PathWave
Test Sync Executive 2023
Programming Example 7:

RF Sweeps using M320x AWGs, M5300 RF AWGs,
and M9046A Chassis
with High-Performance Clock Reference

KEYSIGHT

PROGRAMMING EXAMPLE

Table of Contents

KS2201A – Programming Example 7 – RF Sweeps using M320x AWGs M5300 RF AWGs and M9046 Chassis

In this programming example, PathWave Test Sync Executive is used to define a real-time algorithm to be executed by the FPGA (Field Programmable Gate Array) of Arbitrary Waveform Generators (AWGs) from the Keysight M3xxx and the M5xxx PXIe families. This enables the AWG channels to be used to output pulsed signals that are swept in amplitude and frequency, to perform a pulsed characterization of a Device-Under-Test. This example also shows how to use the HVI API to use the precise clock reference provided by the M9046 PXIe Chassis.

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

System Requirements

To run this series of programming examples, all the necessary pieces of SW 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) or the "Firmware Update" window of the "Utilities" menu of the SFP of M5xxx or M9xxx instruments.

The list below refers to the whole KS2201A Prog. Examples series. Please check the next section of this document for info about the exact instrument models necessary to run this programming example. You will need to install SW and FW only for the instrument models that you are using to run this example. You do not need to install KF9000B PathWave FPGA if you are not programming your instrument FPGA with a custom design.

The versions of software, FPGA firmware, drivers, and other components that were used to test this programming example are listed below. Newer versions of the SW driver or FPGA FW used to test this example are also typically expected to work. For complete details about SW and FW compatibility please visit www.keysight.com/find/ks2201a-firmware-version-requirements.

List of **tested** versions of software, Keysight instrument drivers, and FPGA firmware:

1. Software versions:
   - Python 3.9.13 64-bit, including Python packages time, numpy, matplotlib
   - Keysight KS2201A PathWave Test Sync Executive 2023 (v3.19.2)
   - Keysight KF9000B PathWave FPGA 2022 Update 1.0 (v3.7.15.0)

2. Keysight instrument driver versions:
   - Keysight IO Libraries Suite 2023 (v18.3.29324.3)
   - Keysight PXIe Chassis Family Driver v1.7.913.1
   - Keysight M9546A High-Performance Reference Clock Source Driver v1.1.282.1
   - Keysight SD1 Drivers, Libraries, and SFP v3.4.8
   - Keysight M5302A Drivers, Libraries, and SFP v1.3.51002
   - Keysight M5300A Drivers, Libraries, and SFP v1.1.51002
   - Keysight M5200A Drivers, Libraries, and SFP v1.1.51004
   - Keysight M9032A / M9033A Drivers, Libraries, and SFP v1.1.225.0

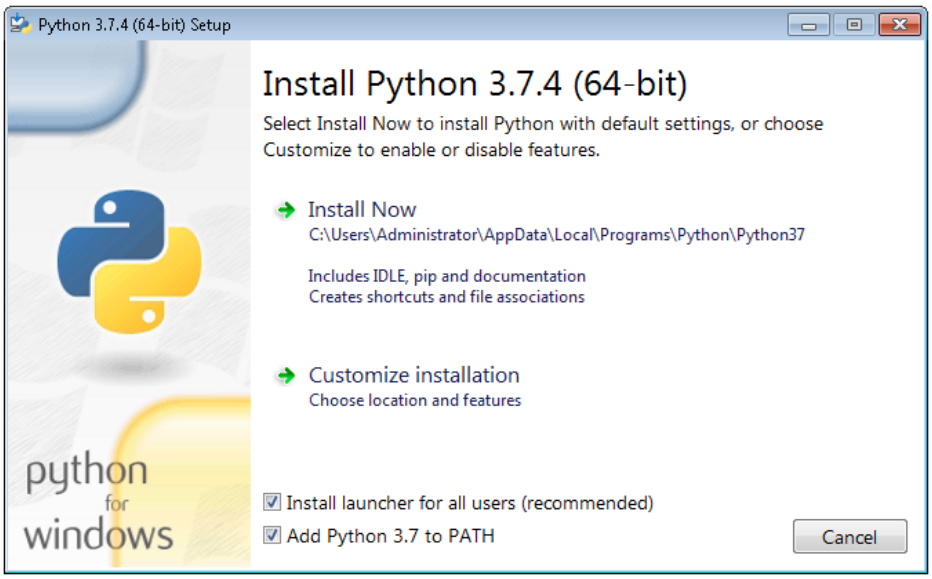
3. Keysight instrument FPGA FW versions (to be installed using Keysight instrument SFP):
   - Keysight Chassis M9019A firmware v2019EnhTrig
   - Keysight Chassis M9046A firmware v2023A
   - M3202A AWG v4.3.0
   - M3201A AWG v4.4.0
   - M3102A Digitizer v2.3.0
   - M5302A Digital I/O v5.9.42
   - M5300A RF AWG v1.1.414
   - M5200A Digitizer v1.1.409
   - M9032A System Synchronization Module v0.1.248
   - M9033A System Synchronization Module v4.1.248

NOTE The above-mentioned list of instrument drivers and firmware requirements includes all the Keysight PXIe instruments compatible with KS2201A. Please check the next section of this document for info about the exact instrument models necessary to run this programming example. Users can modify the example code to include additional compatible instruments. To run this example you need to install only the drivers of the instruments you use.

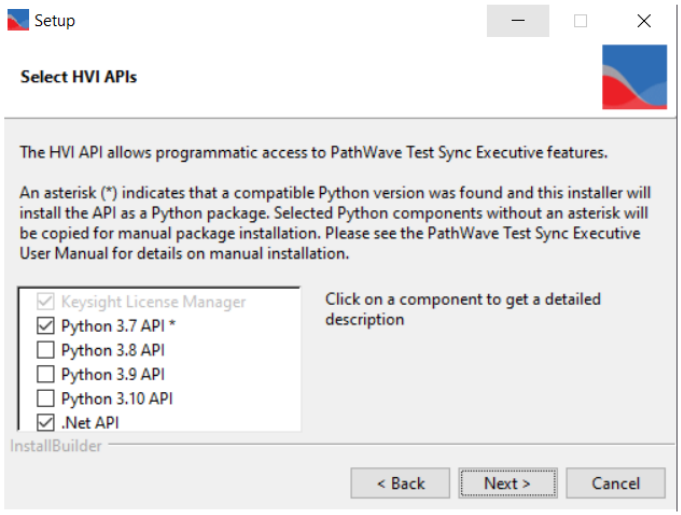
NOTE PathWave Test Sync Executive **licenses** must be installed before running the programming example Python code. To request and install a license please consult the PathWave Test Sync Executive User Manual available on www.keysight.com.

How to Install Python 3.x 64-bit

This programming example requires you to install Python 64-bit version equal to or greater than 3.7.x for all users. The Python installer can be downloaded from the Python official webpage https://www.python.org. Make sure you add Python 3.x to the PATH system Variable. This can be done at the installation step by checking the right checkboxes as shown in the screenshot below.



Once Python is installed, you can install KS2201A. When running the KS2201A installer, it will detect which Python 3.x 64-bit is installed in your system and is compatible with the `keysight_hvi` package delivered by the installer. The detected compatible version(s) will appear with a check in its checkbox. In the screenshot example below the Python 3.7 API is checked and will be installed. If you wish to install other instances of the `keysight_hvi` package, compatible with other Python 3.x 64-bit versions, then please manually check other additional checkboxes at this step of the installation procedure.

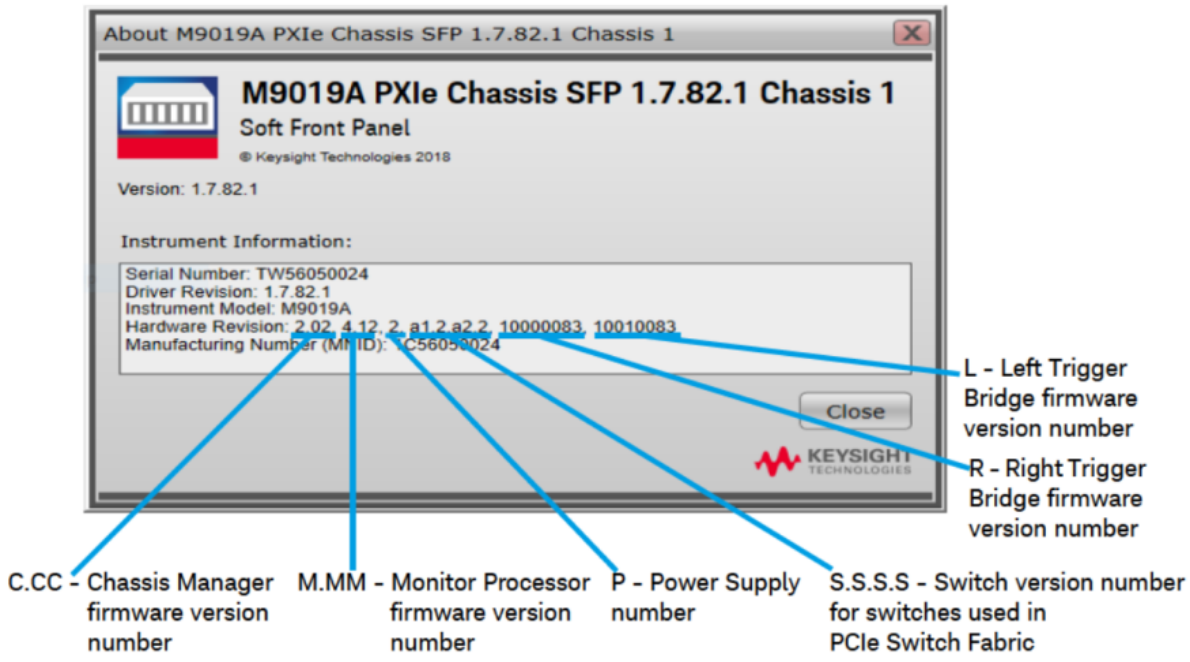


NOTE PathWave Test Sync Executive programming examples require the Python packages *time* , *numpy* and *matplotlib* . These packages can be installed using the Python package installer pip. For more information about pip and how to use it, please visit https://pypi.org/project/pip/.

NOTE Users installing Python through a distribution that is different than the one available from the Python official webpage https://www.python.org (e.g. Anaconda distribution) need to make sure that their PATH environment Variable includes the path to set up the HVI API Python library. This can be done by adding to the programming example Python code a line that includes that path, for example: sys.path.append(C:\Program Files\Keysight\PathWave Test Sync Executive <year>\api\python) where year shall be replaced with the year of the release you are using, for example <year> = 2022.

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 "Firmware Update" window found in the "Utilities" menu of the 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, Keysight Instrument Driver and FPGA Firmware

After installing your development environment (Python or C#), and installing the chassis, the next step is to install PathWave Test Sync Executive and the drivers for the Keysight instruments that you are using. After installing all the necessary software, the instrument FPGA firmware can be updated from their Software Front Panel (SFP) installed together with the instrument drivers. For more details on how to install SW and FPGA FW for Keysight instruments, please visit the instrument technical support page on www.keysight.com.

How to Install KF9000B PathWave FPGA

Some programming examples include PathWave FPGA project files designed using KF9000B PathWave FPGA. To install KF9000B and obtain a license please consult the product webpage on www.keysight.com. PathWave FPGA also requires Xilinx Vivado software to run. For further information please consult the PathWave FPGA User Manual on www.keysight.com.

Multi-Chassis System Setup using the M9032A/M9033A PXIe System Synchronization Module

In a multi-chassis system connected with Keysight PXIe System Synchronization Modules, you must include one SSM in each chassis that is part of the system. Each SSM must be inserted in the **timing slot** of your chassis. This is typically slot 10 in Keysight 18-slot chassis, but it can be a different slot number in different chassis models. The SSMs are connected to each other with System Sync cables.

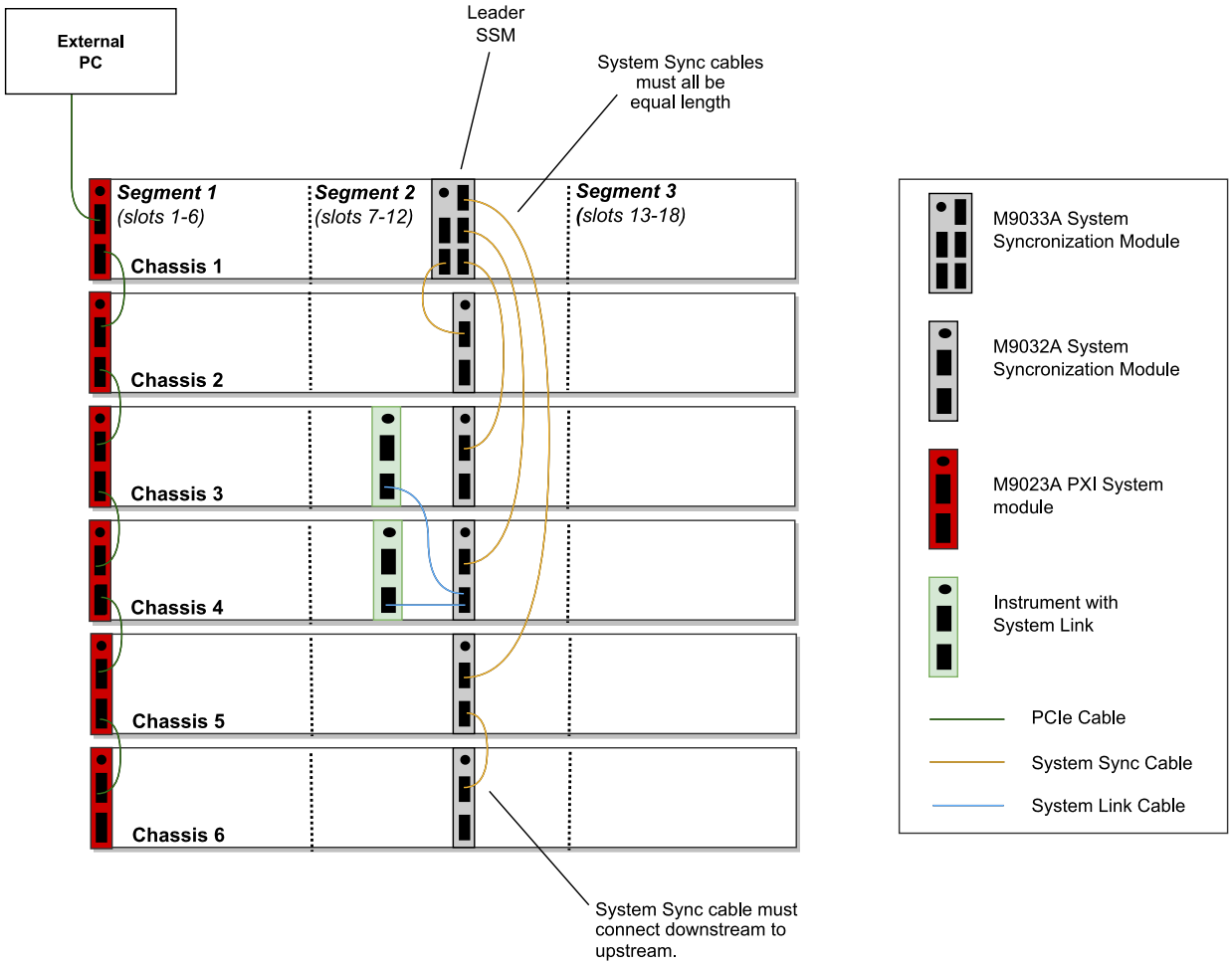
One SSM is automatically chosen as a leader and it is used to synchronize all the instruments in the multi-chassis system. The SSM chosen as leader is the SSM that has no incoming connection to its System Sync Upstream port. The leader SSM distributes a replica of the reference clock signal to the SSMs located in the other chassis. It does this through point-to-point connections between System

Sync Downstream/Upstream ports. In the example multi-chassis system shown in the following diagram, the leader SSM is in Chassis 1.

A multi-chassis PXIe system may be configured to use many different reference options. For a list of those options and descriptions of how to configure them, see the section *Clocking* in this document. For one of those reference options, an SSM is chosen as a leader and uses its internal Oven Controlled Crystal Oscillator (OCXO) clock to synchronize all the instruments included in the multi-chassis system.

NOTE A Multi-chassis system based on the older M9031A modules required an external reference clock generator to distribute the precise common 10 MHz reference clock signal across different chassis. In the multi-chassis topology delivered by PathWave Test Sync Executive, the SSM assumes the function of the **reference clock signal generator/distributor**, by sharing a reference clock generated by an internal PLL. This PLL can be fed by different sources (as explained later in this document) including the OCXO inside the SSM, which generates a 10 MHz sine wave. An external 10 or 100 MHz reference signal can still be connected to the SSM SClk / Ref Input port, to sync it together with other clusters.

The following diagram shows an example of a 6 chassis system connected with SSMs. In this system an M9033A SSM in chassis 1 distributes the reference clock to four M9032A SSMs located in each of the other chassis. The SSM in chassis 5 also forwards the clock to a sixth chassis.



For further information please refer to the KS2201A System Setup Guide available on www.keysight.com/find/KS2201A-downloads.

Programming Example Overview

This programming example illustrates how to deploy the Hard Virtual Instrument (HVI) to synchronously generate Radio Frequency (RF) pulsed signals from multiple M320x Arbitrary Waveform Generators (AWGs) and M5300 RF AWGs. Two nested Sync While loops are used to sweep both the amplitude and frequency of the RF pulses. The pulsed RF sweeps can be used for applications such as spectroscopy of quantum bits or characterization of 5G components.

Users can set the AWG and RF AWG settings, and RF sweep parameters using the Variables defined within the ApplicationConfig class shown below.

```
"""
AWG settings
"""
self.awg_channel = 2
self.freq_start_value = 5e6 # [Hz]
self.freq_step = 20e6 # [Hz]
self.off_value = 0 # [V]
self.ampl_start_value = 0.5 # [V]
self.ampl_step = 0.25 # [V]
self.pulse_duration = 200 # [ns]
"""
RF AWG settings
"""
self.M5xxx_fpga_clock_period = 10/3 # [ns]
self.rf_awg_sample_rate = 4.8 # [GSa/s]
self.waveform_file = "Waveforms/DC_m0p5_I_2048.csv"
self.rf_awg_carrier_freq = 25.0e6
self.rf_awg_pulse_duration = 200 # [ns]
self.rf_awg_channel = 1 # channel playing wfm
self.rf_awg_start_delay = 0
self.rf_awg_num_cycles = 0 # use 0 for infinite cycles
self.rf_awg_ampl_start_value = 30 # [%]
self.rf_awg_ampl_step = 30 # [%]
self.rf_awg_amplitude = 100 # [%]
self.set_amplitude_latency = 100 # [ns]
# Trigger settings
self.rf_awg_trigger_mode = ktm5300.TriggerMode.SW_HVI_PER_CYCLE_TRIG
"""
RF sweep parameters
"""
self.num_freqs = 2
self.num_amplitudes = 3
self.ampl_sweep_duration = 1000 # [nS]
```

The HVI functionalities to implement the required application are:

1. Off-shelf inter-instrument synchronization capabilities
2. Scalability to an arbitrary number of instruments. In this specific case M32xx PXIe AWGs and M5300 PXIe RF AWGs
3. System initialization procedure for M5300 RF AWGs
4. Mapping of physical quantities (amplitude, frequency, phase, etc.) into HVI registers
5. Chassis interconnections and synchronization using the System Sync Module (SSM)
6. Usage of M9046A Chassis with High-Performance Reference Clock Source (HPRCS)

More details about the HVI statements used in the real-time algorithm can be found in the section "HVI Application Programming Interface (API): Detailed Explanations".

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
```

Afterwards, it is necessary to specify the actual chassis number and slot number where the real PXI instruments are located. You must update the model numbers of the PXI instruments if they are different from the instrument models used in this programming example. This example uses PXI instruments from the Keysight M3xxx and M5xxx family.

The first step to control these instruments is to create an object using the instrument API. For a complete description of the instrument object creation 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 (or resource ID), and options.

This programming example can be deployed on an arbitrary number of instruments to be defined using the module-descriptor class. All instruments included in the Python code execute the synchronized real-time operations defined by the HVI instance. Please update the properties in each module-descriptor object before running the programming example:

```
# Define module descriptors below with your instruments information
self.module_descriptors = [
    ModuleDescriptor('M3202A', 1, 4, self.sd1_options, self.awg_engine_name),
    ModuleDescriptor('M3201A', 2, 2, self.sd1_options, self.awg_engine_name),
    ModuleDescriptor('M5300', 2, 12, self.rf_awg_instrument_options, self.rf_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 be specified and listed by chassis number:

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

In the case of a multi-chassis setup, define each System Sync Module and its connections:

```
# Multi-chassis setup
# Define the System Sync Modules included in your system.
self.ssm_options = ''
self.ssm_simulation_options = 'Simulate=true,DriverSetup=Model=M9033A'
self.system_sync_modules_descriptors = [
    SystemSyncModuleDescriptor('PXI0::CHASSIS1::SLOT10::INDEX0::INSTR', self.ssm_options),
    SystemSyncModuleDescriptor('PXI0::CHASSIS2::SLOT10::INDEX0::INSTR', self.ssm_options)]
# For each SSM define which SSM is connected to its downstream connectors.
# Each connectivity item is a triple (ssm1_chassis, ssm1_downstream_connector_number, ssm2_
chassis)
self.ssm_connections = [
    SystemSyncModuleConnection(ssm1_chassis=1, ssm1_downstream_connector_number=1, ssm2_
chassis=2)]
```

Please note that in every programming example, PXI trigger resources need to be reserved so that the HVI instance can use them for their execution. PXI lines to be assigned as trigger resources to HVI can be selected by updating the code snippet below:

```
# Assign triggers to HVI object to be used for HVI-managed synchronization, data sharing, etc #
NOTE: In a multi-chassis setup ALL the PXI lines listed below need to be shared among each M9031
board pair by means of SMB cable connections
self.pxi_sync_trigger_resources = [ kthvi.TriggerResourceId.PXI_TRIGGER0,
kthvi.TriggerResourceId.PXI_TRIGGER1, kthvi.TriggerResourceId.PXI_TRIGGER2,
kthvi.TriggerResourceId.PXI_TRIGGER3]
```

PXI lines allocated to be used as HVI trigger resources cannot be used by the programming example for other purposes. The vector pxi_sync_trigger_resources specified above must include at least the necessary number of PXI lines for the programming example to execute. Please check the programming example code for the actual number of PXI lines that needs to be reserved. The HVI compiler also returns, for a given HVI sequence, the number of necessary PXI lines that must be reserved.

How to Configure the Reference Clock

This programming example illustrates all the possible options to setup the reference clock in your measurement system. For further information please refer to the *KS2201 System Setup Guide* available on www.keysight.com/find/KS2201A-downloads.

The *RefClockConfig* class in the example's code contains all the constants that can be used to configure the reference clock:

```
class RefClockConfig:
    """
    Class for configuring the Ref. Clock source and modes
    """
    def __init__(self):
        # Reference clock source
        self.SOURCE_CHASSIS_REF = 0 # select this mode to use a M904x chassis internal reference
as a ref. clock source
        self.SOURCE_SYSTEM_SYNC_MODULE = 1 # select this mode to use a System Sync Module (SSM)
as a ref. clock source
        self.SOURCE_CHASSIS_HPRCS = 2 # select this mode to use the chassis High Performance
Reference Clock Source (HPRCS) as a ref. clock source
        self.SINGLE_CHSSSIS_NO_SOURCE = 3 # select this mode to use the Chassis BackPlane (BP)
in a 1-chassis system with no SSM or HPRCS
        # Internal/External clock modes
        self.MODE_INTERNAL = 0 # The ref. clock source is not locked to any external clock
source
        self.MODE_EXTERNAL = 1 # The ref. clock source is locked to an external clock source
```

These constants can be used to define the reference clock configuration inside the *ApplicationConfig* class defined at the beginning of the example's Python code:

```
"""
Define reference clock configuration
"""
# For detailed info about all the possible configurations please read the System Setup Guide on
www.keysight.com.
# Create Ref. Clock configuration object
self.ref_clock_config = RefClockConfig()
# Choose the constant that reflects the ref. clock source that you are using
self.ref_clock_source = self.ref_clock_config.SOURCE_CHASSIS_HPRCS
# Specify the chassis number containing the ref clock source to be used
self.ref_source_chassis_number = 2
# Choose the ref. clock mode that you are using
self.ref_clock_mode = self.ref_clock_config.MODE_INTERNAL
# Specify the freq. of the signal coming form the external clock source (if connected)
self.external_ref_frequency = 100e6 # [Hz]
# For M5300 RF AWGs, specify if you want to connect and use the Chassis 2.4 GHz Analog Reference
self.analog_ref_mode = self.ref_clock_config.MODE_EXTERNAL
```

To sum up, the reference clock configuration consists in defining two components:

1. The reference clock source
2. The reference clock mode

The reference clock signal is always a 50% duty-cycle clock signal with a frequency of 100 MHz, generated by a Phase Locked Loop (PLL) inside the System Sync Module or inside the PXIe chassis. Here, with reference clock source, we refer to the source signal driving such PLL. The characteristics

of such source signal determine the purity of the generated 100 MHz clock and ultimately impact the performance of your measurement results. Detailed explanations are provided in the ***KS2201A System Setup Guide*** available on www.keysight.com/find/KS2201A-downloads. Keysight PXIe chassis systems supported by PathWave Test Sync Executive 2022 (or later) allow the following possible reference clock sources:

1. M9032A or M9033A PXIe System Sync Module (SSM): in this case the OCXO (Oven-Controlled Xtal Oscillator) inside the SSM is used as a reference clock source
2. High Performance Reference Clock Source output featured in the Keysight M9046A PXIe Chassis
3. Internal clock reference of a Keysight M9044A or M9046A PXIe Chassis
4. 100 MHz signal taken from the chassis backplane

To configure each of the possible sources and modes, please refer to the ***KS2201A System Setup Guide*** for anything concerning the HW connections to be made. On top of HW connections, the corresponding source and mode configuration must be set in the Python code by setting the *ref_clock_source* and *ref_clock_mode* Python Variables to the corresponding constants defined in the *RefClockConfig* class. The chassis number where the reference clock source is located must be also specified, together with the frequency of the external reference (if external mode is configured).

In the case of a single-chassis system with no SSM, HPRCS or internal chassis clock reference (for example, because the chassis model is different than M904x), the 100 MHz clock reference from the chassis backplane is used. For such case, please select *ref_clock_source = self.ref_clock_config.SINGLE_CHASSIS_NO_SOURCE*.

How to Initialize your System

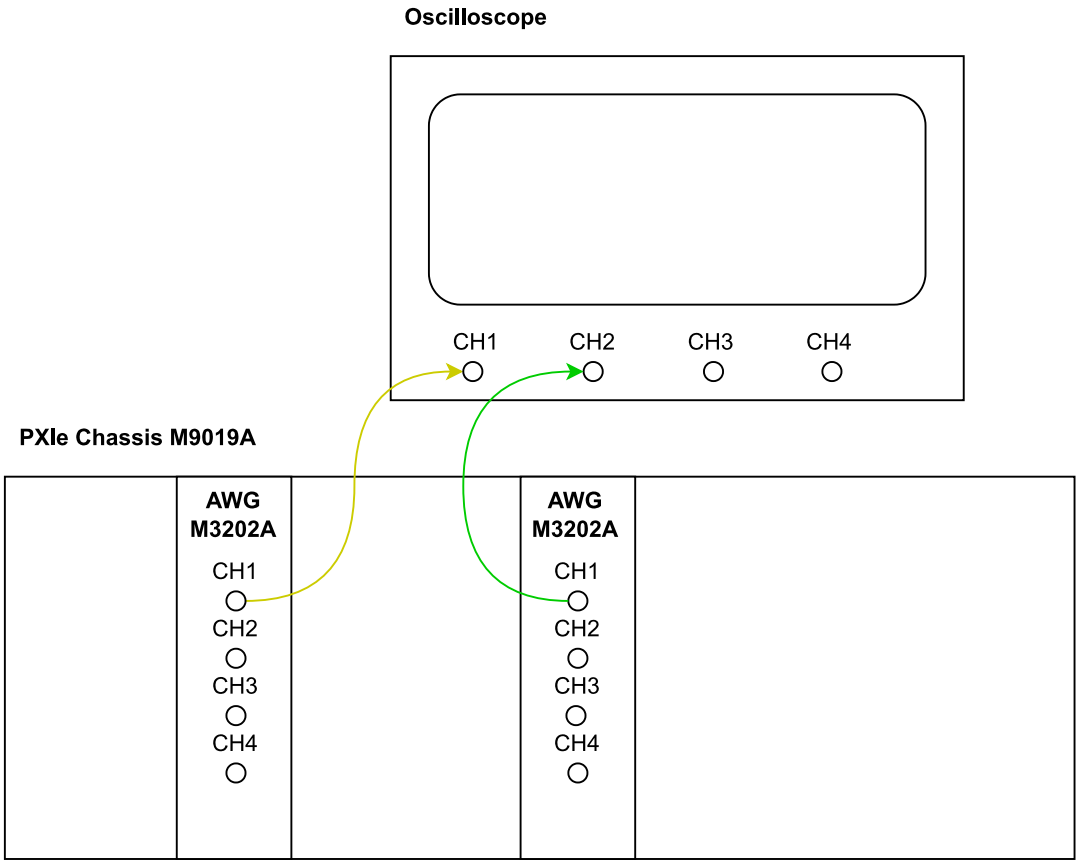
If you use this example code with M5300 PXIe RF AWGs, then you need to be aware of the dedicated initialization procedure required by this type of instrument due to the specific calibration data it requires. Further information about the M5300 calibration procedures can be found in the M5300 RF AWG User's Manual . The detailed necessary system initialization steps for various use case scenarios are described in the KS2201A System Setup Guide and in the "System Initialization" section of the KS2201A PathWave Test Sync Executive User Manual, both available at www.keysight.com/find/KS2201A-downloads.

The SystemInitializationConfig class in the example's code contains all the constants that can be used to perform any of the required system initialization steps:

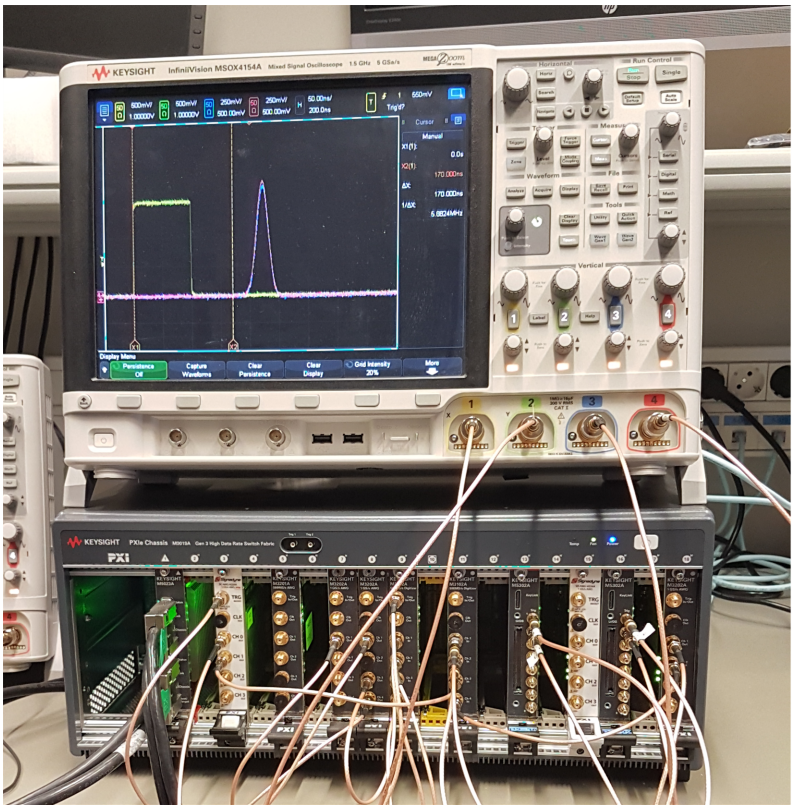
```
class SystemInitializationConfig:
    """
    Class for configuring the System Initialization. After putting the setup together the system
initialization steps are:
    1. PRE_CALIBRATION
    2. Wait for the instruments to warmup
    3. RESET_CALIBRATION (necessary only to obtain channel skew below 50ps)
    4. MEAS_OPERATION
    Once the setup is warmed up and running the measurement operation mode (MEAS_OPERATION) can
be used.
    Consult the M5300 RF AWG User Manual for more details about its initialization process.
    """
    def __init__(self):
        # System Initialization Modes
        self.PRE_CALIBRATION = 0
        self.RESET_CALIBRATION = 1
        self.MEAS_OPERATION = 2
```

Measurement Results

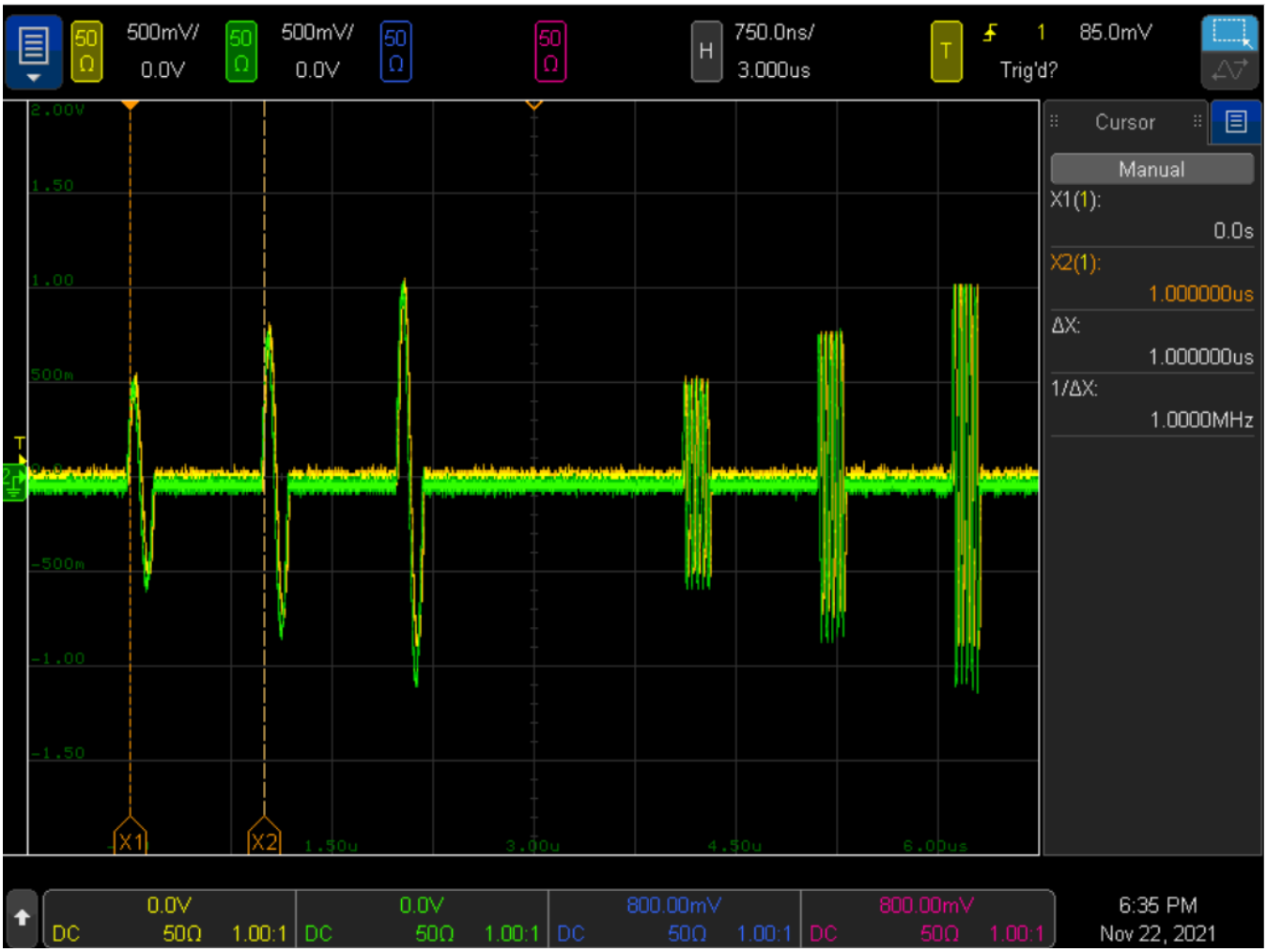
The programming example capabilities are illustrated through some example measurement results obtained using the measurement setup depicted below where the first channels (CH1) of two M3202A AWGs are connected to two channels of a Keysight Oscilloscope.



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

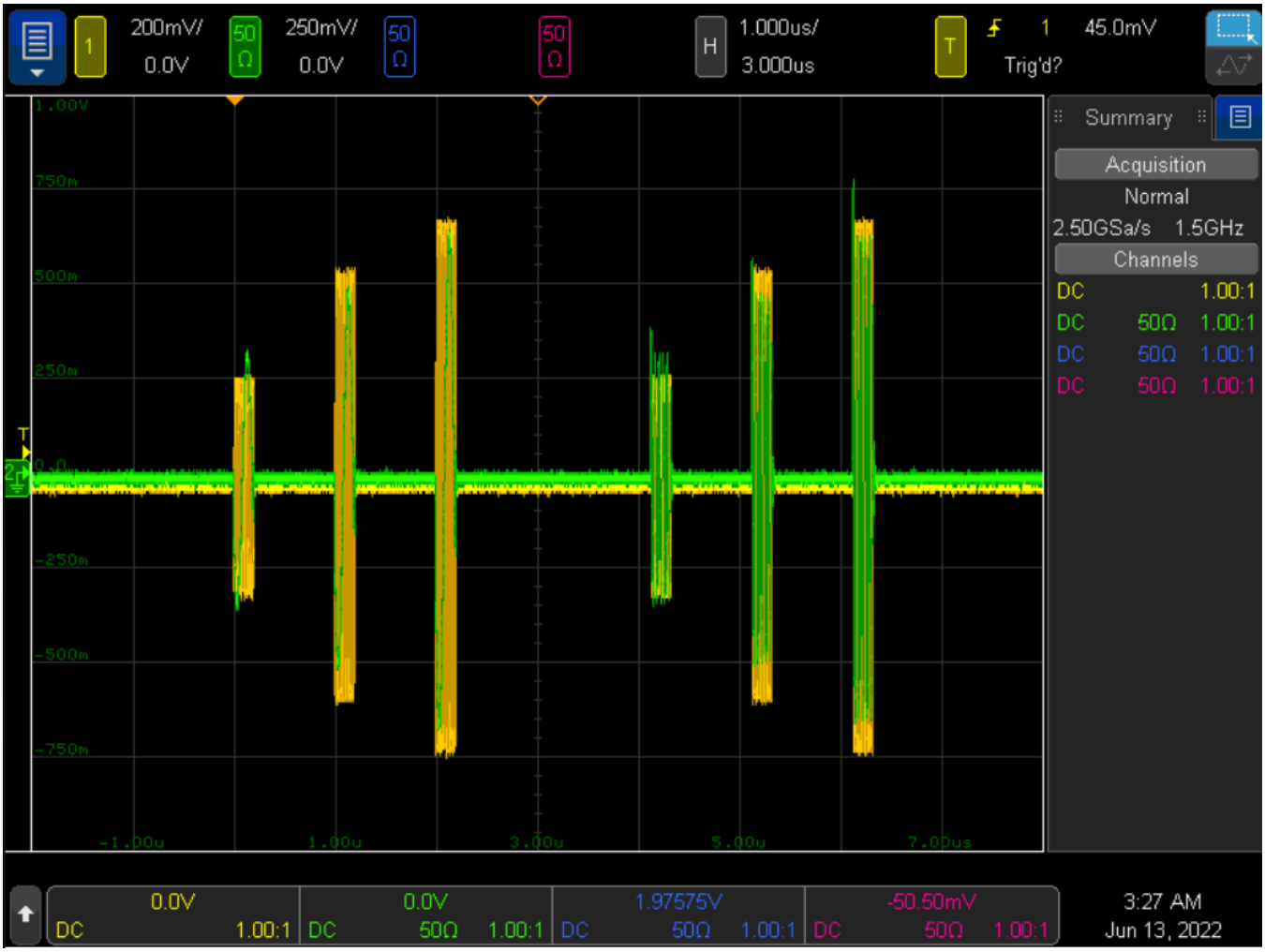


The two nested loops programmed using HVI are used in this programming example to implement two nested RF sweeps. The outmost sweep changes the AWG channel frequency, whereas the inner sweep changes the channel amplitude. The scope screenshot below shows an example measurement taken with a sweep of three amplitude values (from 0.5 to 1 V) and two frequency values (from 5 to 25 MHz). The waveform displayed in yellow and green are measured at the CH1 of two M3202A AWGs controlled by HVI. More details on the HVI sequenced programmed to implement the RF sweeps are described in the next section of this document.



By executing a channel phase reset from HVI, we can ensure that all AWGs included in the application are precisely synchronized and in-phase. In this example measurement the channel phase is also set to start from 0 degrees, however any other arbitrary value can be set. The duration of each RF pulse is set in the Python code to 200 ns. The duration of the amplitude sweep is set to 1 us, which is the time elapsed between two subsequent RF pulses with the same frequency value.

If we change the cabled connection and connect an M5300 RF AWG to CH1 of the oscilloscope, the measurement result shown below is obtained. M5300 RF AWGs do not allow to sweep the frequency real-time, hence only an amplitude sweep is performed.



An example of the console output obtained when executing this programming example is shown in the screenshot below. At the end of the execution the final values of the HVI registers use to sweep frequency and amplitude can be read. Please note that the values read by SW will be incremented by an additional step amount with respect to the last values set in the RF sweeps. This is because of the order of operation implemented to optimize the real-time algorithm. The HVI registers have integer values, therefore to convert the values to double precision values representing quantities in Volts or Hz, the *voltsToInt()* and *freqToInt()* SW methods of the SD1 API are used.

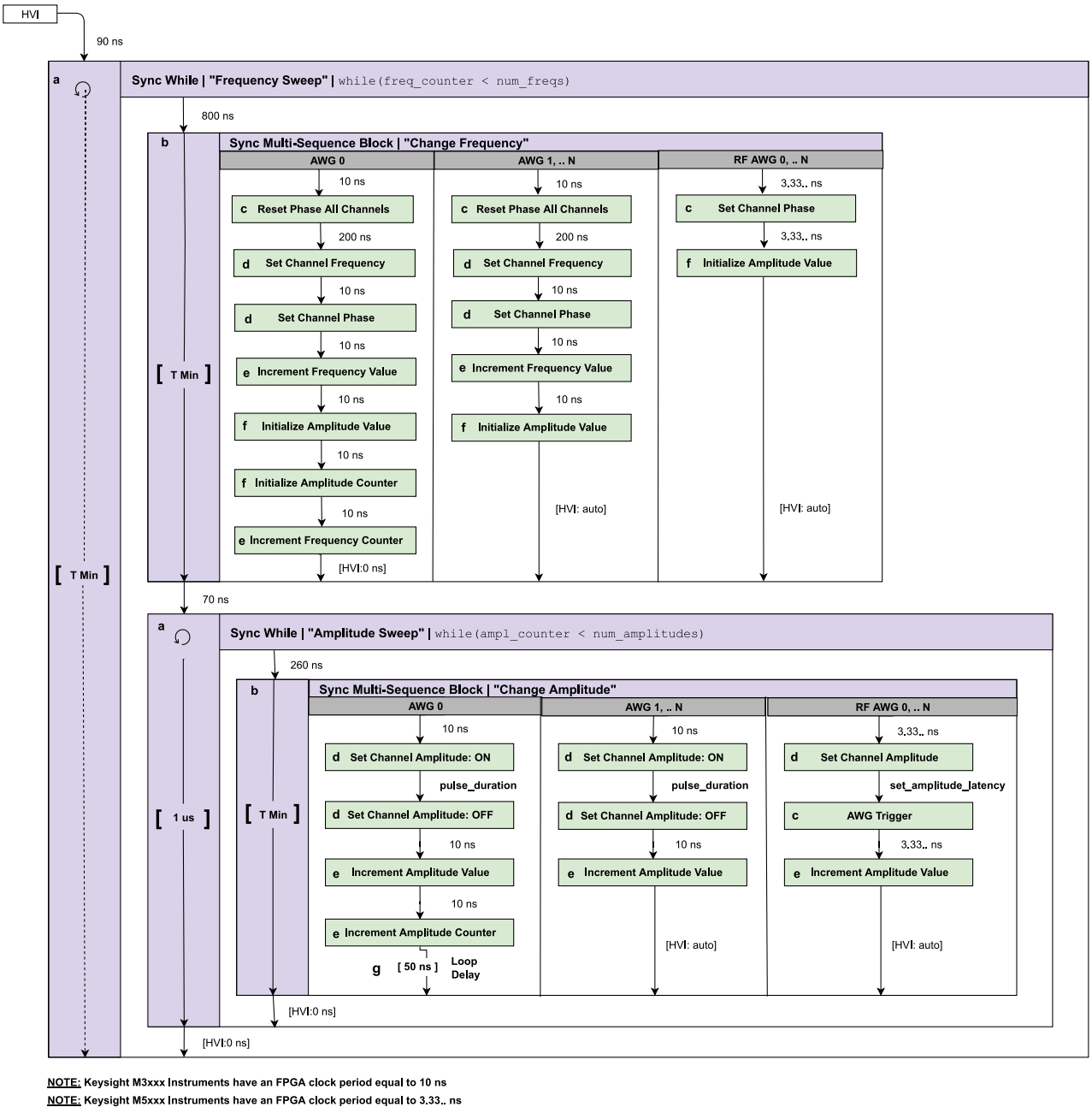
```
OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS

--------------------------
RF Sweeps using M3xxxA AWGs
--------------------------
Frequency Sweep: Start 5.0 MHz, Step 20.0 MHz, 2 steps, Stop 25.0 MHz
Amplitude Sweep: Start Ampl. 0.5 V, Step 0.25 V, 3 steps, Stop 1.0 V

------------------
System Definition
------------------
The PathWave Test Sync Executive API installed on your system has an HVI core version 1.14.2

HW Instruments:
- Model: M3201A in chassis: 1, slot: 13, HVI Engine Name: AwgEngine0, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
- Model: M3202A in chassis: 1, slot: 14, HVI Engine Name: AwgEngine1, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
- Model: M3202A in chassis: 1, slot: 15, HVI Engine Name: AwgEngine2, HVI core version: 1.14.2, HVI Engine FPGA IP version: 1.6.0
System Sync Modules:
- System Sync Module in chassis: 1, slot: 10 with 1 Upstream Ports and 4 Downstream Ports
----------------------
Program HVI Sequences
----------------------
Initializing the defined system...
Programming the HVI sequences...
Programmed HVI sequences exported to file
------------------
Execute HVI
------------------
Compiling HVI...

HVI Compiled
This HVI needs to reserve 2 PXI trigger resources to execute
HVI Loaded to HW
HVI Running...
Final Register Values: Amplitude 1.25 V, Frequency 44.99999999999645 MHz
Press enter to run HVI again, q to exit...
q
Releasing HW...
PXI modules closed
```

HVI Application Programming Interface (API): Detailed Explanations

PathWave Test Sync Executive implements the next generation of HVI technology and delivers the HVI Application Programming Interface (API). This section explains how to implement the use case of this programming example using HVI API. The sequence of operations executed by each of the instruments using HVI technology is explained in the diagram below. The diagram depicts the HVI sequences executed within this programming example and the HVI statements used to program the sequences. Every HVI statement is described in detail later in this section, referencing with a letter the equivalent block in the HVI diagram and explaining in detail the corresponding HVI API code block and the HVI functionalities that it implements.

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

NOTE The duration of each iteration of the Sync While loop used in this example is set to an arbitrary value using the Duration property of the SyncWhile object. The default duration of each sync statement is set to "T Min", which corresponds to the minimum duration to comply with the start delays specified by the user for each statement programmed into the local sequences contained in it.



NOTE: Keysight M3xxx Instruments have an FPGA clock period equal to 10 ns
NOTE: Keysight M5xxx Instruments have an FPGA clock period equal to 3.33.. ns

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 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 HW platform: chassis, interconnections, PXI trigger resources, synchronization, HVI
clocks
    define_hw_platform(sys_def, config)
    # Define ref. clock configuration
    define_clocking(sys_def, config)
```

Define Platform Resources: Chassis, PXI triggers, Synchronization

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

```
def define_hw_platform(sys_def, config):
    """
    Define HW platform: chassis, interconnections, PXI trigger resources, synchronization, HVI
clocks
    """
    # Define chassis resources
    # For multi-chassis setup details see programming example documentation
    for chassis_number in config.chassis_list:
        if config.hardware_simulated:
# This simulation options require to install the chassis driver:
# sys_def.chassis.add_with_options(chassis_number, 'Simulate=True,DriverSetup=Model=M9019A')
# As an alternative, the GenericPxieChassis allows to run simulations without installing the
chassis driver
            sys_def.chassis.add(chassis_number,
'Simulate=True,DriverSetup=Model=GenericPxieChassis')
        else:
            sys_def.chassis.add(chassis_number)

# Define System Sync Modules (SSMs)
    if config.system_sync_modules_descriptors:
        interconnects = sys_def.interconnects
        ssm_list = []
        for descriptor in config.system_sync_modules_descriptors:
            if config.hardware_simulated:
                ssm = interconnects.add_sync_module(descriptor.resource_id, config.ssm_
simulation_options)
            else:
                ssm = interconnects.add_sync_module(descriptor.resource_id, descriptor.options)
            ssm_list.append(ssm)
```

```
# Define connections between SSMs
if config.ssm_connections:
    for connection in config.ssm_connections:
        connector_number = connection.ssm1_downstream_connector_number
        for ssm in ssm_list:
            if ssm.chassis == connection.ssm1_chassis:
                ssm1 = ssm
            if ssm.chassis == connection.ssm2_chassis:
                ssm2 = ssm
        # Implement each user-defined connection
        try:
            # Set connection. SSMs have always one upstream port
            ssm1.connectivity.systemsync_downstream[connector_number].set_connection
(ssm2.connectivity.systemsync_upstream[1])
        except:
            exit("Exception! Please check the valued defined for SyncModule resource ids,
chassis numbers and connections")

    # Assign the defined PXI trigger resources
    sys_def.sync_resources = config.pxi_sync_trigger_resources
    # Assign clock frequencies that are outside the set of the clock frequencies of each HVI
engine
    # Use the code line below if you want the application to be in sync with the 10 MHz clock
    sys_def.non_hvi_core_clocks = [10e6]
```

Define Reference Clock Configuration

The reference clock source and mode defined in the *ApplicationConfig* class are used to configure the system reference clock by using the *clocking* interface of the *SystemDefinition* object. The possible clock sources and modes are described in the previous section titled "How to Configure the Reference Clock". The Python code implementing all the configuration API calls in contained in the *define_clocking* function reported below.

```
def define_clocking(sys_def, config):
    """
    Define reference clock source, mode and settings.
    For detailed info about all the possible configurations please read the System Setup Guide
on www.keysight.com.
    """
    # Define the Reference Clock Source
    # See System Setup Guide for detailed HW connections necessary for each configuration
    if config.ref_clock_source == config.ref_clock_config.SINGLE_CHSSSIS_NO_SOURCE:
        # Chassis BackPlane (BP) is the default clock source used in a 1-chassis system that has
no SSM, HPCR or internal reference
        ref_chassis = sys_def.chassis[config.ref_source_chassis_number]
        sys_def.clocking.reference_source = ref_chassis.clock_source
    elif config.ref_clock_source == config.ref_clock_config.SOURCE_CHASSIS_HPRCS:
        # Configures High Performance Reference Clock Source (HPRCS) as a source
        # It requires to connect the chassis RF Out to the SSM REF In port
        ref_chassis = sys_def.chassis[config.ref_source_chassis_number]
        hprc_source = ref_chassis.high_performance_clock_source
```

```
        sys_def.clocking.reference_source = hprc_source
    elif config.ref_clock_source == config.ref_clock_config.SOURCE_CHASSIS_REF:
        # Configures Chassis RF Analog Reference as a source. Only for M904x Chassis models
        ref_chassis = sys_def.chassis[config.ref_source_chassis_number]
        sys_def.clocking.reference_source = ref_chassis.clock_source
    elif config.ref_clock_source == config.ref_clock_config.SOURCE_SYSTEM_SYNC_MODULE:
        # Configures System Sync Module as a source
        if sys_def.interconnects:
            for index in range(sys_def.interconnects.count):
                if sys_def.interconnects[index].chassis == config.ref_source_chassis_number:
                    ssm_source = sys_def.interconnects[index]
            sys_def.clocking.reference_source = ssm_source.clock_source
        else:
            print("WARNING: Ref. Clock Source set to System Sync Module but no Sync Modules have
been defined in your code! ")
            print("System clocking set to the default configuration SINGLE_CHSSSIS_NO_SOURCE")
    # Define the Ref. Clock Mode
    if config.ref_clock_mode == config.ref_clock_config.MODE_EXTERNAL:
        # Set up the Ext. Ref. source (if connected)
        sys_def.clocking.reference_source.set_mode(kthvi.ClockingReferenceMode.EXTERNAL,
config.external_ref_frequency)
    else:
        # Default internal mode
        sys_def.clocking.reference_source.set_mode(kthvi.ClockingReferenceMode.INTERNAL)
    # Activate M904x Chassis Analog RF Outputs if the user connected them to M5xxx modules
    if config.analog_ref_mode == config.ref_clock_config.MODE_EXTERNAL:
        # Activate 2.4 GHz FP output for the M904x Chassis used as a reference
        clock_output_2_4GHz = ref_chassis.clock_outputs["FP2.4GHzOut"]
        clock_output_2_4GHz.set_enabled(True)
        # Control M5xxx instruments to use an external 2.4 GHz reference
        sys_def.clocking.enable_external_analog_clocks([2.4e9])
```

Define HVI Engines

All HVI Engines to be included in the HVI instance need to be registered into the *EngineCollection* class instance. Each HVI Engine object added to the engine collection contains collections of its own that allow you to access the actions, events and triggers that each specific engine will control and use within the HVI.

Python

```
# HVI engines
self.rf_awg_engine_name = "RfAwgEngine"
self.engine_name = "AwgEngine"
self.leader_engine_name = ""

def define_hvi_engines(sys_def, module_dict):
    """
    Define all the HVI engines to be included in the HVI
```

```
    """
    # For each instrument to be used in the HVI application add its HVI Engine to the HVI Engine
Collection
    for engine_name in module_dict.keys():
        sys_def.engines.add(module_dict[engine_name].hvi.engines.main_engine, engine_name)
```

Define HVI Actions

In this programming example the Channel Phase Reset action is defined and added to the HVI Action Collection so that it can be executed from the HVI sequence using the ActionExecute HVI instruction. By executing the channel phase reset from HVI, you can ensure that AWG channels on independent instrument are in-phase.

Python

```
def define_hvi_actions(sys_def, module_dict, config):
    """
    Define Channel Phase Reset Actions for M320x AWGs and AWG Trigger actions for M5300 RF AWGs
    """
    for engine_name, module in zip(module_dict.keys(), module_dict.values()):
        if compare_engine_type(engine_name, config.rf_awg_engine_name):
            # The phase of each M5300 RF AWG is reset by using the ABSOLUTE phase mode
            # AWG Trigger actions are necessary to trigger AWG pulses
            for ch_index in range(1, module_dict[engine_name].num_channels + 1):
                # Actions need to be added to the engine's action list so that they can be
executed
                action_name = config.awg_trigger_action_name + str(ch_index) # arbitrary user-
defined name
                instrument_action = "awg{}_trigger".format(ch_index) # name decided by
instrument API
                action_id = getattr(module_dict[engine_name].hvi.actions, instrument_action)
                # Add AWG Trigger actions for each instrument channel
                sys_def.engines[engine_name].actions.add(action_id, action_name)
        else:
            # The phase of all M320x AWGs channels are reset synchronously from HVI
            # This way AWG channels on different instruments are ensured to be in-phase
            # For each engine, add each HVI Actions to be executed to its own HVI Action
Collection
            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.ch1_reset_
phase, 'AWG1_trigger')
                action_name = config.awg_phase_reset_name+ str(ch_index) # arbitrary user-
defined name
                instrument_action = "ch{}_reset_phase".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 Sequences

HVI sequences can be programmed using the *Sequencer* class. HVI starts the execution through a global sequence (defined by the *SyncSequence* class) that takes care of synchronizing and encapsulating the local sequences corresponding to each HVI engine included in the application. In this programming example, the HVI global sync sequence contains two nested Sync While loops implemented using the *SyncWhile* API class.

Python

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


def program_rf_sweeps(sequencer, module_dict, config):
    """
    Programs the nested loops for RF sweeps
    """
    # Define HVI registers
    define_registers(sequencer, module_dict, config)
    # Define sync while condition
    freq_counter = sequencer.sync_sequence.scopes[config.leader_engine_name].registers
[config.freq_counter]
    sync_while_condition = kthvi.Condition.register_comparison(freq_counter,
kthvi.ComparisonOperator.LESS_THAN, config.num_freqs)
    # Add a Sync While
    sync_while = sequencer.sync_sequence.add_sync_while("Frequency Sweep", 90, sync_while_
condition)
    # Program Freq Sweep
    program_freq_sweep(sync_while.sync_sequence, module_dict, 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

```
def define_registers(sequencer, module_dict, config):
    """
    Defines all registers for each HVI engine
    """
    for engine_name in module_dict.keys():
        # Define registers for engines in AWG-type instruments
        if compare_engine_type(engine_name, config.awg_engine_name):
            freq_value = sequencer.sync_sequence.scopes[engine_name].registers.add(config.freq_
value_name, kthvi.RegisterSize.LONG)
            freq_value.initial_value = module_dict[engine_name].instrument.freqToInt
(config.freq_start_value)
            ampl_value = sequencer.sync_sequence.scopes[engine_name].registers.add(config.ampl_
value_name, kthvi.RegisterSize.SHORT)
            ampl_value.initial_value = module_dict[engine_name].instrument.voltsToInt
(config.ampl_start_value)
        # Define registers for engines in RF AWG-type instruments
        elif compare_engine_type(engine_name, config.rf_awg_engine_name):
            ampl_value = sequencer.sync_sequence.scopes[engine_name].registers.add(config.ampl_
value_name, kthvi.RegisterSize.SHORT)
            ampl_value.initial_value = module_dict[engine_name].instrument.channels[0].convert_
amplitude_to_int(config.rf_awg_ampl_start_value)
        # Define additional registers in the leader engine
        if engine_name==config.leader_engine_name:
            freq_counter = sequencer.sync_sequence.scopes[engine_name].registers.add
(config.freq_counter_name, kthvi.RegisterSize.LONG)
            freq_counter.initial_value = 0
            ampl_counter = sequencer.sync_sequence.scopes[engine_name].registers.add
(config.ampl_counter_name, kthvi.RegisterSize.SHORT)
            ampl_counter.initial_value = 0
```

Synchronized While (a)

This corresponds to statements (a) in the HVI diagram. Synchronized While (Sync While) statements belong to the set of HVI Sync Statements and are defined by the API class *SyncWhile*. A Sync While allows you to synchronously execute multiple local HVI sequences until a user-defined condition is met, that is, the sync while condition. Please note that for local sequences to be defined within the Sync While, it is necessary to use synchronized multi-sequence blocks. The duration of each iteration of the Sync While loop can be set using the *Duration* property and the *Time* class. Please note that the duration cannot be set to a deterministic quantity if the Sync While contains any flow control statement, i.e. If, While, Wait or WaitTime statements. Please consult Chapter 7 of the KS2201A User Manual for further information.

Python

```
# Define sync while condition
freq_counter = sequencer.sync_sequence.scopes[config.leader_engine_name].registers[config.freq_
counter]
sync_while_condition = kthvi.Condition.register_comparison(freq_counter,
kthvi.ComparisonOperator.LESS_THAN, config.num_freqs)
```

```
# Add a Sync While
sync_while = sequencer.sync_sequence.add_sync_while("Frequency Sweep", 90, sync_while_condition)
# Program Freq Sweep
program_freq_sweep(sync_while.sync_sequence, module_dict, config)
```

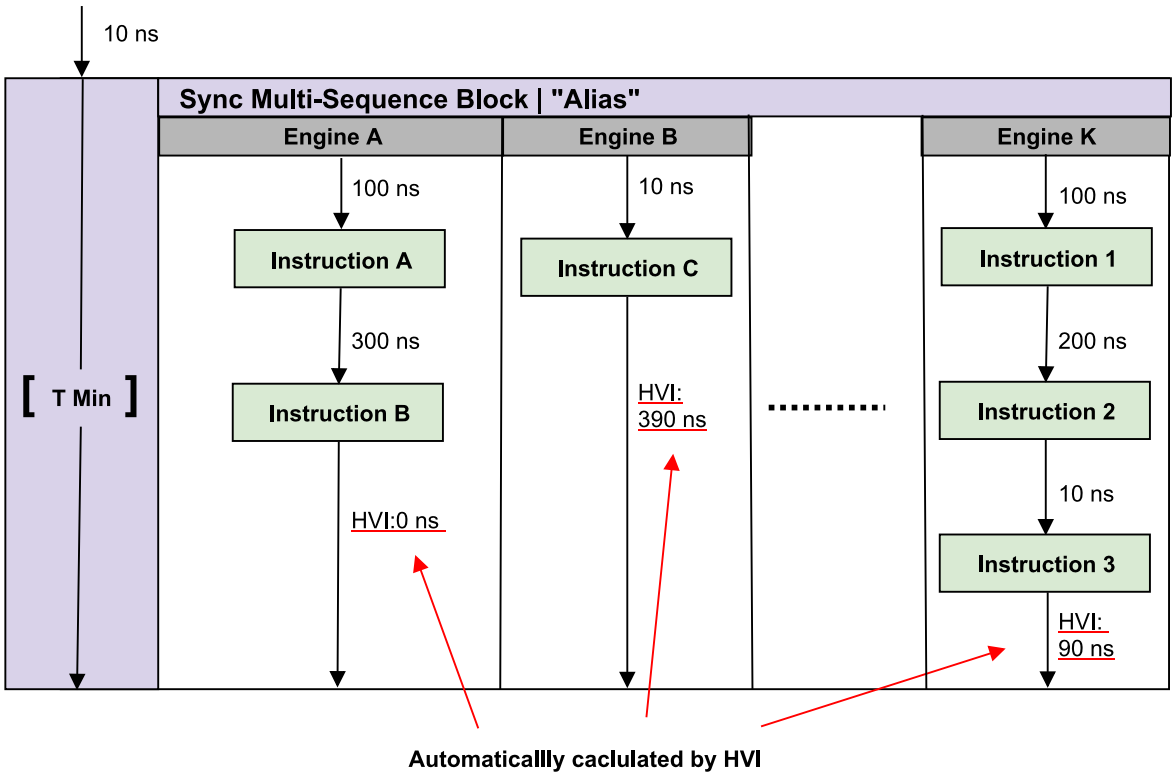
Synchronized Multi-Sequence Block (b)

This block synchronizes all the HVI engines that are part of the sync sequence and 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. The duration of the Sync Multi-Sequence Block (SMSB) can be set using the *Duration* property and the *Time* class. In this example, the SMSB duration is set to minimum, which means that the SMSB will last according to the start delays specified by the user for each statement programmed into the local sequences contained in it. Please note that the duration cannot be set to a deterministic quantity if the SMSB contains any flow control statement, i.e. If, While, Wait or WaitTime statements. Please consult Chapter 7 of the KS2201A User Manual for further information.

Python

```
# Add Sync Multi Sequence Block
sync_block = sync_sequence.add_sync_multi_sequence_block("Change Frequency", 510)
# Set SMSB duration
sync_block.duration = kthvi.time.Minimum()
# Programs the SMSB
program_change_freq(sync_block, module_dict, config)
```

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 synchronously start and end their execution within the sync multi-sequence block. Users can define the exact time when 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" is the default value for any sync statement, it corresponds to the minimum time necessary to complete the operations included inside. KS2201A Update 1.0 release provides the *Duration* property in Sync Statement objects that allows users to set an arbitrary duration value larger than "T Min". The timing at the end of each local sequence is automatically adjusted by HVI according to the duration specified by the user for the SMSB. In the case of duration "T min", HVI will automatically add no time to the local sequence with the longest duration and adjust the other sequences accordingly, as in the example depicted in the figure above. The resolution for HVI-defined time adjustment at the end of a sync multi-sequence block corresponds to the 10 ns FPGA clock period for an application including instruments that are all within the Keysight M3xxx family. For further explanations about the timing of HVI sequence execution, please refer to the KS2201A PathWave Test Sync Executive User Manual available on www.keysight.com

HVI Native Instruction: Action-Execute (c)

Actions to be used within an HVI sequence need to be added to the instrument HVI engine using the API "add" method of the *ActionCollection* class. Once the wanted actions are added within the list of the instruments' HVI engine actions, an instruction to execute them can be added to the instrument's

HVI sequence using the HVI API class *InstructionsActionExecute*. One or multiple actions can be executed at the same time within the same "Action Execute" instruction.

Python

```
# Retrieve the engine sequence from the collection of HVI Local Sequences
# HVI Local Sequence Collection is automatically created form the user-defined HVI Engine
Collection
# Each HVI Local Sequence can be retrieved using the name alias of the corresponding HVI Engine
sequence = sync_block.sequences[engine_name]
# Retrieve the module
module = module_dict[engine_name]
# Execute Phase reset
instruction = sequence.add_instruction("Reset Phase All Channels", 10, sequence.instruction_
set.action_execute.id)
instruction.set_parameter(sequence.instruction_set.action_execute.action.id,
sequence.engine.actions)
```

HVI Instrument-Specific Instruction (d)

Statements (c) of the HVI diagram implement instrument-specific HVI instructions. Such statements execute instructions that are specific of the instrument and can be used for example to set the amplitude or frequency of an AWG channel. Native HVI instructions are common to every Keysight product. The HVI 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 M3xxx instruments can be found in the M3xxx User Guide available on www.keysight.com.

Python

```
# Set Channel Frequency
freq_value = sequence.scope.registers[config.freq_value]
instruction = sequence.add_instruction("Set Channel Frequency", 10, module.hvi.instruction_
set.set_frequency.id)
instruction.set_parameter(module.hvi.instruction_set.set_frequency.channel.id, config.awg_
channel)
instruction.set_parameter(module.hvi.instruction_set.set_frequency.value.id, freq_value)
```

HVI Native Instruction: Register Increment (e)

This corresponds to statements (d) in the HVI diagram. A register increment can be implemented within an HVI sequence using an instance of the API instruction class *InstructionsAdd*. The same instruction can be used to add registers and constant values (operands) and put the result in another register (result). The register to be incremented needs to be added previously to the scope of the corresponding HVI engine.

Python

```
# Increment freq_value
freq_value = sequence.scope.registers[config.freq_value]
instruction = sequence.add_instruction("Increment Frequency Value", 10, sequence.instruction_
set.add.id)
instruction.set_parameter(sequence.instruction_set.add.destination.id, freq_value)
instruction.set_parameter(sequence.instruction_set.add.left_operand.id, freq_value)
instruction.set_parameter(sequence.instruction_set.add.right_operand.id,
module.instrument.freqToInt(config.freq_step))
```

HVI Native Instruction: Register Assign (f)

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

Python

```
# Initialize ampl_counter = 0
ampl_counter = sequence.scope.registers[config.ampl_counter]
instruction = sequence.add_instruction("Initialize Amplitude Counter", 10, sequence.instruction_
set.assign.id)
instruction.set_parameter(sequence.instruction_set.assign.destination.id, ampl_counter)
instruction.set_parameter(sequence.instruction_set.assign.source.id, 0)
```

Delay Statement (g)

This type of statement can be found in statements (f). 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 you to specify a delay before a statement, whereas the delay statement allows you 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, use the WaitTime statement instead.

Python

```
# Delay statement
sequence.add_delay("Loop Delay", 50)
```

Export the Programmed HVI Sequences to Text Format

KS2201A provides a feature to export the programmed HVI sequences to text format, which can be used both as a development and debug tool. The sequences can be exported using the to_string() method of the SyncSequence class, as illustrated in the code snippet below. Once exported to text format, the HVI sequences can be written to a text file or displayed on the console output. An

example text file containing the HVI sequences exported from this programming example is provided together with this example's files.

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

Compile, Load, Execute the HVI Instance

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

Compile HVI

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

Python

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

Load HVI to Hardware

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

Python

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

Execute HVI

HVI execution is controlled by the run() API method. HVI can be run in a blocking or non-blocking mode. In this programming example, the blocking mode is used. In this mode, the SW execution is blocked at the HVI execution code line for a fixed amount of time specified by the timeout input parameter. The SW execution can be blocked until the HVI sequences finish their execution if timeout = hvi.no_timeout is used as an input parameter.

Python

```
# Execute HVI in blocking mode: SW waits until HVI sequences ends their execution
# Eventually enter a timeout for the HVI execution to be stopped: timeout = timedelta
(seconds=0), hvi.run(timeout)
```

```
hvi.run(hvi.no_timeout)
print("HVI Running...")
```

Release Hardware

API method release_hw() shall be called once the HVI execution is finished to release all the HW resources that were reserved during the HVI execution, including the PXI trigger resources that had been locked by HVI for its execution.

Python

```
# Unlock and release HW resources
hvi.release_hw()
print("Releasing HW...")
```

Further HVI API Explanations

Detailed explanations of each class and functionality of the HVI API can be found in the PathWave Test Sync Executive User Manual or in the Python help file that is provided with the HVI installer, available at: C:\Program Files\Keysight\PathWave Test Sync Executive <year>\api\python\Help\index.html, where <year> shall be replaced with the year of the release you are using, for example <year> = 2023.

Conclusions

In this programming example, PathWave Test Sync Executive was used to define a real-time algorithm to be executed by the FPGA (Field Programmable Gate Array) of Arbitrary Waveform Generators (AWGs) from the Keysight M3xxx and the M5xxx PXIe families. HVI registers and instructions were used to update teal-time the amplitude and frequency of AWG channels. This enabled real-time RF sweeps on the AWG channels, to perform a pulsed characterization of a Device-Under-Test.

Edition 2023_U0_00, May, 2023
Printed in USA

KS2201-90007
www.keysight.com