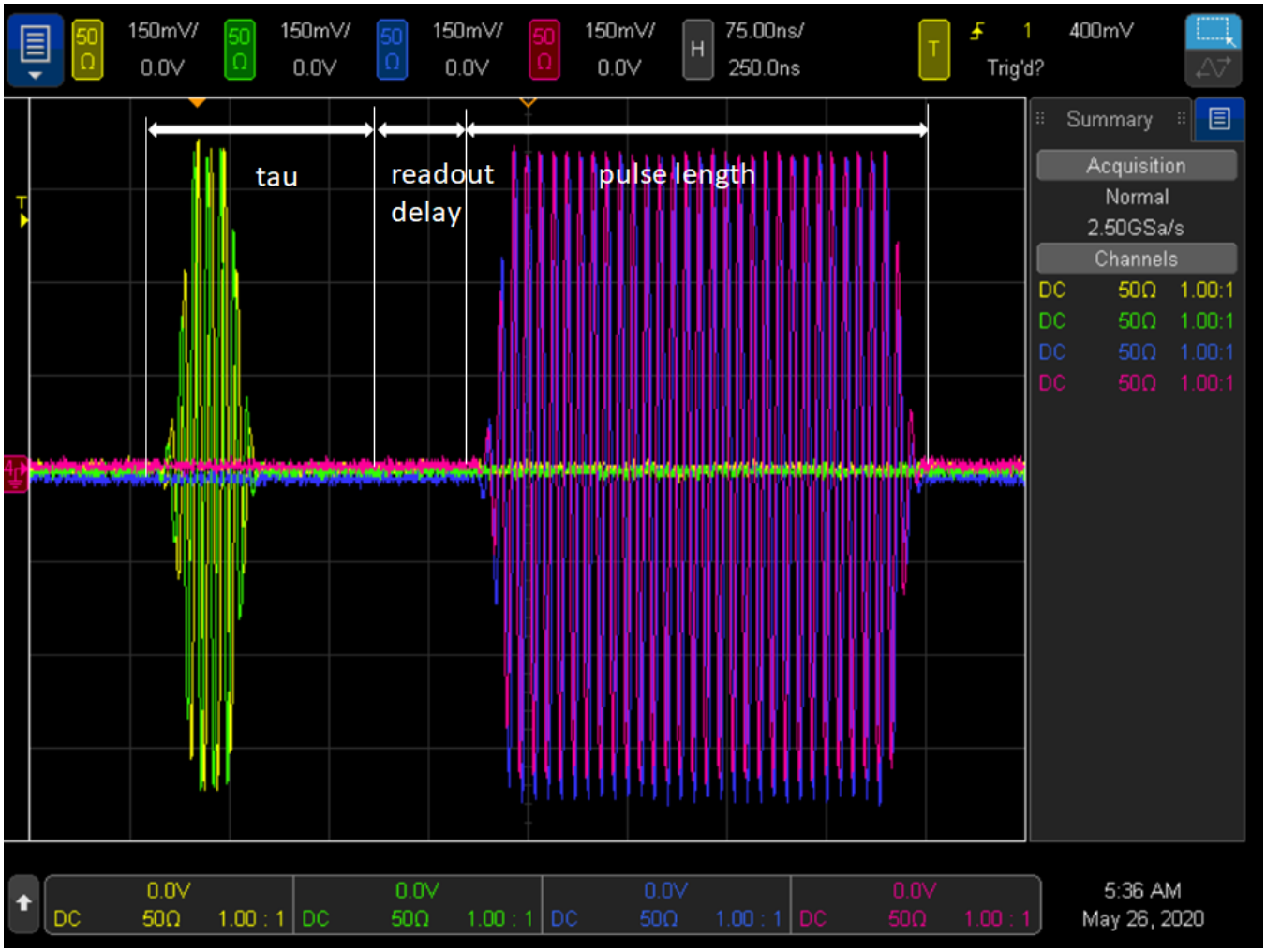
The experiment repeats for a number of iteration steps. At each step parameters such as the delay tau between the saturation pulse and the readout pulse can be incremented by an incremental quantity defined as an experiment step (*tau_step* Python code Variable listed above). Each step iteration is repeated 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.



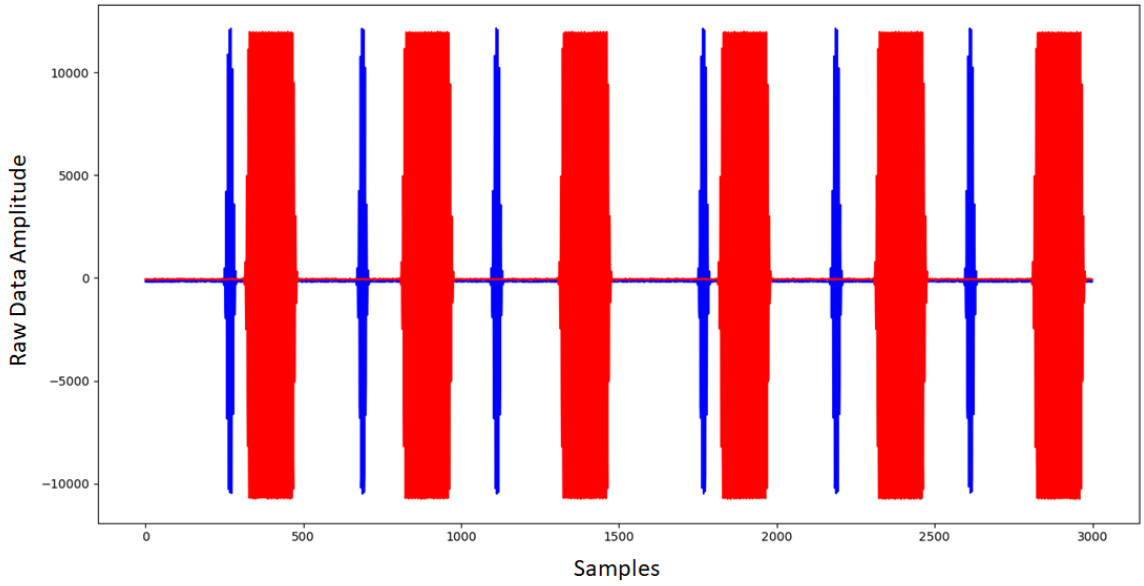
The experiment is then repeated for a number of experiment loops. Each experiment loop can start after a user-defined loop delay to allow the DUT to return into its equilibrium state before the next series of experiment steps can be performed. By increasing the number of experiment loops the user can collect repeated DUT measurements that can allow to calculate a 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*).



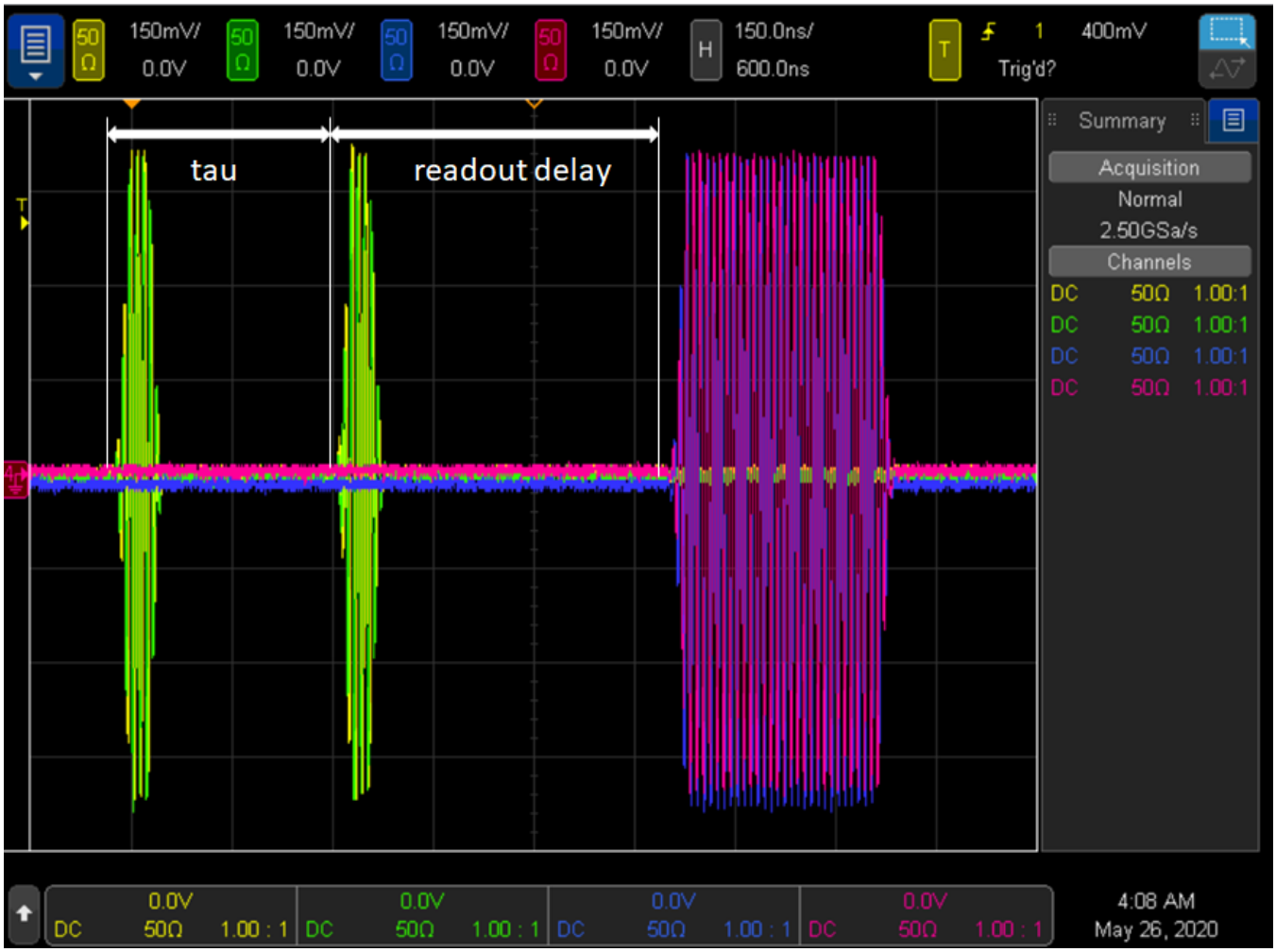
The oscilloscope measurement below represents the experiment parameters tau, readout delay and readout pulse length, all implemented using HVI registers. More details on the HVI resources and sequences programmed to implement the programming example functionalities are provided in the next section.



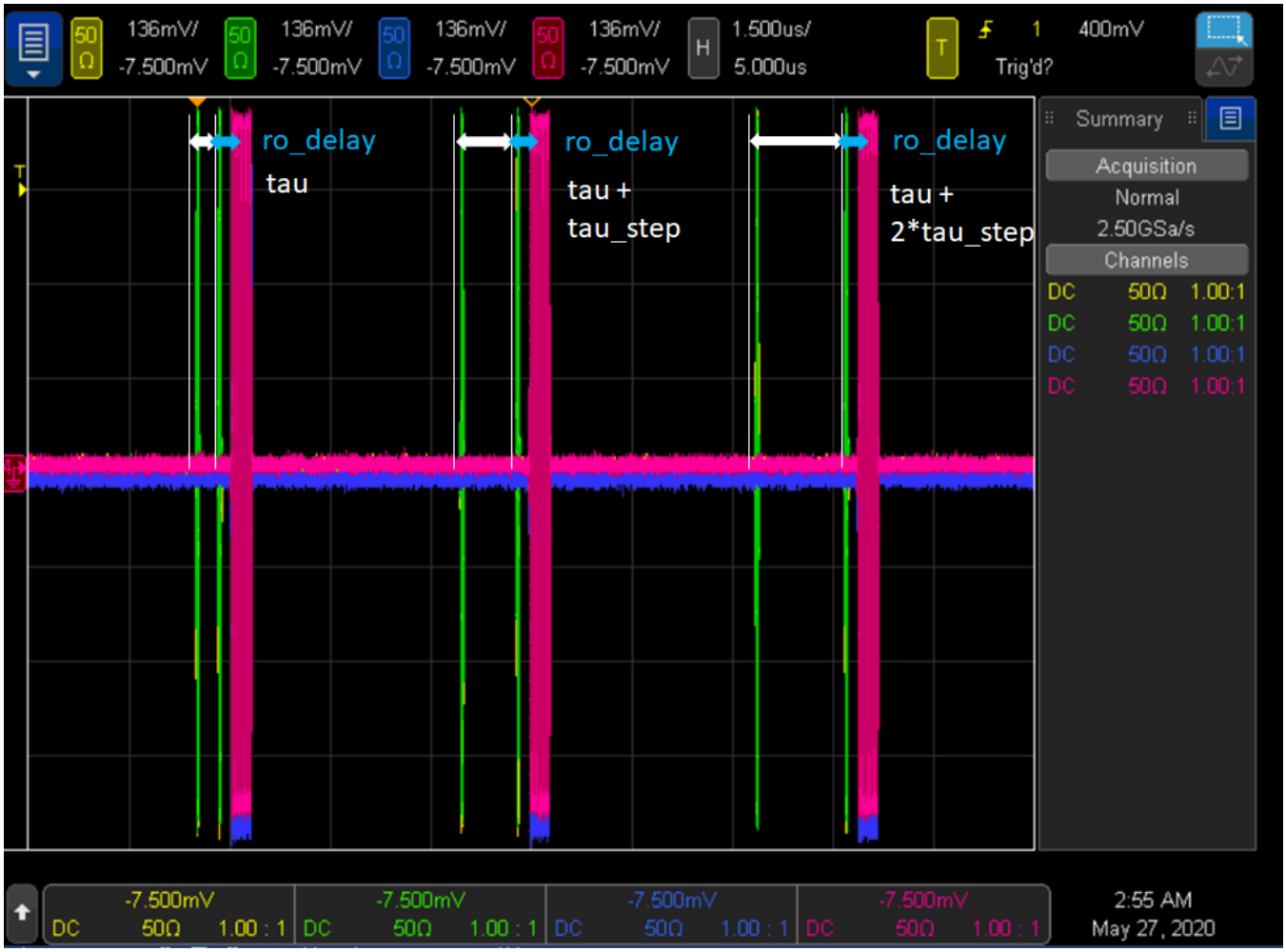
Thanks to the powerful synchronization capabilities of **PathWave Test Sync Executive** and HVI technology, each digitizer acquisition cycle can be precisely triggered synchronously with the time window of the waveform generated by the AWG. Users can adjust the starting point of the acquisition time 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 tau 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.



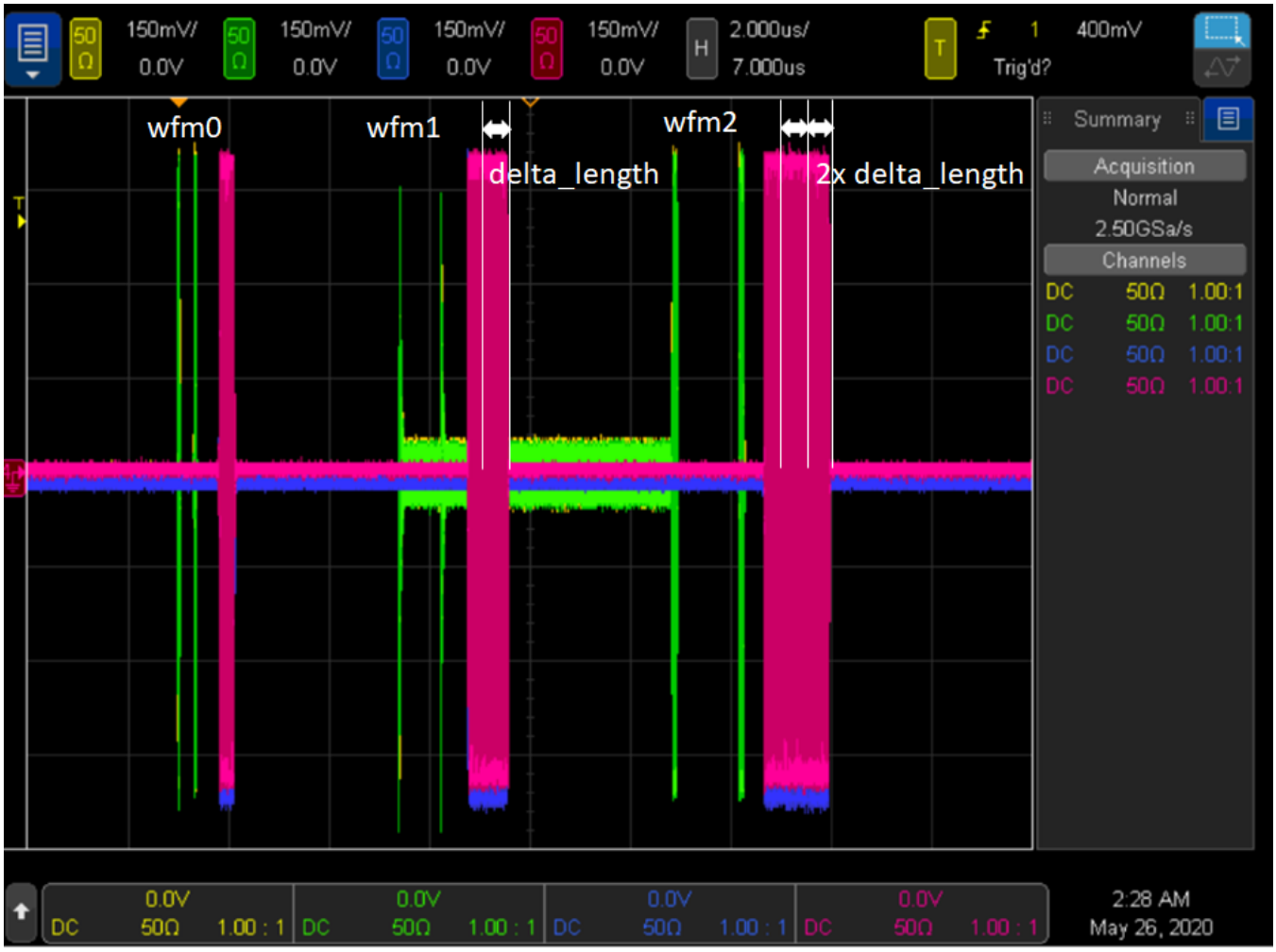
The activation of the T2_flag parameters allows to run this programming example to perform an experiment typically used for the characterization of the the T2 time, i.e. dephasing time of quantum bits. This experiment is also known as Ramsey experiment. The osciloscope measurements below represent three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



Finally, two additional features included in the experiment template of this programming example allow 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 real-time between a pool of different waveforms is also known as fast branching. Users can enable this capability by setting the 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** 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 are implemented using the HVI statement Wait Time. The selection of a different waveform real-time is achieved using the Sync Register Sharing functionality. In the general case the digitizer instrument can communicate the decision on the next wavefrom 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

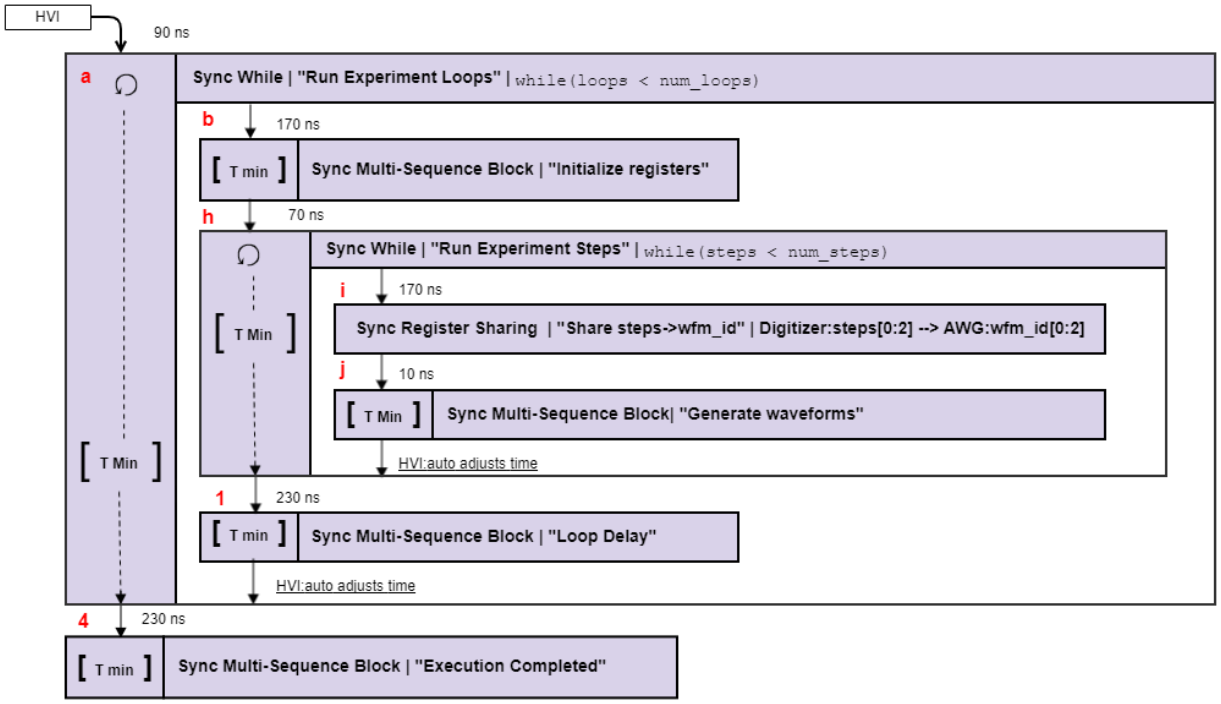
Getting Started with HVI

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

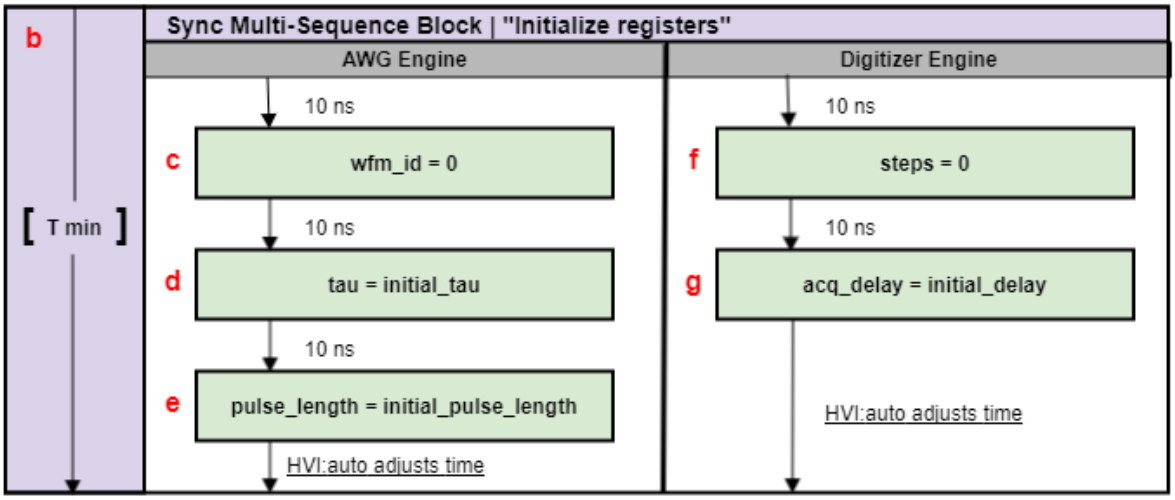
this programming example and the HVI statements used to program the sequences. Every HVI statement is described in detail later in this section, referencing with a letter the equivalent block in the HVI diagram.

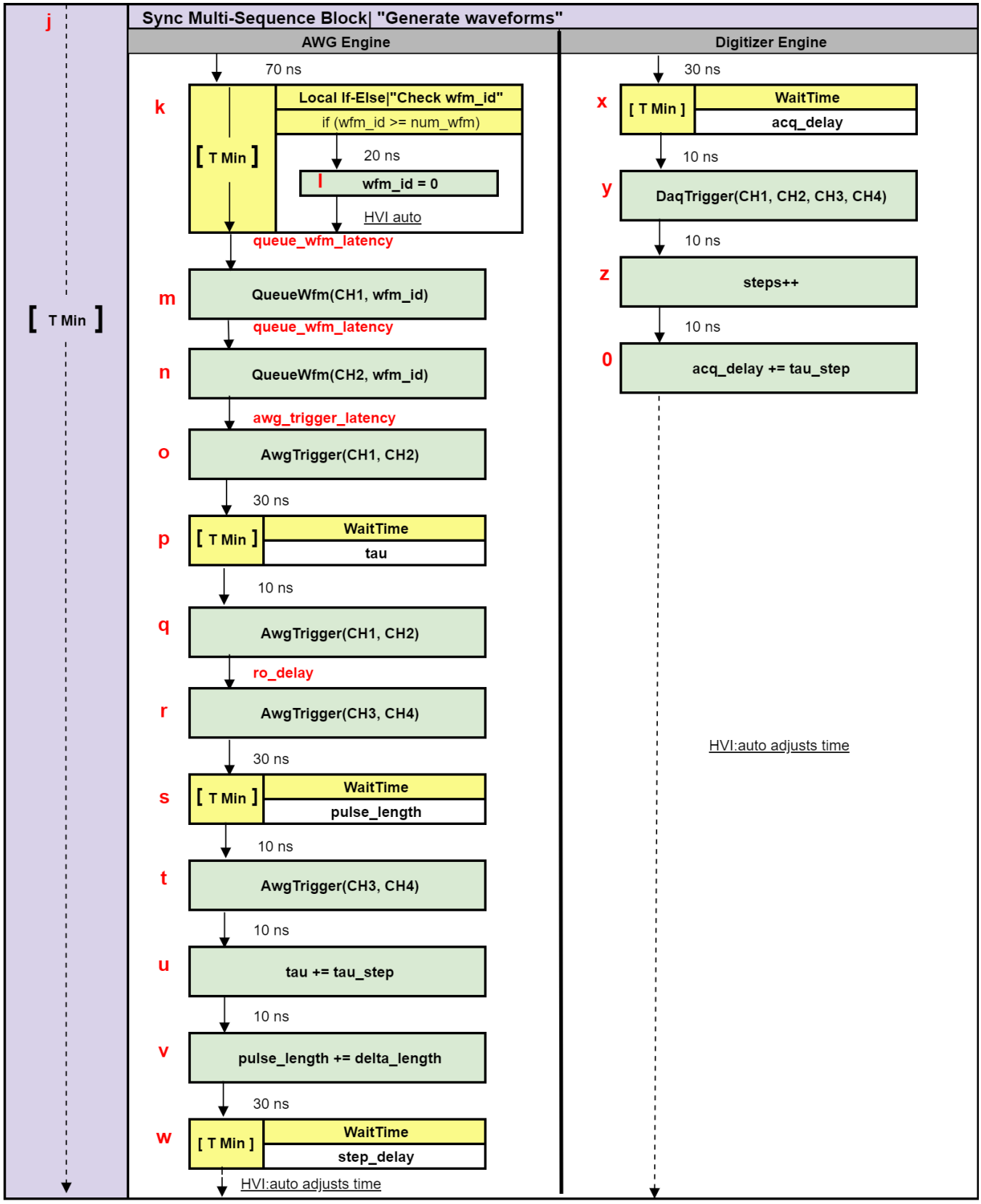
In the HVI diagram below two nested HVI Sync While loops are used to implement the experiment iteration steps and loops. The functionality to increment both the tau delay and the readout pulse length at each experiment iteration step are implemented using the HVI statement Wait Time. Delays between waveforms 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.

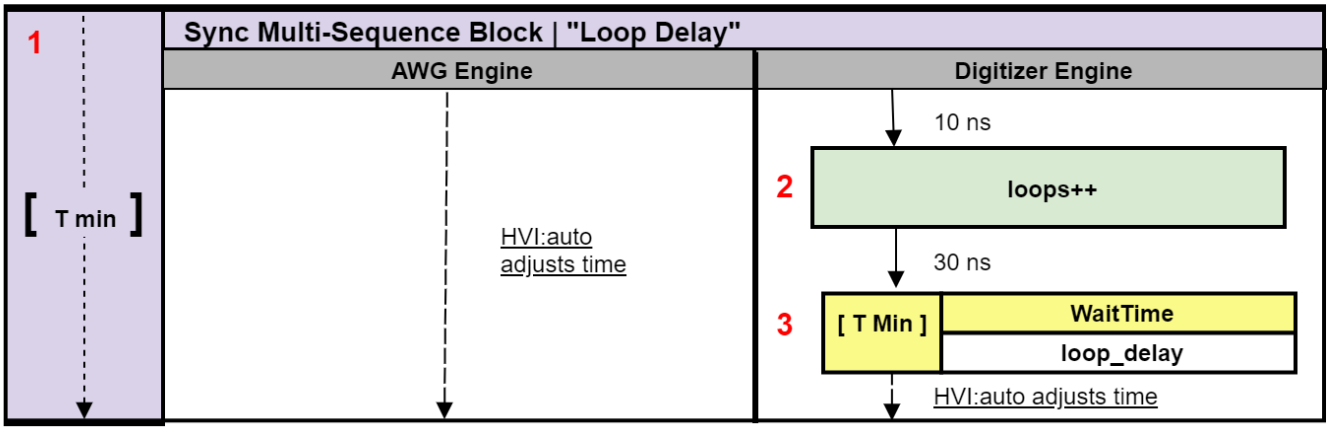
HVI instrument-specific instructions are used to queue and play the waveforms form the M3202A AWG. These instructions are represented by the green boxes labeled 'QueueWfm(...)' and 'AwgTrigger(...)' in the HVI diagram depicted below. For additional information about the M3202A AWG functionalities and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

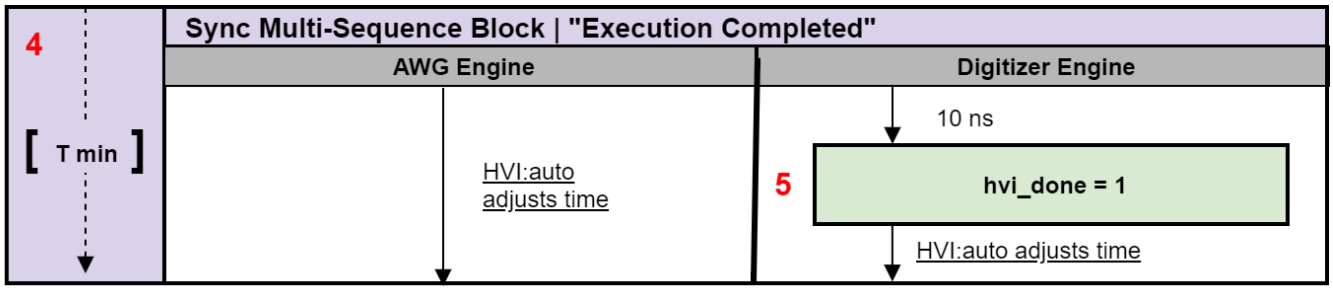


NOTE: 10 ns is the FPGA clock period for M3xxxA instruments


b
[T min]
Sync Multi-Sequence Block | "Initialize registers"
AWG Engine
Digitizer Engine
10 ns
c
wfm_id = 0
10 ns
d
tau = initial_tau
10 ns
e
pulse_length = initial_pulse_length
HVI:auto adjusts time
10 ns
f
steps = 0
10 ns
g
acq_delay = initial_delay
HVI:auto adjusts time


j
[T Min]
Sync Multi-Sequence Block| "Generate waveforms"
AWG Engine
Digitizer Engine
70 ns
k
[T Min]
Local If-Else|"Check wfm_id"
if (wfm_id >= num_wfm)
20 ns
l
wfm_id = 0
HVI auto
queue_wfm_latency
m
QueueWfm(CH1, wfm_id)
queue_wfm_latency
n
QueueWfm(CH2, wfm_id)
awg_trigger_latency
o
AwgTrigger(CH1, CH2)
30 ns
p
[T Min]
WaitTime
tau
10 ns
q
AwgTrigger(CH1, CH2)
ro_delay
r
AwgTrigger(CH3, CH4)
30 ns
s
[T Min]
WaitTime
pulse_length
10 ns
t
AwgTrigger(CH3, CH4)
10 ns
u
tau += tau_step
10 ns
v
pulse_length += delta_length
30 ns
w
[T Min]
WaitTime
step_delay
HVI:auto adjusts time
30 ns
x
[T Min]
WaitTime
acq_delay
10 ns
y
DaqTrigger(CH1, CH2, CH3, CH4)
10 ns
z
steps++
10 ns
0
acq_delay += tau_step
HVI:auto adjusts time





NOTE Python Variable *ro_delay* is used to parametrize the readout delay parameter in the experiment described in this programming example. Users can update it before execution using the *Experiment_parameters* class. The readout delay is specified using a Python Variable because it is expected to stay fixed during the HVI execution. Delays to be changed during HVI execution (e.g. tau, step_delay) are implemented using the WaitTime statement instead. 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 HVI application. 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

```
def define_hvi_resources(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.
    # Create system definition object
    sys_def = kthvi.SystemDefinition('ExperimentSetup')
    # 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)
    return sys_def
```

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. The SystemDefinition class also allows 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
    # Add chassis resources
    for chassis_number in chassis_list:
        if hardware_simulated:
            sys_def.chassis.add_with_options(chassis_number,
'Simulate=True,DriverSetup=model=M9018B,NoDriver=True')
        else:
            sys_def.chassis.add(chassis_number)
```

```
        #
    # 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.
    # To add each interconnected pair of M9031 modules use:
    # interconnects.add_M9031_modules(1stM9031_chassis_number, 1stM9031_slot_number,
2ndM9031_chassis_number, 2ndM9031_slot_number);
    # First and last chassis have only one M9031 module in the middle segment. Middle
chassis have two M9031 modules
    # in middle and lateral segments respectively. Adjacent chassis have their M9031
modules connected in diagonal.
    # See programming example  documentation for more details.
        #
    # 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]
    #
    return
```

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 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.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, module in zip(module_dict.keys(), module_dict.values()):
```

```
        sys_def.engines.add(module.instrument.hvi.engines.main_engine, engine_Name)
    #
    return
```

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_action_Names:
    # Defines the HVI action Names to be used by each HVI engine
    def __init__(self):
        self.awg_trigger = 'AWG_Trigger'
        self.daq_trigger = 'DAQ_Trigger'

def define_hvi_actions(sys_def, module_dict):
    # define_hvi_actions(hvi, module_dict):
       #
    # hvi = HVI instance
    # module_dict = dictionary containing modular instrument objects previously created
    #
    # 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 intrument's number of
channels.
       #
    # Load previously defined resources
    hvi_eng_Names = HVI_engine_Names()
    hvi_act_Names = HVI_action_Names()
       #
    # 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 == hvi_eng_Names.dig_engine:
                action_Name = hvi_act_Names.daq_trigger + str(ch_index) # arbitrary user-
defined Name
                instrument_action = 'daq{}_trigger'.format(ch_index) # Name decided by
instrument API
            else:
                action_Name = hvi_act_Names.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.instrument.hvi.actions, instrument_action)
            sys_def.engines[engine_Name].actions.add(action_id, action_Name)
```

```
    #
return
```

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 consist of two nested sync while statements that allow to implement a cycle of experiment steps nested within a number of experiment loops.

Python

```
def program_hvi_sequence(sys_def, module_dict):
    # This method programs the HVI sequence of this programming example.
    # 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.
    # Create sequencer object
    sequencer = kthvi.Sequencer('mySequencer', sys_def)
       #
    # Define registers within the scope of the outmost sync sequence
    define_registers(sequencer)
    #
    # Add and program a Sync While statement
    program_sync_while(sequencer.sync_sequence, module_dict)
       #
    # Add and program 4th Sync Multi-Sequence Block
    program_sync_block_4(sequencer.sync_sequence)
       #
    return sequencer
```

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.steps = 'Steps'
        self.loops = 'Loops'
        self.wfm_id = 'Waveform ID'
        self.tau = 'Tau'
        self.pulse_length = 'Pulse Length'
        self.acq_delay = 'Acquisition Delay'
        self.step_delay = 'Step Delay'
        self.loop_delay = 'Loop Delay'
        self.counter_register = 'Counter Register'
        self.dig_counter = 'Digitizer Counter'
        self.hvi_done = 'HVI Done'

def define_registers(sequencer):
    # Defines all registers for each HVI engine in the scope af the global sync sequence
    # Load previously defined resources
    exp_params = Experiment_parameters()
    hvi_eng_Names = HVI_engine_Names()
    register_Names = HVI_register_Names()
       #
    # Digitizer registers
    loops = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.loops, kthvi.RegisterSize.SHORT)
    loops.initial_value = 0
    steps = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.steps, kthvi.RegisterSize.SHORT)
    steps.initial_value = 0
    acq_delay = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.acq_delay, kthvi.RegisterSize.SHORT)
    acq_delay.initial_value = 0
    loop_delay = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.loop_delay, kthvi.RegisterSize.SHORT)
    loop_delay.initial_value = exp_params.loop_delay
    hvi_done = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.hvi_done, kthvi.RegisterSize.SHORT)
    hvi_done.initial_value = 0
    dig_counter = sequencer.sync_sequence.scopes[hvi_eng_Names.dig_engine].registers.add
(register_Names.dig_counter, kthvi.RegisterSize.SHORT)
    dig_counter.initial_value = 0
       #
    # AWG registers
    awg_counter = sequencer.sync_sequence.scopes[hvi_eng_Names.awg_engine].registers.add
(register_Names.awg_counter, kthvi.RegisterSize.SHORT)
    awg_counter.initial_value = 0
    tau = sequencer.sync_sequence.scopes[hvi_eng_Names.awg_engine].registers.add(register_
Names.tau, kthvi.RegisterSize.SHORT)
    tau.initial_value = 0
    wfm_id = sequencer.sync_sequence.scopes[hvi_eng_Names.awg_engine].registers.add
(register_Names.wfm_id, kthvi.RegisterSize.SHORT)
    wfm_id.initial_value = 0
    pulse_length = sequencer.sync_sequence.scopes[hvi_eng_Names.awg_engine].registers.add
```

```
(register_Names.pulse_length, kthvi.RegisterSize.SHORT)
    pulse_length.initial_value = 0
    step_delay = sequencer.sync_sequence.scopes[hvi_eng_Names.awg_engine].registers.add
(register_Names.step_delay, kthvi.RegisterSize.SHORT)
    step_delay.initial_value = exp_params.step_delay
       #
    return
```

Synchronized While

Synchronized While appears in statements (a, h). 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(loops,
kthvi.ComparisonOperator.LESS_THAN, exp_params.num_loops)
# Add Sync While Statement
sync_while = sync_sequence.add_sync_while('Run Experiment Loops', 60, sync_while_condition)
```

```
def program_sync_while(sync_sequence, module_dict):
    # Adds and programs the outmost Sync While statement of the HVI Sync Sequence
    # Load previously defined parameters and resource Names
    exp_params = Experiment_parameters()
    register_Names = HVI_register_Names()
    hvi_eng_Names = HVI_engine_Names()
    #
    # Previously defined registers
    loops = sync_sequence.scopes[hvi_eng_Names.dig_engine].registers[register_Names.loops]
       #
    # 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', 60, sync_while_
condition)
       #
    # Add and program 1st Sync Multi-Sequence Block: 'Initialize registers'
    program_sync_block_1(sync_while.sync_sequence)
       #
    # Add and program 2nd Sync While: 'Run Experiment Steps'
    program_sync_while_2(sync_while.sync_sequence, module_dict)
       #
    # Add and program 3rd Sync Multi-Sequence Block
    program_sync_block_3(sync_while.sync_sequence)
       #
    return
```

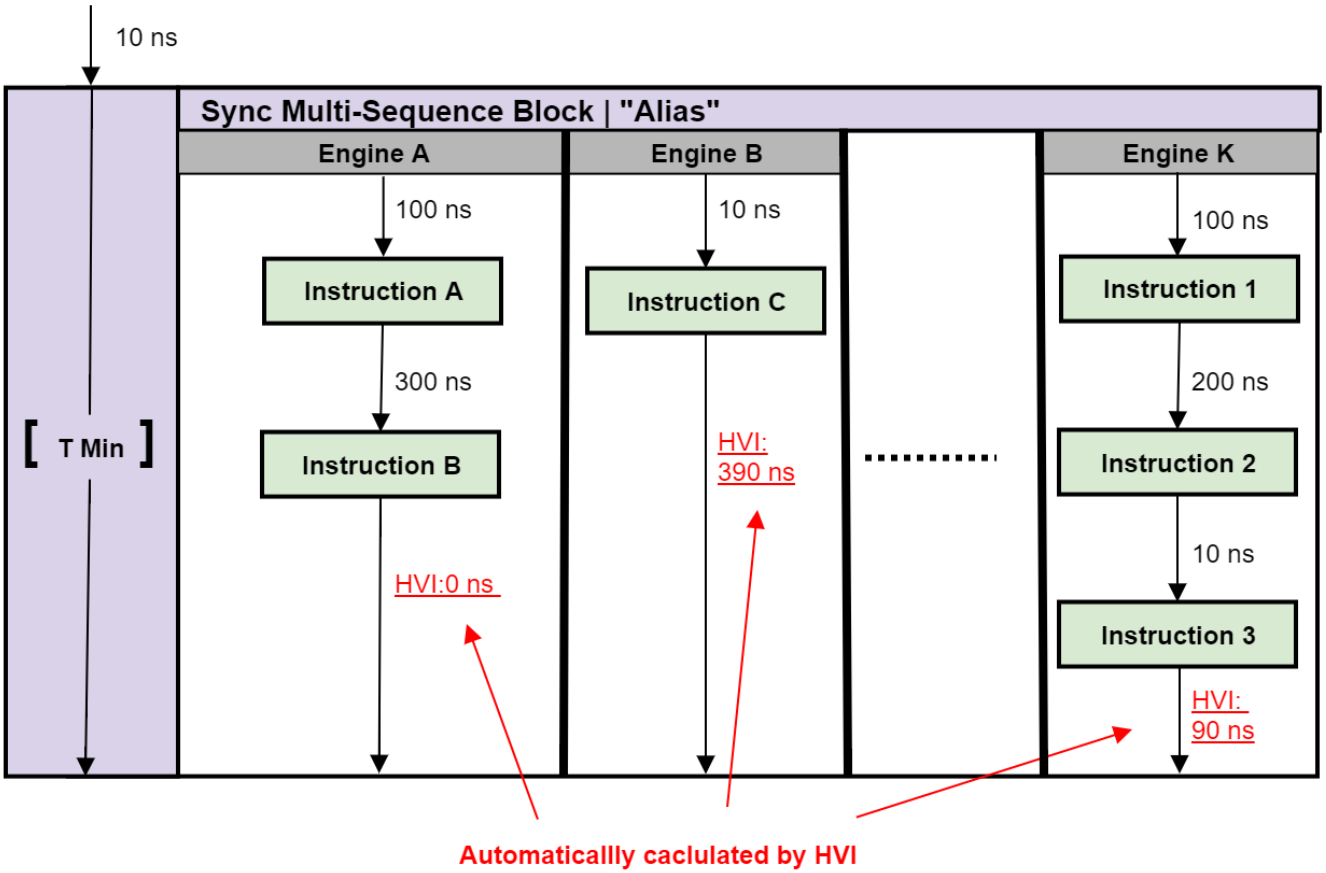
Synchronized Multi-Sequence Block

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.



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

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
exp_params = Experiment_parameters()
register_Names = HVI_register_Names()
hvi_eng_Names = HVI_engine_Names()
#
# Previously defined registers
awg_sequence = sync_block_1.sequences[hvi_eng_Names.awg_engine]
tau = awg_sequence.scope.registers[register_Names.tau]

# 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[register_Names.tau])
instruction.set_parameter(awg_sequence.instruction_set.assign.source.id, exp_
params.initial_tau)
```

Sync Register Sharing

It corresponds to statement (i) in the HVI diagram. Register sharing is a functionality defined and programmed using the *RegisterSharing* class. Register sharing allows 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 a 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[hvi_eng_Names.dig_engine].registers[register_Names.steps]
wfm_id = sync_sequence.scopes[hvi_eng_Names.awg_engine].registers[register_Names.wfm_id]



# Add sync register sharing
bits_to_share = 2
sync_while_2.sync_sequence.add_sync_register_sharing('Share steps->wfm_id', 10, steps, wfm_
id, bits_to_share)
```

IF-ELSEIF-ELSE Statement

It corresponds to statement (k) 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

```
# Load previously defined parameters and resources
exp_params = Experiment_parameters()
register_Names = HVI_register_Names()
hvi_eng_Names = HVI_engine_Names()
#
# Previously defined
wfm_id = sync_sequence.scopes[hvi_eng_Names.awg_engine].registers[register_Names.wfm_id]
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, exp_params.num_wfms)
enable_ifbranches_time_matching = True
# Add If statement
if_statement = awg_sequence.add_if('Check wfm_id', 60, 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', 20, 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

It corresponds to statements (m, n) in the HVI diagram.This statement executes a product-specific HVI instruction. 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

```
# Load previously defined parameters and resources
exp_params = Experiment_parameters()
awg_params = AWG_parameters()
register_Names = HVI_register_Names()
hvi_eng_Names = HVI_engine_Names()
#
# Previously defined
wfm_id = sync_sequence.scopes[hvi_eng_Names.awg_engine].registers[register_Names.wfm_id]
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]



# Queue waveform to CH1, CH2
# NOTE: the next instructions needs a start delay of at least 60 ns to make sure wfm_id is
updated
for awg_ch in range(1, 3):
    instrLabel = 'QueueWaveform(CH' + str(awg_ch) + ' , wfm_id)'
    instruction0 = awg_sequence.add_instruction(instrLabel, 60, 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, wfm_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, awg_params.trigger_mode)
    instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.start_
delay.id, awg_params.start_delay)
    if exp_params.T2_flag:
        instruction0.set_parameter(awg_module.hvi.instruction_set.queue_waveform.cycles.id,
awg_params.wfm_cycles)
    else:
        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,
awg_params.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 parameters and resources
awg_params = AWG_parameters()
hvi_eng_Names = HVI_engine_Names()
hvi_act_Names = HVI_action_Names()
#
# Previously defined actions to be executed within the experiment
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]
awg_trigger_12 = [
    awg_sequence.engine.actions[hvi_act_Names.awg_trigger+str(1)],
    awg_sequence.engine.actions[hvi_act_Names.awg_trigger+str(2)]]



# AWG trigger CH1, CH2 - Generates first pulse
inst_awg_trigger = awg_sequence.add_instruction('AwgTrigger(CH1, CH2)', awg_
params.awgtrigger_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, x, 3).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. Time unit is expressed as integer multiple of the instrument clock cycle duration. For example in M3xxxA PXI modules a clock cycle lasts 10 ns.

Python

```
# Load previously defined parameters and resources
exp_params = Experiment_parameters()
register_Names = HVI_register_Names()
hvi_eng_Names = HVI_engine_Names()
#
# Previously defined
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]
tau = sync_sequence.scopes[hvi_eng_Names.awg_engine].registers[register_Names.tau]



# WaitTime: tau
awg_sequence.add_wait_time('WaitTime: tau', 10, 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 be added previously to the scope of the corresponding HVI engine.

Python

```
# Load previously defined parameters and resources
exp_params = Experiment_parameters()
register_Names = HVI_register_Names()
hvi_eng_Names = HVI_engine_Names()
#
# Previously defined
awg_sequence = sync_block.sequences[hvi_eng_Names_Names.awg_engine]
tau = sync_sequence.scopes[hvi_eng_Names.awg_engine].registers[register_Names.tau]



# 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, exp_
params.tau_step)
```

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 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.