PROGRAMMING EXAMPLE 3

PathWave Test Sync Executive Integration with PathWave FPGA

PATHWAVE

In this programming example we show how to establish a communication between a sequence of real-time instruction designed using PathWave Test Sync Executive and a custom FPGA (Field Programmable Gate Array) design integrated into the sandbox of a Keysight instrument using Keysight PathWave FPGA software.

KEYSIGHT TECHNOLOGIES

Table of Contents

KS2201A - Programming Example 3 - PathWave Test Sync Executive Integration with PathWave FPGA

In this programming example we show how to establish a communication between a sequence of real-time instruction designed using PathWave Test Sync Executive and a custom FPGA (Field Programmable Gate Array) design integrated into the sandbox of a Keysight instrument using Keysight PathWave FPGA software.

System Setup

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

System Requirements

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

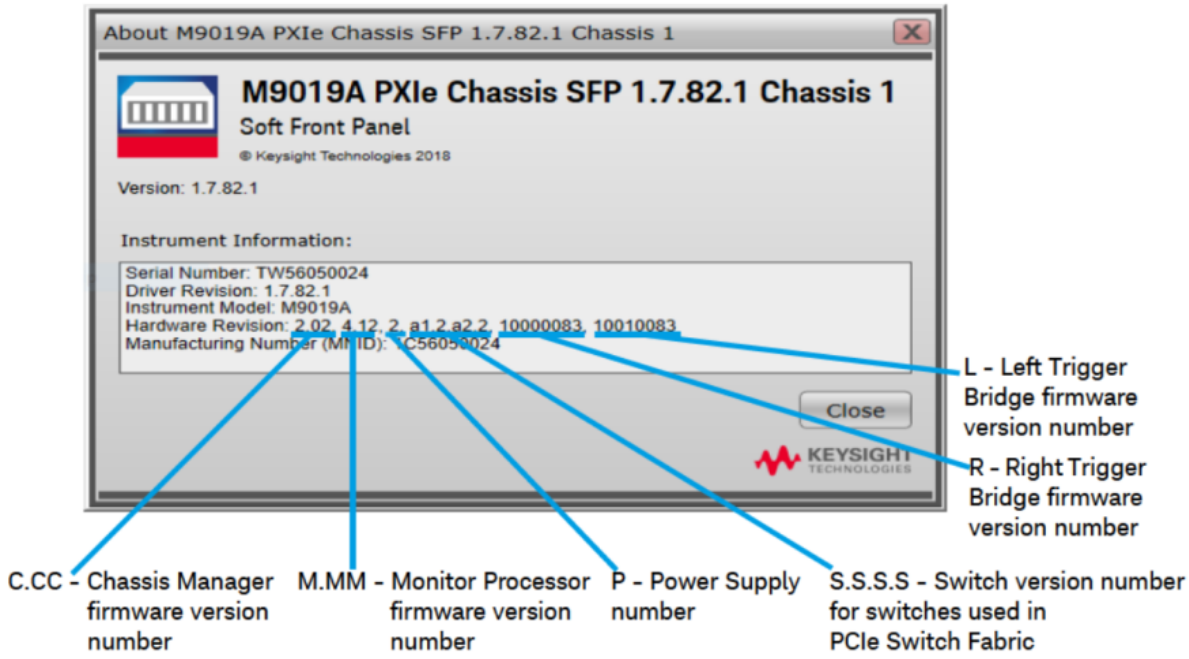
1. Software versions required:
   - Keysight IO Libraries Suite 2020 (v18.1.25310.1 or later)
   - Keysight SD1 Drivers, Libraries and SFP (v3.00.95 or later)
   - Keysight PathWave Test Sync Executive Update 0.2 (v1.00.18 or later)
   - Keysight PathWave FPGA 2020 Update 1.0
2. Chassis firmware and 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)

How to install Python 3.7.x 64-bit

This programming example requires you to install Python 64-bit version 3.7.x for all users. The Python installer can be downloaded from the Python webpage. Make sure you add Python 3.7.x to the PATH system Variable. This can be done at the installation step by checking the right check-boxes as shown in the screenshot below.

How to Install Chassis Driver, SFP and Firmware

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



M9019A Firmware Version Components

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

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

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

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

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

How to run this programming example

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

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

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

Each PXI instrument is described in the code using a module description class that contains the module model number, chassis number, slot number and options. Chassis and slot number in the code snippet below need to be updated before running the programming example:

```
# Previously defined engine Names
hvi_eng_Names = HVI_engine_Names()
# Update module descriptors below with your instruments information
module_descriptors = [
    module_descriptor('M3202A', 2, 4, options, hvi_eng_Names.primary_engine),
    module_descriptor('M3202A', 2, 10, options, hvi_eng_Names.secondary_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 for synchronization, data sharing, etc
# NOTE: In a multi-chassis setup ALL the PXI lines listed below need to be shared among
# each M9031 board pair by means of SMB cable connections
pxi_sync_trigger_resources = [
    kthvi.TriggerResourceId.PXI_TRIGGER0,
    kthvi.TriggerResourceId.PXI_TRIGGER1,
    kthvi.TriggerResourceId.PXI_TRIGGER2,
    kthvi.TriggerResourceId.PXI_TRIGGER3]
```

PXI lines allocated to be used as HVI trigger resources cannot be used by the programming example for other purposes. In this programming example PXI lines 4-7 are used to exchange information between primary and secondary modules through the instrument FPGA sandbox. Therefore, PXI lines 4-7 cannot be added as HVI PXI trigger resources in the code snippet above.

PathWave Test Sync Executive Integration with PathWave FPGA

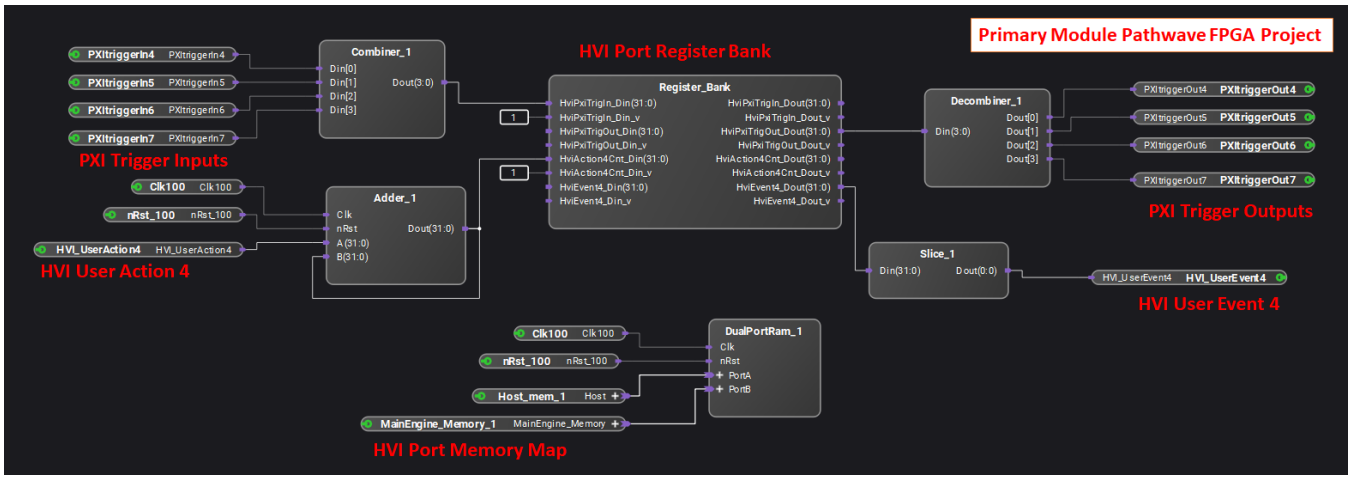
This programming example illustrates the following functionalities:

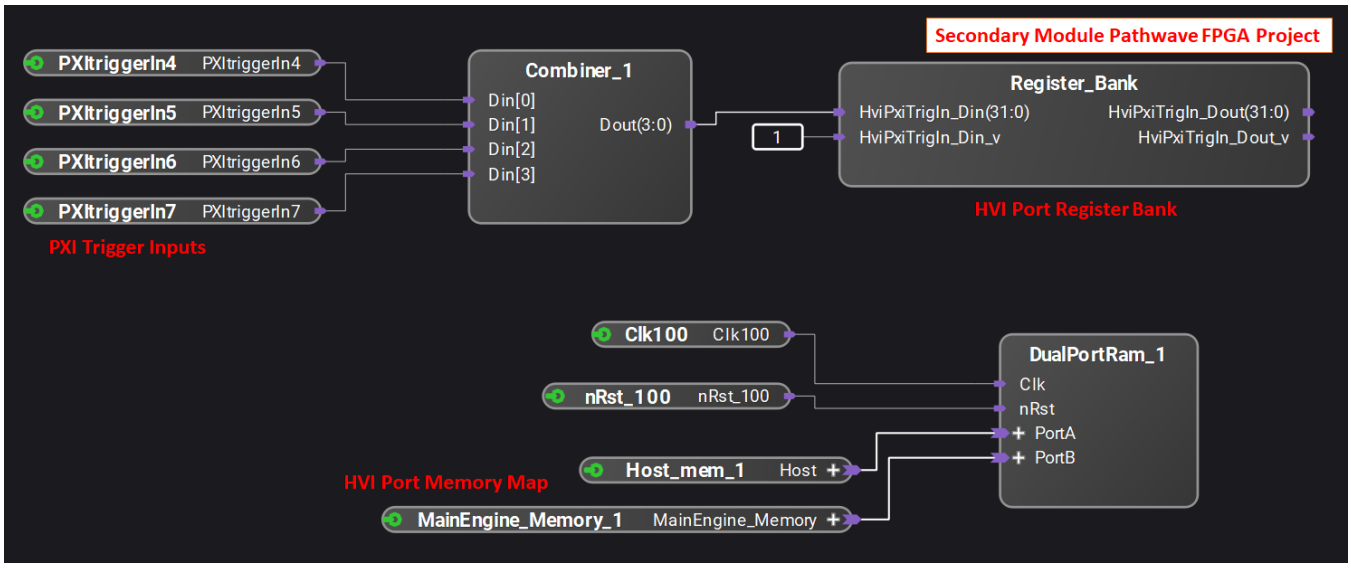
1. Read/write data from/to an HVI sequence to/from an HVI Memory Map inserted in an instrument FPGA sandbox
2. Read/write data from/to an HVI sequence to/from an HVI Register bank inserted in an instrument FPGA sandbox
3. Read/write PXI line values through instrument FPGA sandbox
4. Usage of HVI Actions and Events to communicate with an instrument FPGA sandbox

These functionalities are implemented using the combination of **Keysight PathWave Test Sync Executive** and **Keysight PathWave FPGA** software.

PathWave FPGA Project

This programming example is based on the implementation of custom blocks within the FPGA sandbox of both the primary and secondary module. The pictures below illustrate the PathWave FPGA projects for the primary module and secondary module respectively.





In the pictures above we can distinguish the following blocks:

- **HVI Memory Map:** this block allows to exchange data between an HVI sequence and an instrument FPGA sandbox by using a serial interface based on reading/writing data arrays
- **HVI Register Bank:** this block allows to exchange data between an HVI sequence and an instrument FPGA sandbox by reading/writing any of the register in the bank
- **PXI Trigger I/O**: these ports allow to read/write the PXI line on the chassis backplane from the FPGA sandbox of an M3xxxA instrument

- **HVI User Action**: actions are signals sent from an instrument HVI engine to the outside (the FPGA sandbox in this case). They can be associated to a PXI line, an internal/external trigger, or any of the product-defined actions
- **HVI User Event**: events are signals sent from the outside (the FPGA sandbox in this case) to an instrument HVI engine. They can be associated to a PXI line, an internal/external trigger, or any of the product-defined events

PathWave FPGA project files provided with this programming example are targeting M3202A AWG model. However, projects can be easily adapted to target different M3xxxA PXI instruments. This re-targeting functionality is explained in the **PathWave FPGA User Guide**. For a complete overview of Keysight PathWave FPGA and more information about all its functionalities please visit www.keysight.com.

PathWave Test Sync Executive Measurement Results

When this application Python code correctly executes, it shows a list of registers and memory blocks that are loaded to FPGA sandbox of both primary and secondary engines when loading the .k7z files generated by compiling the PathWave FPGA projects described in the previous sub-section of this document. Afterwards, the HVI sequence starts to execute and waits for the user to trigger a user event and execute a user action (user action 4) each time the user hits the enter key. The executed FPGA sandbox actions are counted at each iteration. Another counter starting from 1000 is incremented and read back after writing it to a dual port RAM. The user action counter value is written to PXI lines value so that it can also be read by the secondary module.

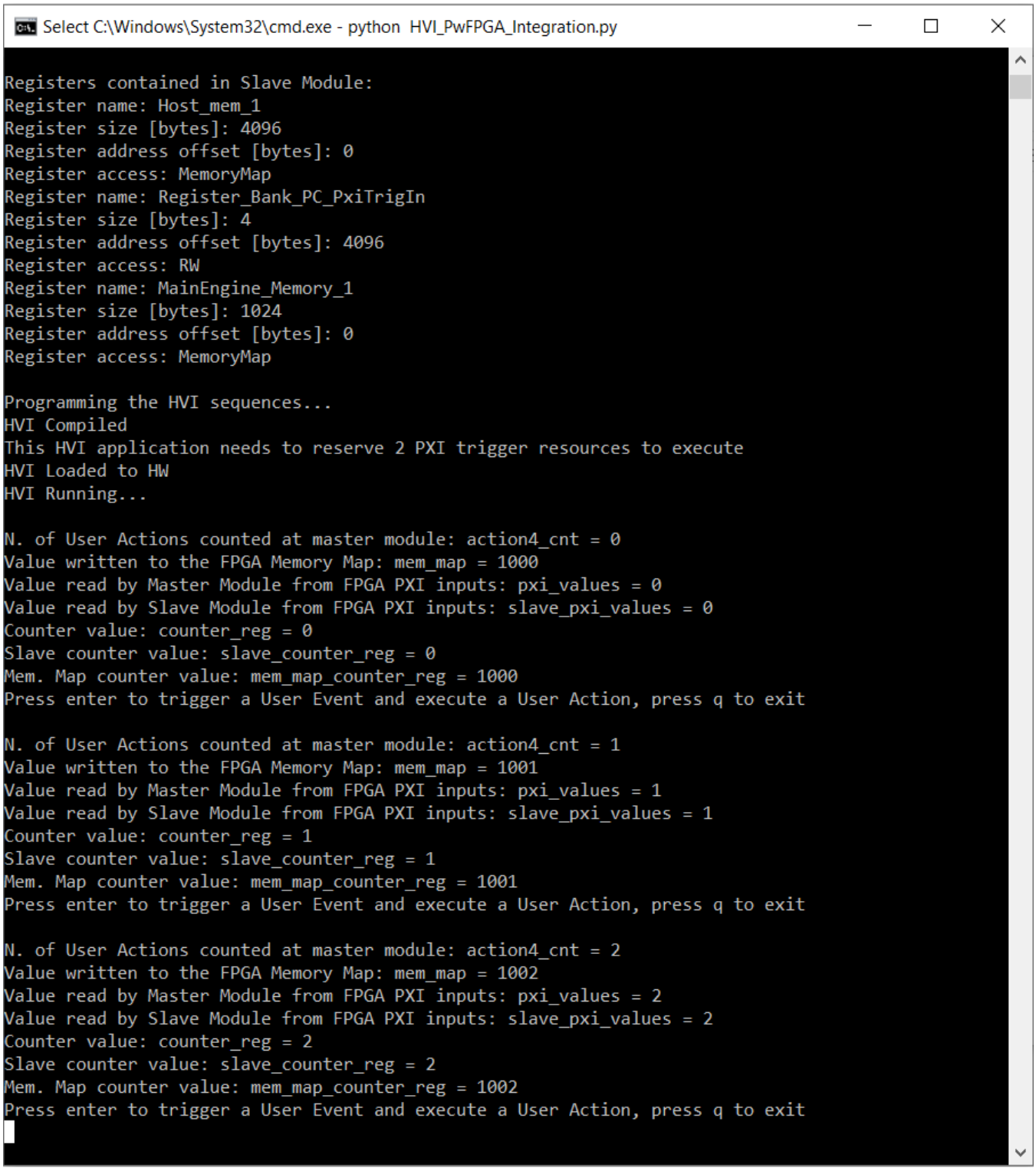
User events and actions available in an instrument FPGA sandbox depends on the specific instrument capability and are documented in the instrument documentation and user guides. In particular, documentation of user action 4 and user event 4 used in this programming example (represented by blocks "HVI_UserAction4" and "HVI_UserEvent4" in the primary module PathWave FPGA project) can be found in the M32xxA Arbitrary Waveform Generators User's Guide.

A more detailed programming example execution is described as follows. Within the Sync Multi-Sequence Block (SMSB) 'FPGA Read/Write Operations' all the four type of possible read/write operation to/from an FPGA sandbox register or memory map are performed. HVI register and HVI memory maps are part of the PathWave FPGA blockset "RealTime HVI" and they are described in details in the PathWave FPGA User Manual. The first statements reads a register in the FPGA sandbox (register 'Register_Bank_HviAction4Cnt' in the PathWave FPGA project) that is connected to a counter of user action 4 instances. The value is read into an HVI register Named 'Action4 Counter'. The subsequent FPGA write operation writes the user action 4 counter value into an FPGA register connected to PXI lines 4-7 outputs. This way the user action 4 counter value is written to PXI lines with a resolution of 4 bits. The following two statements validate the memory map read/write by first writing the value of a register counter called 'Memory Map Counter' into the memory map (block "MainEngine_ Memory1" in the PathWave FPGA projects) and then reading it back. The counter starts from 1000 and users can verify the counter value is written and read back correctly from the memory map during the example execution.

The next SMSB contains a register read operation in both local HVI sequences of primary and secondary instruments. Both primary and secondary modules have in their sandbox a register connected to PXI lines 4-7 inputs in the sandbox. This way both modules can read the PXI lines values through that register and hence can

read the user action 4 counter value that was previously written in those lines. The HVI sequence then waits for an user event 4 which can be generated by the user by pressing Enter from the console. Once the user event 4 is received the HVI sequence triggers an user action 4 instance that is counted by the counter register connected to the user action 4 input in the sandbox.

Finally in the last SMSB of the HVI sequence all the HVI register counters are incremented. The registers value increments are printed out on the console terminal at each iteration of the programming example. See screenshot below as an example of the programming example execution on the console terminal.

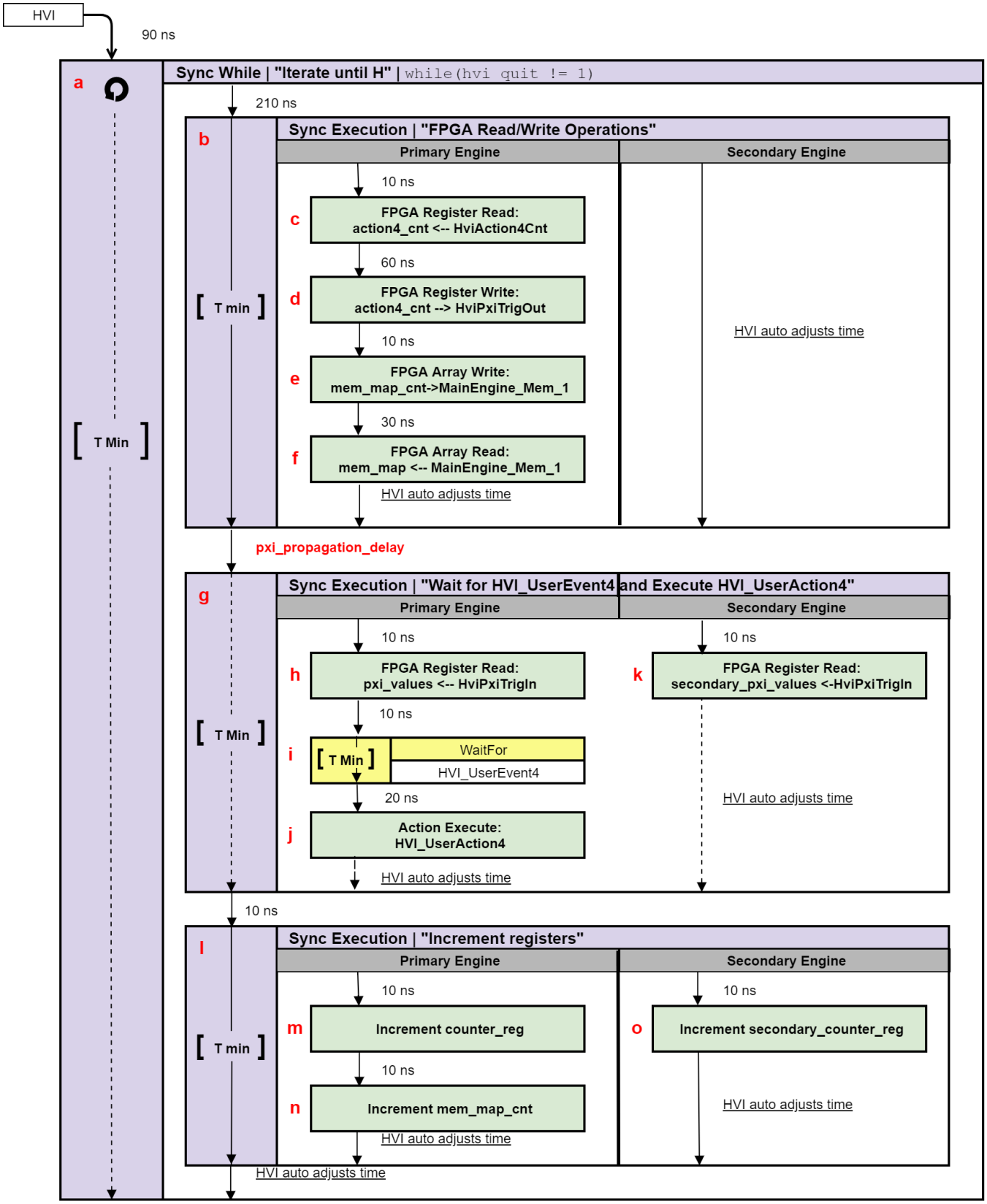


The example measurement results shown in the execution screenshot above can be measured on an oscillosope as well, by using an M9031A module to connect the PXI lines 4-7 to oscilloscope channels and

visualize their value update at each iteration of the programming example execution. The next section of this document provides further details about the HVI sequences executed and each HVI statement contained in them.

Getting Started with HVI Application Programming Interface (API)

PathWave Test Sync Executive implements the next generation of HVI technology and delivers the HVI Application Programming Interface (API). This section explains how to implement the use case of this programming example using HVI API. The sequence of operations executed by each of the instruments using HVI technology are represented 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 presented below with a letter referencing to the equivalent block in the HVI flowchart.



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

NOTE Python Variable *pxi_propagation_delay* is used to parametrize the start delay between the synchronized multi-sequence blocks "FPGA Read/Write operations" and "Wait for HVI_ UserEvent4 and Execute HVI_UserAction4". Thie *pxi_propagation_delay* is necessary to allow enough time for the Action4 counter register to write its value to the PXI lines, before the primary and secondary modules try to read that same value. This way we ensure the value read is up to date.

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.

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 Guide**. 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('MultiChassisSetup')
       #
    # 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 FPGA actions, events and other configurations
    define_fpga_resources(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 Guide**.

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.

Python

```
class HVI_engine_Names:
    # Defines the HVI engines and their Names
    def __init__(self):
        self.primary_engine = 'PrimaryEngine'
        self.secondary_engine = 'SecondaryEngine'



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 each AWG needs to trigger both a FP pulse and a waveform very precisely. To do that the AWG trigger actions are issued from within the HVI execution. In the HVI use model, actions need to be added to the action collection of each HVI engine before they can be executed. FP trigger needs to be added to the HVI Trigger Collection and configured. This is done in this programming example as explained in the code snippets below.

Python

```
class HVI_resource_Names:
    # Defines the HVI resources and their Names
    def __init__(self):
        # NOTE: The M3xxxA_sandbox Name is not arbitrary and cannot be changed.
        # The sandbox Name is defined by each instrument. See SD1 3.x M3xxxA documentation
for further info
        self.M3xxxA_sandbox = 'sandbox0'
        self.hvi_user_event_4 =  'FpgaUserEvent4'
        self.hvi_user_action_4 = 'FpgaUserAction4'

class FPGA_resources:
    # Defines the resources in the FPGA sandbox
    The FPGA resource Names are not arbitrary. They correspond to the Names defined in the
PathWave FPGA project files
    def __init__(self):
        self.primary_project_file = '../bitfiles/HviPortExamplePrimary.k7z'
        self.secondary_project_file = '../bitfiles/HviPortExampleSecondary.k7z'
        self.num_primary_regs = 6 # number of mem. maps and registers placed in the primary
PathWave FPGA project
        self.num_secondary_regs = 3 # number of mem. maps and registers placed in the
secondary PathWave FPGA project
```

```
        self.memory_map = 'MainEngine_Memory_1'
        self.reg_action4_cnt = 'Register_Bank_HviAction4Cnt'
        self.reg_event4 = 'Register_Bank_HviEvent4'
        self.reg_pxi_out = 'Register_Bank_HviPxiTrigOut'
        self.reg_pxi_in = 'Register_Bank_HviPxiTrigIn'
        self.secondary_reg_pxi_in = 'Register_Bank_HviPxiTrigIn'

def define_fpga_resources(sys_def, module_dict):
    # Define FPGA actions, events and other configurations
    # Load previously defined resources
    hvi_res_Names = HVI_resource_Names()
    hvi_eng_Names = HVI_engine_Names()
    fpga_resources = FPGA_resources()
       #
    # Primary module, secondary module
    primary_module = module_dict[hvi_eng_Names.primary_engine].instrument
    secondary_module = module_dict[hvi_eng_Names.secondary_engine].instrument
       #
    # Events: add FpgaUserEvent4 to the list of events of the primary engine
    fpga_user_event4 = primary_module.hvi.events.fpga_user_4
    sys_def.engines[hvi_eng_Names.primary_engine].events.add(fpga_user_event4, hvi_res_
Names.hvi_user_event_4)
       #
    # Actions: add FpgaUserAction4 to the list of actions of the primary engine
    fpga_user_action4 = primary_module.hvi.actions.fpga_user_4
    sys_def.engines[hvi_eng_Names.primary_engine].actions.add(fpga_user_action4, hvi_res_
Names.hvi_user_action_4)
       #
    # Get engine sandbox
    sandbox_Name = hvi_res_Names.M3xxxA_sandbox
    primary_sandbox = sys_def.engines[hvi_eng_Names.primary_engine].fpga_sandboxes[sandbox_
Name]
    secondary_sandbox = sys_def.engines[hvi_eng_Names.secondary_engine].fpga_sandboxes
[sandbox_Name]
    # Load to the sandboxes .k7z project created using Pathwave FPGA
    # This operation is necessary for HVI to list all the FPGA blocks contrined in the
designed FPGA FW
    primary_sandbox.load_from_k7z(fpga_resources.primary_project_file)
    secondary_sandbox.load_from_k7z(fpga_resources.secondary_project_file)
    #
    # Enable PXI lines to be written from the FPGA sandbox of primary engine only using
FPGATriggerOutConfig()
    # NOTE: Only one PXI module per segment shall be allowed to write backplane PXI lines.
It would cause conflicts and misbehavior to configure the PXI lines for the secondary
engine also
    primary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI4, direction =keysightSD1.SD_FpgaTriggerDirection.INOUT,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    primary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI5, direction =keysightSD1.SD_FpgaTriggerDirection.INOUT,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    primary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI6, direction =keysightSD1.SD_FpgaTriggerDirection.INOUT,
```

```
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    primary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI7, direction =keysightSD1.SD_FpgaTriggerDirection.INOUT,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
       #
    secondary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI4, direction =keysightSD1.SD_FpgaTriggerDirection.IN,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    secondary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI5, direction =keysightSD1.SD_FpgaTriggerDirection.IN,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    secondary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI6, direction =keysightSD1.SD_FpgaTriggerDirection.IN,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    secondary_module.FPGATriggerConfig(externalSource=keysightSD1.SD_
TriggerExternalSources.TRIGGER_PXI7, direction =keysightSD1.SD_FpgaTriggerDirection.IN,
polarity= keysightSD1.SD_TriggerPolarity.ACTIVE_LOW, syncMode = keysightSD1.SD_
SyncModes.SYNC_NONE, delay5Tclk=0)
    #
    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 HVI global sync sequence consists of a synchronized while statement containing three synchronized multi-sequence blocks.

Python

```
def program_hvi_sequence(sys_def):
    # 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)
       #
    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 register Names to be used within the scope of each HVI engine
    def __init__(self):
        self.hvi_quit = 'HVI Quit'
        self.action4_cnt = 'Action4 Counter'
        self.counter_reg = 'Loop Counter'
        self.mem_map = 'Memory Map Value'
        self.mem_map_counter = 'Memory Map Counter'
        self.pxi_values = 'PXI Values'
        self.secondary_pxi_values = 'Secondary PXI Values'
        self.secondary_counter_reg = 'Secondary Counter'



def define_registers(sequencer):
    # Defines all registers for each HVI engine in the scope af the global sync sequence
    # Load previously defined resource Names
    hvi_eng_Names = HVI_engine_Names()
    register_Names = HVI_register_Names()
       #
    # Define registers for primary engine
    hvi_quit = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_engine].registers.add
(register_Names.hvi_quit, kthvi.RegisterSize.SHORT)
    hvi_quit.initial_value = 0
    action4_cnt = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_
engine].registers.add(register_Names.action4_cnt, kthvi.RegisterSize.SHORT)
    action4_cnt.initial_value = 0
    counter_reg = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_
engine].registers.add(register_Names.counter_reg, kthvi.RegisterSize.SHORT)
    counter_reg.initial_value = 0
    mem_map = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_engine].registers.add
(register_Names.mem_map, kthvi.RegisterSize.SHORT)
    mem_map.initial_value = 0
    mem_map_counter = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_
engine].registers.add(register_Names.mem_map_counter, kthvi.RegisterSize.SHORT)
    mem_map_counter.initial_value = 1000
```

```
    pxi_values = sequencer.sync_sequence.scopes[hvi_eng_Names.primary_engine].registers.add
(register_Names.pxi_values, kthvi.RegisterSize.SHORT)
    pxi_values.initial_value = 0
    # Define registers for primary engine
    secondary_counter_reg = sequencer.sync_sequence.scopes[hvi_eng_Names.secondary_
engine].registers.add(register_Names.secondary_counter_reg, kthvi.RegisterSize.SHORT)
    secondary_counter_reg.initial_value = 0
    secondary_pxi_values = sequencer.sync_sequence.scopes[hvi_eng_Names.secondary_
engine].registers.add(register_Names.secondary_pxi_values, kthvi.RegisterSize.SHORT)
    secondary_pxi_values.initial_value = 0
       #
    return
```

Synchronized While

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

Python

```
# Define sync while condition
sync_while_condition = kthvi.Condition.register_comparison(hvi_quit,
kthvi.ComparisonOperator.NOT_EQUAL_TO, 1)
# Add Sync While Statement
sync_while = sync_sequence.add_sync_while('User-controlled sync loop', 10, sync_while_
condition)
```

```
def program_sync_while(sync_sequence):
    # Adds and programs the outmost Sync While statement of the HVI Sync Sequence
    # Load previously defined resource Names
    hvi_eng_Names = HVI_engine_Names()
    register_Names = HVI_register_Names()
    #Previously defined registers
    hvi_quit = sync_sequence.scopes[hvi_eng_Names.primary_engine].registers[register_
Names.hvi_quit]
       #
    # Define sync while condition
    sync_while_condition = kthvi.Condition.register_comparison(hvi_quit,
kthvi.ComparisonOperator.NOT_EQUAL_TO, 1)
    # Add Sync While Statement
    sync_while = sync_sequence.add_sync_while('User-controlled sync loop', 60, sync_while_
condition)
       #
    # Add and program 1st Sync Multi-Sequence Block
    program_sync_block_1(sync_while.sync_sequence)
       #
    # Add and program 2nd Sync Multi-Sequence Block
    program_sync_block_2(sync_while.sync_sequence)
       #
    # Add and program 3rd Sync Multi-Sequence Block
    program_sync_block_3(sync_while.sync_sequence)
```

```
        #
    return
```

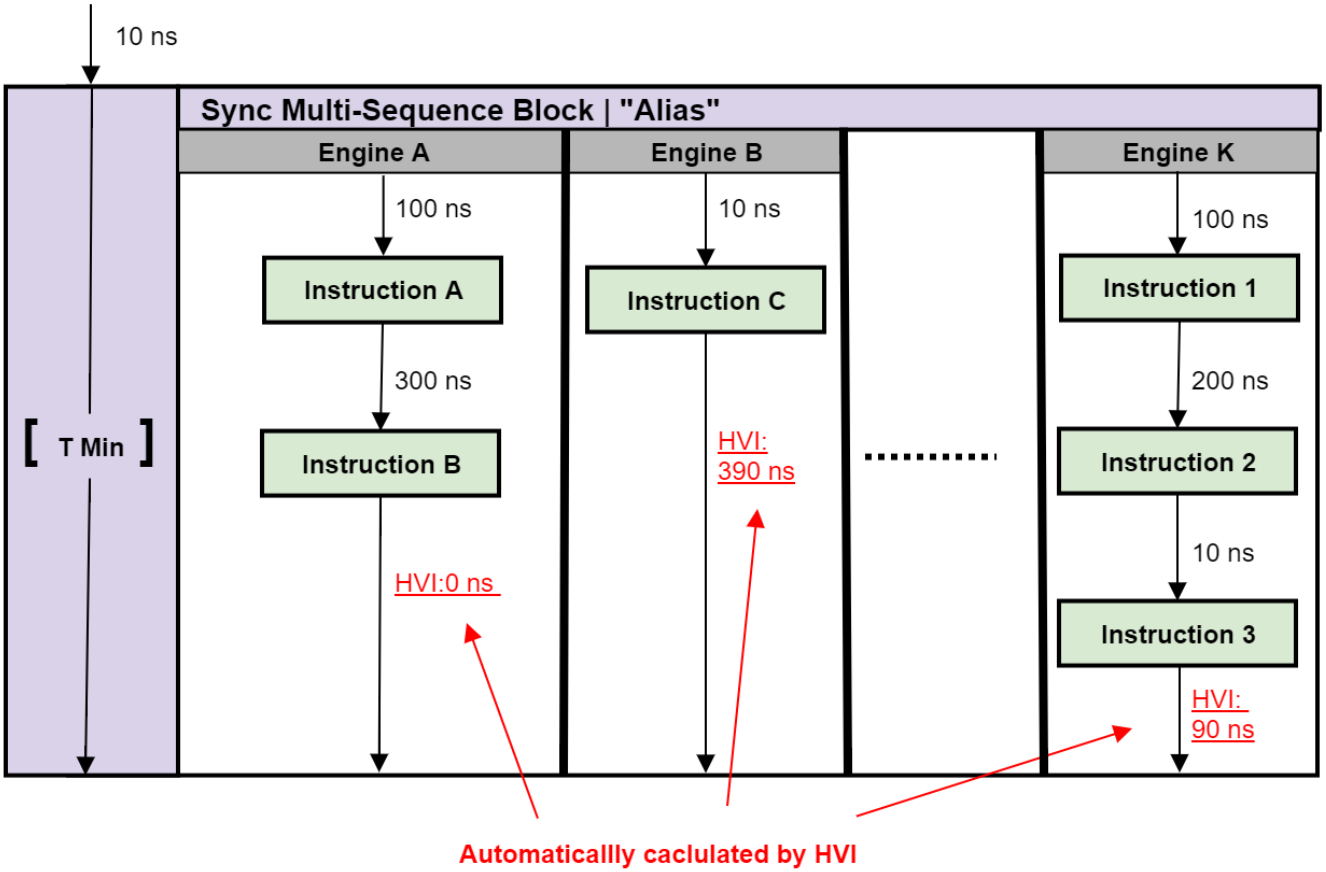
Synchronized Multi-Sequence Block

It corresponds to statements (b, g, l) in the HVI diagram. Synchronized multi-sequence blocks are defined by the API class *SyncMultiSequenceBlock.* This type of sync statement synchronizes all the HVI engines that are part of the sync sequence. It allows you to program each HVI Engine to do specific operations by exposing a local sequence for each engine. By calling the API method *add_multi_sequence_block()* a synchronized multi-sequence block is added to the Sync (global) Sequence.

Python

```
# Add 1st Sync Multi-Sequence Block to the Sync While sequence
sync_block_1 = sync_sequence.add_sync_multi_sequence_block('FPGA Read/Write Operations',
210)
primary_sequence = sync_block_1.sequences[hvi_eng_Names.primary_engine]
```

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

FPGA Register Read

It corresponds to statements (c, h, k) in the HVI diagram. *InstructionFpgaRegisterRead* is an HVI core instruction that allows reading an HVI Register Bank placed into an FPGA sandbox design. The value read from the HVI Port Register will be written into a destination HVI register.

Python

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_1.sequences[hvi_eng_Names.primary_engine]
# Previously defined registers and FPGA resources
action4_cnt = sync_sequence.scopes[hvi_eng_Names.primary_engine].registers[register_
Names.action4_cnt]
fpga_reg_action4_cnt = primary_sequence.engine.fpga_sandboxes[hvi_res_Names.M3xxxA_
sandbox].fpga_registers[fpga_resources.reg_action4_cnt]



# Read FPGA Register Register_Bank_HviAction4Cnt
readFpgaReg0 = primary_sequence.add_instruction('Read FPGA Register_Bank_HviAction4Cnt',
10, primary_sequence.instruction_set.fpga_register_read.id)
readFpgaReg0.set_parameter(primary_sequence.instruction_set.fpga_register_
read.destination.id, action4_cnt)
readFpgaReg0.set_parameter(primary_sequence.instruction_set.fpga_register_read.fpga_
register.id, fpga_reg_action4_cnt)
```

FPGA Register Write

It corresponds to statement (d) in the HVI diagram. *InstructionFpgaRegisterWrite* is an HVI core instruction that allows writing an HVI Register Bank placed into an FPGA sandbox. The value to be written into the HVI Register Bank is taken from an HVI register or from a literal.

Python

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_1.sequences[hvi_eng_Names.primary_engine]
# Previously defined registers and FPGA resources
action4_cnt = sync_sequence.scopes[hvi_eng_Names.primary_engine].registers[register_
Names.action4_cnt]
fpga_reg_pxi_out = primary_sequence.engine.fpga_sandboxes[hvi_res_Names.M3xxxA_
sandbox].fpga_registers[fpga_resources.reg_pxi_out]



# Write FPGA Register Register_Bank_HviPxiTrigOut
# Register_Bank_HviPxiTrigOut is connected to PXI lines Outputs.
# The value written to the FPGA register will be written to PXI lines
# NOTE: Please allow at least 60 ns between these instructions to ensure
# the HVI register action4_cnt is updated before writing its content to PXI lines
writeFpgaReg0 = primary_sequence.add_instruction('Write FPGA Register_Bank_HviPxiTrigOut',
60, primary_sequence.instruction_set.fpga_register_write.id)
writeFpgaReg0.set_parameter(primary_sequence.instruction_set.fpga_register_write.fpga_
register.id, fpga_reg_pxi_out)
writeFpgaReg0.set_parameter(primary_sequence.instruction_set.fpga_register_write.value.id,
action4_cnt)
```

FPGA Memory Map Write

It corresponds to statement (e) in the HVI diagram. *InstructionFpgaArrayWrite* is an HVI core instruction that allows writing to an HVI Memory Map placed into an FPGA sandbox. The value to be written into the HVI Memory Map is taken from an HVI register or from a literal.

Python

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_1.sequences[hvi_eng_Names.primary_engine]
# Previously defined registers and FPGA resources
mem_map_counter = sync_sequence.scopes[hvi_eng_Names.primary_engine].registers[register_
Names.mem_map_counter]
fpga_memory_map = primary_sequence.engine.fpga_sandboxes[hvi_res_Names.M3xxxA_
sandbox].fpga_memory_maps[fpga_resources.memory_map]



# Write Memory Map
# At each iteration a different value is written to the memory map
writeMemoryMap = primary_sequence.add_instruction('Write FPGA Memory Map', 10, primary_
sequence.instruction_set.fpga_array_write.id)
writeMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_write.fpga_memory_
map.id, fpga_memory_map)
writeMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_write.value.id,
mem_map_counter)
writeMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_write.fpga_memory_
map_offset.id, 0)
```

FPGA Memory Map Read

It corresponds to statement (f) in the HVI diagram. *InstructionFpgaArrayRead* is an HVI core instruction that allows reading an HVI Memory Map. The value read from the HVI Memory Map will be written into a destination HVI register.

Python

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_1.sequences[hvi_eng_Names.primary_engine]
# Previously defined registers and FPGA resources
mem_map = sync_sequence.scopes[hvi_eng_Names.primary_engine].registers[register_Names.mem_
map]
fpga_memory_map = primary_sequence.engine.fpga_sandboxes[hvi_res_Names.M3xxxA_
sandbox].fpga_memory_maps[fpga_resources.memory_map]



# Read Memory Map
# Reads the value that was written to the block RAM connected to the memory map
# NOTE: Please allow at least 30 ns between these instructions to ensure data is written
# correctly through the memory map before you read it back
readMemoryMap = primary_sequence.add_instruction('Read FPGA Memory Map', 30, primary_
sequence.instruction_set.fpga_array_read.id)
readMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_read.fpga_memory_
map.id, fpga_memory_map)
readMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_
read.destination.id, mem_map)
```

```
readMemoryMap.set_parameter(primary_sequence.instruction_set.fpga_array_read.fpga_memory_
map_offset.id, 0)
```

Wait Statement

It corresponds to statement (i) in the HVI diagram. The wait statement is a local flow control statement that can be implemented using the API class *WaitStatement.* This sequence block sets an instrument to wait for a condition. The condition ca be defined by a trigger, an event, or any combination of them through the usage of logical operators. In this programming example, the wait statement is used to set the primary engine to wait for an event generated by the FPGA sandbox, more specifically the event called 'HVI_UserEvent4'. The wait condition is defined by the wait mode and the sync mode. The wait mode *.WaitMode.TRANSITION* makes sure the wait condition is triggered precisely at the time instant when the event is activated. The sync mode .SyncMode.IMMEDIATE sets the wait event statement to let the execution continue immediately, i.e. as soon as the event is received.

Python

```
# Wait for FPGA_User_Event4
# Define the condition for the wait statement
wait_condition = kthvi.Condition.event(hvi.engines[hvi_resources.primary_engine_
Name].events[hvi_resources.hvi_user_event_4])
# Add wait statement
primary_sequence = sync_block_2.sequences[hvi_eng_Names.primary_engine]
waitEvent = primary_sequence.add_wait('Wait for FPGA_User_Event4', 10, wait_condition)
waitEvent.set_mode(kthvi.WaitMode.TRANSITION, kthvi.SyncMode.IMMEDIATE)
```

Action Execute

It corresponds to statement (j) in the HVI diagram. Actions to be used within an HVI sequence need to be added to the instrument HVI engine using the API 'add' method of the *ActionCollection* class. Once the wanted actions are added within the list of the instruments' HVI engine actions, an instruction to execute them can be added to the instrument's HVI sequence using the HVI API class *InstructionsActionExecute*. One or multiple actions can be executed at the same time within the same 'Action Execute' instruction.

Python

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_2.sequences[hvi_eng_Names.primary_engine]
# Action execute instruction: execute action 4
instAction4 = primary_sequence.add_instruction('Execute Action 4', 20, primary_
sequence.instruction_set.action_execute.id)
instAction4.set_parameter(primary_sequence.instruction_set.action_execute.action.id,
primary_sequence.engine.actions[hvi_res_Names.hvi_user_action_4])
```

Register Increment

It corresponds to statements (m, n, o) 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

```
# AWG local sequences can be accessed from within the Sync Multi-Sequence Block
primary_sequence = sync_block_2.sequences[hvi_eng_Names.primary_engine]
#
# Increment counter register
instr = primary_sequence.add_instruction('Increment counter register', 10, primary_
sequence.instruction_set.add.id)
instr.set_parameter(primary_sequence.instruction_set.add.left_operand.id, counter_reg)
instr.set_parameter(primary_sequence.instruction_set.add.right_operand.id, 1)
instr.set_parameter(primary_sequence.instruction_set.add.destination.id, counter_reg)
```

Compile, Load, Execute the HVI

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

Compile HVI

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

Python

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

Load HVI to Hardware

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

Python

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

Execute HVI

HVI execution is controlled by the run() API method. HVI can be run in a blocking or non-blocking mode. In this programming example the non-blocking mode is used. By using this execution mode, SW execution can interact through registers read/write with the HVI sequence execution.

Python

```
# Execute HVI in non-blocking mode
# This mode allows SW execution to interact with HVI execution
hvi.run(hvi.no_wait)
print('HVI Running...')
```

Release Hardware

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

Python

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

Further HVI API Explanations

Detailed explanations of each class and functionality of the HVI API can be found in the PathWave Test Sync Executive User Manual or in the Python help file that is provided with the HVI installer, available at: C:\Program Files\Keysight\PathWave Test Sync Executive 2020\api\python\Help\index.htm.

Conclusions

This Programming Example illustrated how to use Keysight PathWave Test Sync Executive together with Keysight PathWave FPGA. Custom FPGA block are designed using Keysight PathWave FPGA and loaded to the sandbox of two modular instrument. The two instruments execute HVI sequences that can communicate with the custom FPGA blocks programmed into the sandbox of the module FPGA. Using an HVI Port the HVI sequence can read/write values to any HVI Port Register inserted among the custom FPGA blocks. This application note has also shown how HVI sequence and FPGA sandbox of an instrument can communicate by using actions and events. The exchanged information can also be written to PXI lines.